

Spring框架课堂笔记

1. Spring概述

1.1 什么是框架？

框架 (Framework)：框 (指其约束性) 架 (指其支撑性)，在软件设计中指为解决一个开放性问题而设计的具有一定约束性的支撑结构。在此结构上可以根据具体问题扩展、安插更多的组成部分，从而更迅速和方便地构建完整的解决问题的方案。

- 1、框架本身一般不完整到可以解决特定问题
- 2、框架天生就是为扩展而设计的
- 3、框架里面可以为后续扩展的组件提供很多辅助性、支撑性的方便易用的实用工具 (utilities)，也就是说框架时常配套了一些帮助解决某类问题的库 (libraries) 或工具 (tools)。

如何学习框架呢？

- 1、知道框架能做什么
- 2、学习框架的语法，一般框架完成一个功能需要一定的步骤
- 3、框架的内部实现原理 (扩展)
- 4、尝试实现一个框架 (提升)

1.2 Spring是什么

Spring官网 <https://spring.io>

Spring 被称为 J2EE 的春天，是一个分层的 Java SE/EE **full-stack 开源的轻量级**的 Java 开发框架，是最受欢迎的企业级 Java 应用程序开发框架，数以百万的来自世界各地的开发人员使用 Spring 框架来创建性能好、易于测试、可重用的代码。

Spring具有**控制反转** (IoC) 和**面向切面** (AOP) 两大核心。Java Spring 框架通过声明式方式灵活地进行**事务的管理**，提高开发效率和质量。

Spring 框架不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何 Java 应用都可以从 Spring 中受益。Spring 框架还是一个超级粘合平台，除了自己提供功能外，还提供粘合其他技术和框架的能力。

1.3 Spring的优势

- 1、方便解耦，简化开发
Spring 就是一个大工厂，可以将所有对象的创建和依赖关系的维护交给 Spring 管理。
- 2、方便集成各种优秀框架
Spring 不排斥各种优秀的开源框架，其内部提供了对各种优秀框架 (如 Struts2、Hibernate、MyBatis 等) 的直接支持。
- 3、降低 Java EE API 的使用难度
Spring 对 Java EE 开发中非常难用的一些 API (JDBC、JavaMail、远程调用等) 都提供了封装，使这些 API 应用的难度大大降低。

4、方便程序的测试

Spring 支持 JUnit4，可以通过注解方便地测试 Spring 程序。

5、AOP 编程的支持

Spring 提供面向切面编程，可以方便地实现对程序进行权限拦截和运行监控等功能。

6、声明式事务的支持

只需要通过配置就可以完成对事务的管理，而无须手动编程。

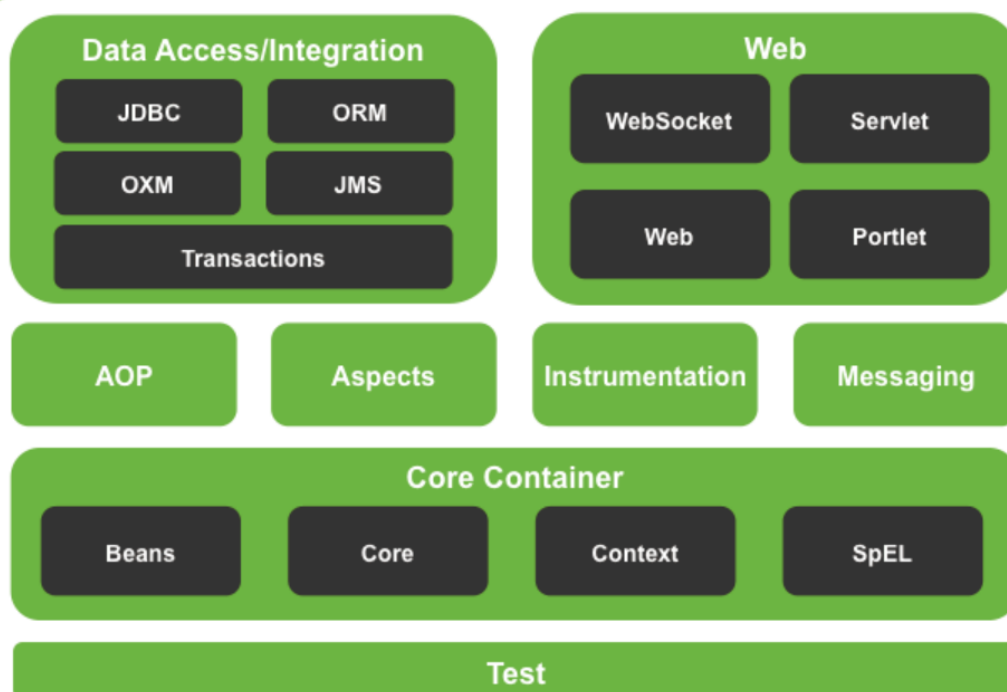
1.4 Spring的体系结构

Spring 为我们提供了一站式解决方案，但Spring 是模块化的，允许咱们挑选和选择适用于项目的模块，不需要把剩余部分也引入。

Spring 框架提供约 20 个模块，可以根据应用程序的要求来选择。



Spring Framework Runtime



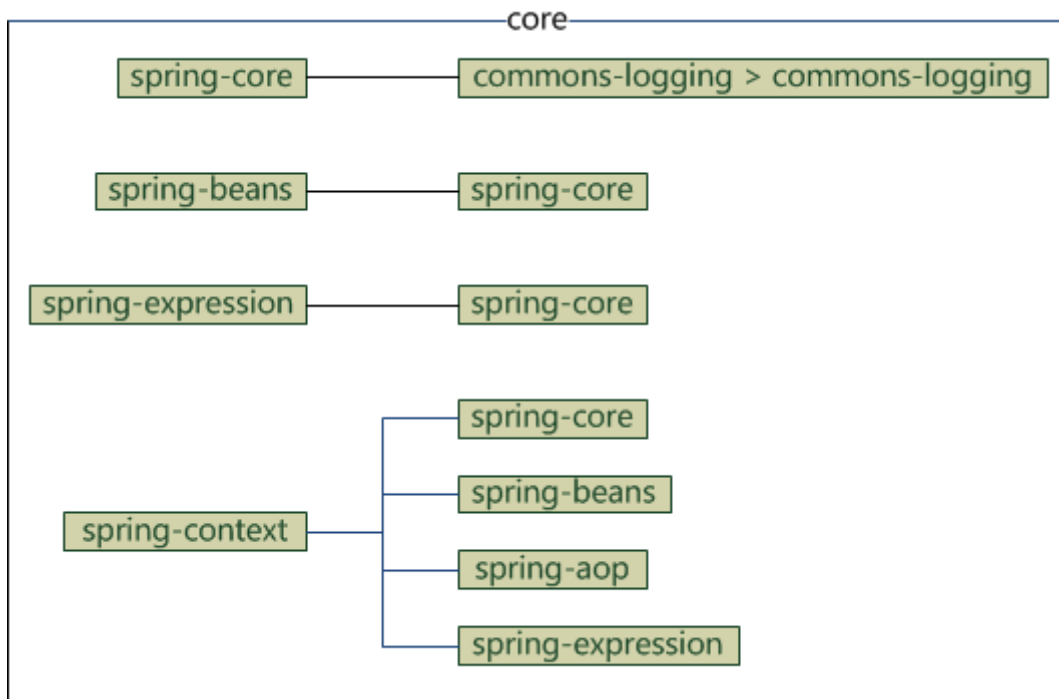
1.4.1 核心容器

核心容器由 **Spring-core**, **Spring-beans**, **Spring-context**, **Spring-context-support**和**Spring-expression** (SpEL, Spring 表达式语言, Spring Expression Language) 等模块组成

- **Spring-core** 模块提供了框架的基本组成部分，包括 IoC 和依赖注入功能。
- **Spring-beans** 模块提供 BeanFactory，工厂模式的微妙实现，它移除了编码式单例的需要，并且可以把配置和依赖从实际编码逻辑中解耦。
- **context** 模块建立在由 **core**和 **beans** 模块的基础上建立起来的，它以种类类似于 JNDI 注册的方式访问对象。Context 模块继承自 Bean 模块，并且添加了国际化（比如，使用资源束）、事件传播、资源加载和透明地创建上下文（比如，通过 Servlet 容器）等功能。Context 模块也支持 Java EE 的功能，比如 EJB、JMX 和远程调用等。**ApplicationContext** 接口是 Context 模块的焦点。**Spring-context-support** 提供了对第三方集成到 Spring 上下文的支持，比如缓存（EhCache, Guava, JCache）、邮件（JavaMail）、调度（CommonJ, Quartz）、模板引擎（FreeMarker, JasperReports, Velocity）等。
- **Spring-expression** 模块提供了强大的表达式语言，用于在运行时查询和操作对象图。它是 JSP2.1 规范中定义的统一表达式语言的扩展，支持 set 和 get 属性值、属性赋值、方法调用、访问数组集

合及索引的内容、逻辑算术运算、命名变量、通过名字从 Spring IoC 容器检索对象，还支持列表的投影、选择以及聚合等。

它们的完整依赖关系如下图所示：



1.4.2 数据访问/集成

JDBC=Java Data Base Connectivity, ORM=Object Relational Mapping, OXM=Object XML Mapping, JMS=Java Message Service

- **JDBC** 模块提供了 JDBC 抽象层，它消除了冗长的 JDBC 编码和对数据库供应商特定错误代码的解析。
- **ORM** 模块提供了对流行的对象关系映射 API 的集成，包括 JPA、JDO 和 Hibernate 等。通过此模块可以让这些 ORM 框架和 Spring 的其它功能整合，比如前面提及的事务管理。
- **OXM** 模块提供了对 OXM 实现的支持，比如 JAXB、Castor、XML Beans、JiBX、XStream 等。
- **JMS** 模块包含生产（produce）和消费（consume）消息的功能。从 Spring 4.1 开始，集成了 Spring-messaging 模块。
- **事务** 模块为实现特殊接口类及所有的 POJO 支持编程式和声明式事务管理。

1.4.3 Web

- **Web** 模块提供面向 web 的基本功能和面向 web 的应用上下文，比如多部分（multipart）文件上传功能、使用 Servlet 监听器初始化 IoC 容器等。它还包括 HTTP 客户端以及 Spring 远程调用中与 web 相关的部分。
- **Web-MVC** 模块为 web 应用提供了模型视图控制（MVC）和 REST Web服务的实现。Spring 的 MVC 框架可以使领域模型代码和 web 表单完全地分离，且可以与 Spring 框架的其它所有功能进行集成。
- **Web-Socket** 模块为 WebSocket-based 提供了支持，而且在 web 应用程序中提供了客户端和服务端之间通信的两种方式。
- **Web-Portlet** 模块提供了用于 Portlet 环境的 MVC 实现，并反映了 Spring-webmvc 模块的功能。

1.4.4 其他

- **AOP** 模块提供了面向方面（切面）的编程实现，允许你定义方法拦截器和切入点对代码进行干净地解耦，从而使实现功能的代码彻底的解耦出来。
- **Aspects** 模块提供了与 **AspectJ** 的集成，这是一个功能强大且成熟的面向切面编程（AOP）框架。
- **Instrumentation** 模块在一定的应用服务器中提供了类 instrumentation 的支持和类加载器的实现。
- **Messaging** 模块为 STOMP 提供了支持作为在应用程序中 WebSocket 子协议的使用。它也支持一个注解编程模型，它是为了选路和处理来自 WebSocket 客户端的 STOMP 信息。
- **测试** 模块支持对具有 JUnit 或 TestNG 框架的 Spring 组件的测试。

2、Spring核心之IoC控制反转

2.1 IoC的概念

IoC—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。

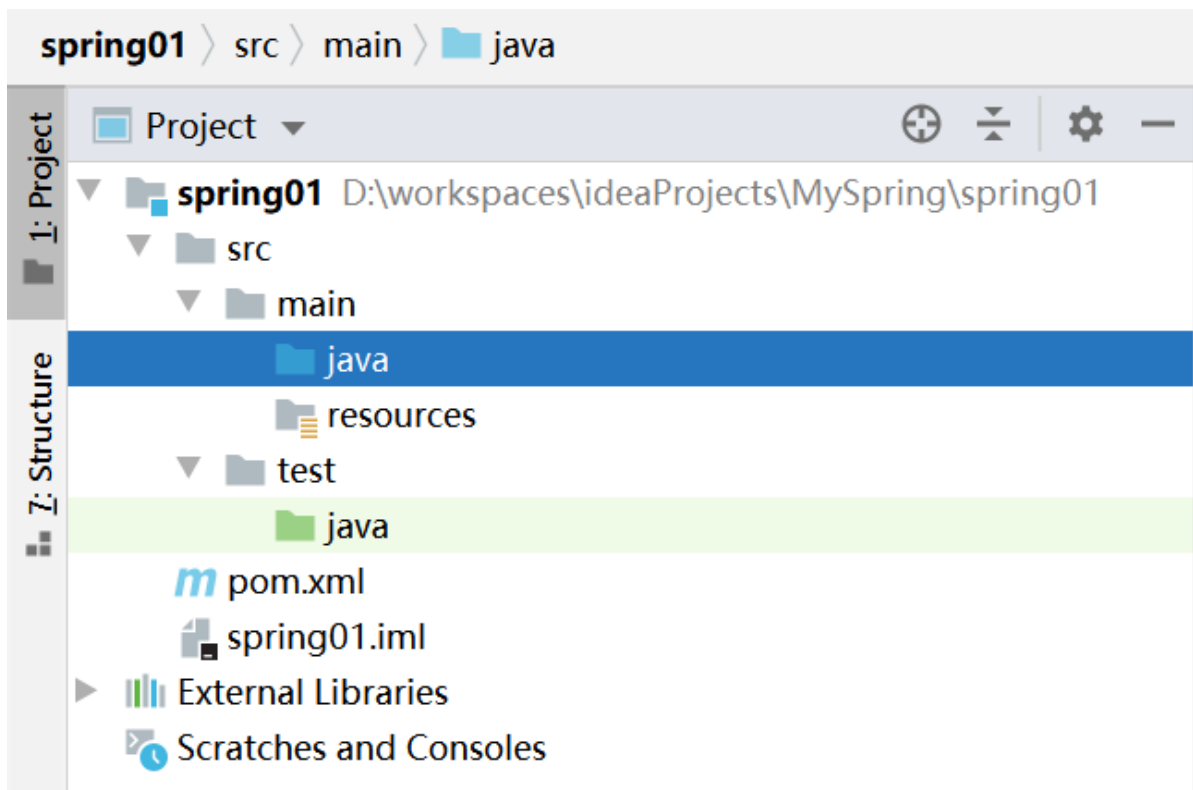
IoC 是指在程序开发中，实例的创建不再由调用者管理，而是由 Spring 容器创建。Spring 容器会负责控制程序之间的关系，而不是由程序代码直接控制，因此，控制权由程序代码转移到了 Spring 容器中，控制权发生了反转，这就是 Spring 的 IoC 思想。

创建实例：new 反射 克隆 序列化 动态代理

2.2 Spring入门案例

2.2.1 创建maven项目

创建完毕的目录结构



2.2.2 pom.xml文件添加依赖和插件

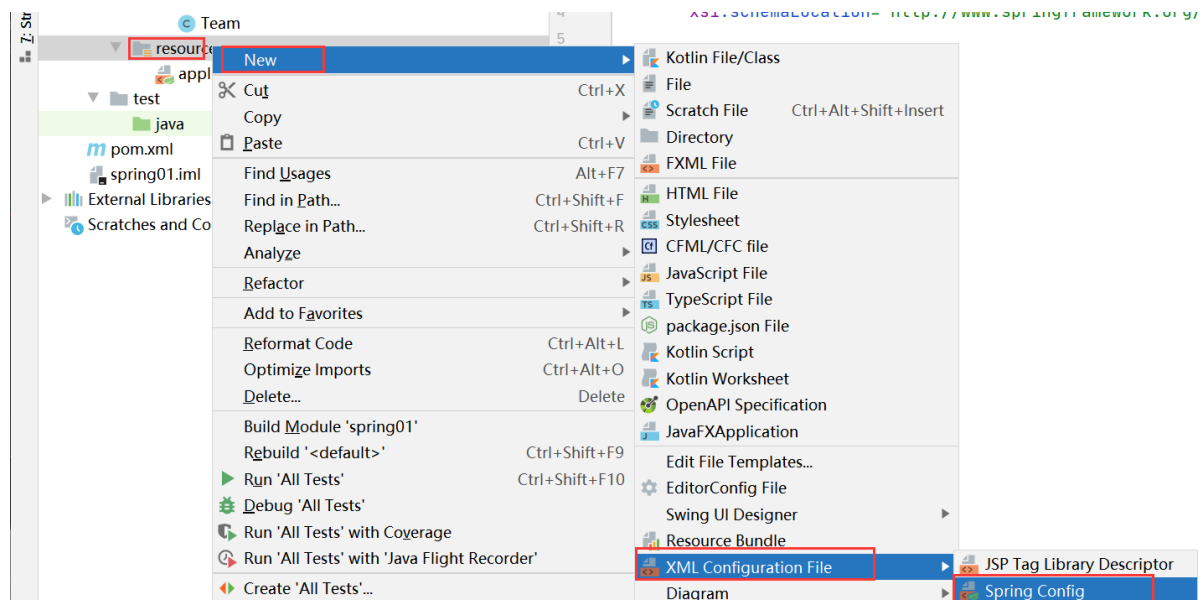
```
<dependencies>
    <!--单元测试-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <!--spring依赖-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.13.RELEASE</version>
    </dependency>
</dependencies>
<build>
    <plugins>
        <!--编译插件-->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.0</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

2.2.3 创建一个实体类

```
package com.lina.pojo;
/**
 * 实体类球队
 */
public class Team {
    private Integer id;
    private String name;
    private String location;

    public Team() {
        System.out.println("Team的默认构造方法被调用: id="+id+",name="+name+",
location="+location);
    }
}
```

2.3.4 创建Spring的配置文件application.xml



2.3.5 使用Spring容器创建对象

配置文件中创建对象

```
<?xml version="1.0" encoding="UTF-8"?>
<!--spring的配置文件
  1、beans: 根标签, spring中java的对象成为bean
  2、spring-beans.xsd 是约束文件(约束XML文件中能编写哪些标签)-->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

  <!--创建对象: 声明bean,通知spring要创建哪个类的对象
  一个bean标签声明一个对象:
    id="自定义的对象名称",要求唯一
    class="类的完全限定名" 包名+类名, spring底层是反射机制创建对象, 所以必须使用类
    名
    相当于 Team team1=new Team();创建好的对象放入一个集合Map中
    例如: springMap.put("team1",new Team());
  -->
  <bean id="team1" class="com.lina.pojo.Team"></bean>
</beans>
```

2.3.6 获取Spring容器

Spring 提供了两种 IoC 容器, 分别为 BeanFactory 和 ApplicationContext.

2.3.6.1 BeanFactory

BeanFactory 是基础类型的 IoC 容器,是一个管理 Bean 的工厂, 它主要负责初始化各种 Bean, 并调用它们的生命周期方法。

BeanFactory 接口有多个实现类, 最常见的是

org.springframework.beans.factory.xml.XmlBeanFactory, 它是根据 XML 配置文件中的定义装配 Bean 的。

```
BeanFactory beanFactory = new XmlBeanFactory(new FileSystemResource(Spring配置文件的名称));
```

2.3.6.2 ApplicationContext

ApplicationContext 是 BeanFactory 的子接口，也被称为**应用上下文**。它不仅提供了 BeanFactory 的所有功能，还添加了对 i18n（国际化）、资源访问、事件传播等方面的良好支持。

ApplicationContext 接口有两个常用的实现类：

2.6.2.2.1 ClassPathXmlApplicationContext——常用

该类从类路径 ClassPath 中寻找指定的 XML 配置文件，找到并装载完成 ApplicationContext 的实例化工作

```
ApplicationContext applicationContext=new ClassPathXmlApplicationContext(Spring配置文件的名称);
```

2.6.2.2.2 FileSystemXmlApplicationContext

```
ApplicationContext applicationContext = new  
FileSystemXmlApplicationContext(String configLocation);
```

它与 ClassPathXmlApplicationContext 的区别是：在读取 Spring 的配置文件时，FileSystemXmlApplicationContext 不再从类路径中读取配置文件，而是通过参数指定配置文件的位置，它可以获取类路径之外的资源，如“D:\application.xml”。

2.3.7 通过上下文对象获取容器中的对象

```
package com.lina.test;  
import com.lina.pojo.Team;  
import org.junit.Test;  
import org.springframework.beans.factory.BeanFactory;  
import org.springframework.beans.factory.xml.XmlBeanFactory;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
import org.springframework.context.support.FileSystemXmlApplicationContext;  
import org.springframework.core.io.FileSystemResource;  
import java.util.Date;  
  
public class Test1 {  
    @Test  
    public void test01(){  
        //使用spring容器创建对象  
        //1、指定spring配置文件的名称  
        String springConfig="application.xml";  
        //2、创建spring容器的对象：  
        //方式1:不推荐，了解  
        //BeanFactory beanFactory = new XmlBeanFactory(new  
        FileSystemResource("D:/workspaces/ideaProjects/MySpring/spring01/src/main/resources/application.xml"));  
        //beanFactory.getBean("team1");//根据ID从IOC容器获取对象  
        //方式2: applicationContext--常用  
        ApplicationContext applicationContext=new  
        ClassPathXmlApplicationContext(springConfig);//这里执行完毕容器中的对象都已经创建完成
```

```

//方式3: applicationContext--了解
//ApplicationContext applicationContext2 = new
FileSystemXmlApplicationContext("D:/workspaces/ideaProjects/MySpring/spring01/src/main/resources/application.xml");
//3、获取容器中的对象
Team team1= (Team) applicationContext.getBean("team1");
//4、容器其他api
int beanDefinitionCount = applicationContext.getBeanDefinitionCount();
System.out.println("spring容器中对象的个数: "+beanDefinitionCount);
String[] beanDefinitionNames =
applicationContext.getBeanDefinitionNames();
System.out.println("spring容器中所有对象的名称: ");
for (String name : beanDefinitionNames) {
    System.out.println(name);
}
}
}

```

2.3.8 创建非自定义对象

pox.xml文件中补充

```

<!--创建非自定义的对象-->
<bean id="date" class="java.util.Date"></bean>

```

上面的测试方法中添加如下内容:

```

//5、获取日期对象
Date date1= (Date) applicationContext.getBean("date1");
System.out.println("日期: "+date1);

```

2.3.9 bean标签的属性

属性	说明
class	指定bean对应类的全路径
name	name是bean对应对象的一个标识
scope	执行bean对象创建模式和生命周期,scope="singleton"和scope="prototype"
id	id是bean对象的唯一标识,不能添加特别字符
lazy-init	是否延时加载 默认值:false。true 延迟加载对象,当对象被调用的时候才会加载,测试的时候,通过getbean()方法获得对象。lazy-init="false" 默认值,不延迟,无论对象是否被使用,都会立即创建对象,测试时只需要加载配置文件即可。注意:测试的时候只留下id,class属性
init-method	只需要加载配置文件即可对象初始化方法
destroy-method	对象销毁方法

示例演示:

Team实体类补充如下方法:


```

public void init(){
    System.out.println("Team ---- init()");
}
public void destroy(){
    System.out.println("Team ---- destroy()");
}

```

application.xml配置文件添加如下内容：

```

<!--
    bean标签的属性：
    id="自定义的对象名称" ,要求唯一
    name="bean对于的一个标识"，一般使用id居多
    class="类的完全限定名"
    scope="singleton/prototype" 单例/多例
        singleton: 默认值，单例：在容器启动的时候就已经创建了对象，而且整个容器只有为一个
        的一个对象
        prototype: 多例，在使用对象的时候才创建对象，每次使用都创建新的对象
    lazy-init="true/false" 是否延迟创建对象，只针对单例有效
        false:默认值，不延迟创建对象，容器加载的时候立即创建
        true:延迟加载，使用对象的时候才去创建对象
    init-method="创建对象之后执行的初始化方法"
    destroy-method="对象销毁方法，调用容器destroy方法的时候执行"
-->
<bean id="team2" class="com.lina.pojo.Team" scope="singleton" lazy-
init="true" init-method="init" destroy-method="destroy"/>
<bean id="team3" class="com.lina.pojo.Team" scope="prototype" />

```

```

@Test
public void test02(){
    String springConfig="application.xml";
    ClassPathXmlApplicationContext applicationContext=new
ClassPathXmlApplicationContext(springConfig);
    Team team1 = (Team) applicationContext.getBean("team1");
    Team team11 = (Team) applicationContext.getBean("team1");
    System.out.println(team1);
    System.out.println(team11);
    Team team2 = (Team) applicationContext.getBean("team2");
    Team team22 = (Team) applicationContext.getBean("team2");
    System.out.println(team2);
    System.out.println(team22);
    applicationContext.close();//关闭容器
}

```

2.3 Spring容器创建对象的方式

2.3.1 使用默认的构造方法

2.3.2 使用带参数的构造方法

2.3.3 使用工厂类

三种方式的案例如下：

Team.java类中添加带参数构造方法：

```
public Team(Integer id, String name, String location) {
    this.id = id;
    this.name = name;
    this.location = location;
    System.out.println("Team - 带参数的构造方法
id="+id+",name="+name+",location="+location);
}
```

创建工厂类：

```
package com.kbb.pojo;
public class MyFactory {
    /**
     * 实例方法
     * @return
     */
    public Team instanceFun(){
        System.out.println("MyFactory-----instanceFun");
        return new Team(1003,"湖人","洛杉矶");
    }

    /**
     * 静态方法
     * @return
     */
    public static Team staticFun(){
        System.out.println("MyFactory-----staticFun");
        return new Team(1004,"小牛","达拉斯");
    }

    public static void main(String[] args) {
        Team team1 = MyFactory.staticFun();
        MyFactory factory=new MyFactory();
        Team team = factory.instanceFun();
    }
}
```

创建新的配置文件createType.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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--spring容器创建对象的方式：
    1、通过默认构造方法
```

```

2、通过带参数的构造方法
3、通过工厂方法：实例方法，静态方法-->
<!--1、通过默认构造方法-->
<bean id="team1" class="com.kbb.pojo.Team"></bean>
<!-- 2、通过带参数的构造方法-->
<bean id="team2" class="com.kbb.pojo.Team">
    <!--name:表示参数的名称-->
    <constructor-arg name="id" value="1001"/>
    <constructor-arg name="name" value="勇士"/>
    <constructor-arg name="location" value="金州"/>
</bean>
<bean id="team3" class="com.kbb.pojo.Team">
    <!--index:表示参数的下标索引-->
    <constructor-arg index="0" value="1002"/>
    <constructor-arg index="1" value="热火"/>
    <constructor-arg index="2" value="迈阿密"/>
</bean>
<!--3、通过工厂方法：
    3.1 静态方法
        Team team1 = MyFactory.staticFun();-->
    <bean id="staticTeam" class="com.kbb.pojo.MyFactory" factory-
method="staticFun"></bean>

    <!--3、通过工厂方法：
        3.2 实例方法
        MyFactory factory=new MyFactory();
        Team team = factory.instanceFun();-->
    <bean id="factory" class="com.kbb.pojo.MyFactory"></bean>
    <bean id="instanceTeam" factory-bean="factory" factory-method="instanceFun">
</bean>
</beans>

```

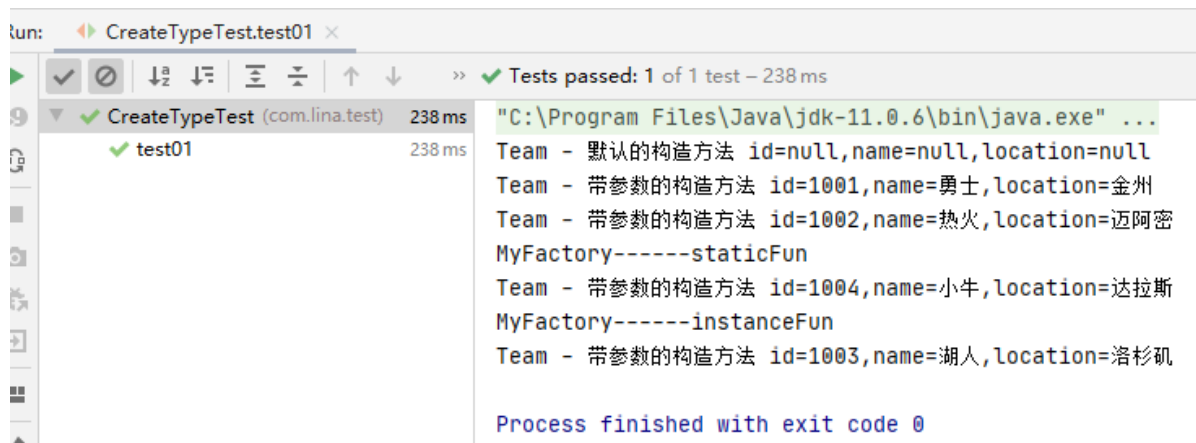
测试类：

```

package com.lina.test;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class CreateTypeTest {
    @Test
    public void test01(){
        ApplicationContext ac=new
        ClassPathXmlApplicationContext("createType.xml");
    }
}

```

测试结果：



```
Run: CreateTypeTest.test01 x
Tests passed: 1 of 1 test - 238 ms
CreateTypeTest (com.lina.test) 238 ms
test01 238 ms
"C:\Program Files\Java\jdk-11.0.6\bin\java.exe" ...
Team - 默认的构造方法 id=null,name=null,location=null
Team - 带参数的构造方法 id=1001,name=勇士,location=金州
Team - 带参数的构造方法 id=1002,name=热火,location=迈阿密
MyFactory-----staticFun
Team - 带参数的构造方法 id=1004,name=小牛,location=达拉斯
MyFactory-----instanceFun
Team - 带参数的构造方法 id=1003,name=湖人,location=洛杉矶
Process finished with exit code 0
```

2.4 基于XML的DI

DI—Dependency Injection，即“依赖注入”：是组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

IoC 是一个概念，是一种思想，其实现方式多种多样。依赖注入就是其中用的比较多的一种方式。

IoC和DI是同一个概念的不同角度描述。IoC是一种思想，概念，DI是实现它的手段。Spring框架使用依赖注入实现IoC。

Spring 容器是一个超级大工厂，负责创建、管理所有的 Java 对象，这些 Java 对象被称为 Bean。Spring 容器管理着容器中 Bean 之间的依赖关系，Spring 使用“依赖注入”的方式来管理 Bean 之间的依赖关系。使用 IoC 实现对象之间的解耦和。

2.4.1 注入分类

bean 实例在调用无参构造器创建对象后，就要对 bean 对象的属性进行初始化。初始化是由容器自动完成的，称为注入。

2.4.1.1 通过set方法

set 注入也叫设值注入是指，通过 setter 方法传入被调用者的实例。这种注入方式简单、直观，因而在 Spring 的依赖注入中大量使用。

2.4.1.2 通过构造方法

构造注入是指，在构造调用者实例的同时，完成被调用者的实例化。使用构造器设置依赖关系。

2.4.1.3. 自动注入

对于引用类型属性的注入，也可不在配置文件中显示的注入。可以通过为标签

设置 autowire 属性值，为引用类型属性进行隐式自动注入（默认是不自动注入引用类型属性）。根据自动注入判断标准的不同，可以分为两种：

byName：根据名称自动注入

byType：根据类型自动注入

1、byName

当配置文件中被调用者 bean 的 id 值与代码中调用者 bean 类的属性名相同时，可使用byName 方式，让容器自动将被调用者 bean 注入给调用者 bean。容器是通过调用者的 bean类的属性名与配置文件的被调用者 bean 的 id 进行比较而实现自动注入的。

2、byType

使用 byType 方式自动注入，要求：配置文件中被调用者 bean 的 class 属性指定的类，要与代码中调用者 bean 类的某引用类型属性类型同源。即要么相同，要么有 is-a 关系（子类，或是实现类）。但这样的同源的被调用 bean 只能有一个。多于一个，容器就不知该匹配哪一个了。

示例：

创建TeamDao.java

```
package com.lina.dao;

public class TeamDao {

    public void add(){
        System.out.println("TeamDao---- add----");
    }
}
```

创建TeamService.java

```
package com.lina.service;

import com.lina.dao.TeamDao;

public class TeamService {

    private TeamDao teamDao;//=new TeamDao();

    public void add(){
        teamDao.add();
        System.out.println("TeamService-----add----");
    }

    public TeamService() {
    }

    public TeamService(TeamDao teamDao) {
        this.teamDao = teamDao;
    }

    public TeamDao getTeamDao() {
        return teamDao;
    }

    public void setTeamDao(TeamDao teamDao) {
        this.teamDao = teamDao;
    }
}
```

创建配置文件DI.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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="teamDao" class="com.lina.dao.TeamDao"></bean>
    <!-- <bean id="teamDao1" class="com.lina.dao.TeamDao"></bean>-->
    <bean id="teamService" class="com.lina.service.TeamService">
        <!--使用set方法注入属性值-->
        <property name="teamDao" ref="teamDao"></property>
    </bean>

    <bean id="teamService2" class="com.lina.service.TeamService">
        <!--使用构造方法注入属性值-->
        <constructor-arg name="teamDao" ref="teamDao"></constructor-arg>
    </bean>
    <!--按名称自动注入：查找容器中id名与属性名一致的对象进行注入-->
    <bean id="teamService3" class="com.lina.service.TeamService"
autowire="byName">
    </bean>

    <!--按类型自动注入：查找容器中类型与属性类型相同或者符合is-a关系的对象进行注入，但是要求类型相同的对象唯一，否则抛出异常：不知道用哪一个匹配-->
    <bean id="teamService4" class="com.lina.service.TeamService"
autowire="byType">
    </bean>

</beans>
```

创建测试类：

```
package com.lina.test;

import com.lina.dao.TeamDao;
import com.lina.service.TeamService;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test02 {

    @Test
    public void test02(){
        ApplicationContext ac=new ClassPathXmlApplicationContext("DI.xml");
        TeamService teamService = (TeamService) ac.getBean("teamService");
        teamService.add();

        TeamService teamService2 = (TeamService) ac.getBean("teamService2");
        teamService2.add();

        TeamService teamService3 = (TeamService) ac.getBean("teamService3");
        teamService3.add();

        TeamService teamService4 = (TeamService) ac.getBean("teamService4");
```

```

        teamService4.add();
    }

    @Test
    public void test01(){
        TeamDao teamDao=new TeamDao();
        TeamService teamService=new TeamService();
        teamService.setTeamDao(teamDao);
        teamService.add();
    }
}

```

2.5、基于注解实现IoC--重要

对于 DI 使用注解，将不再需要在 Spring 配置文件中声明 bean 实例。

2.5.1 声明Bean的注解 @Component

在类上添加注解@Component表示该类创建对象的权限交给Spring容器。注解的value属性用于指定bean的id值，value可以省略。

@Component 不指定 value 属性，bean 的 id 是类名的首字母小写。

```

//@Component 注解标识在类上，表示对象由Spring容器创建 value属性表示创建的id值，value可以省略，值也可省略，默认就是类名的首字母小写
@Component(value="teamDao")
// 相当于xml文件中的<bean id="teamDao" class="com.kkb.dao.TeamDao"></bean>
public class TeamDao {

    public void add(){
        System.out.println("TeamDao---- add----");
    }

    public TeamDao() {
        System.out.println("TeamDao ---- 默认的构造方法");
    }
}

```

除此之外，Spring中还提供了其他3个用于创建对象的注解：

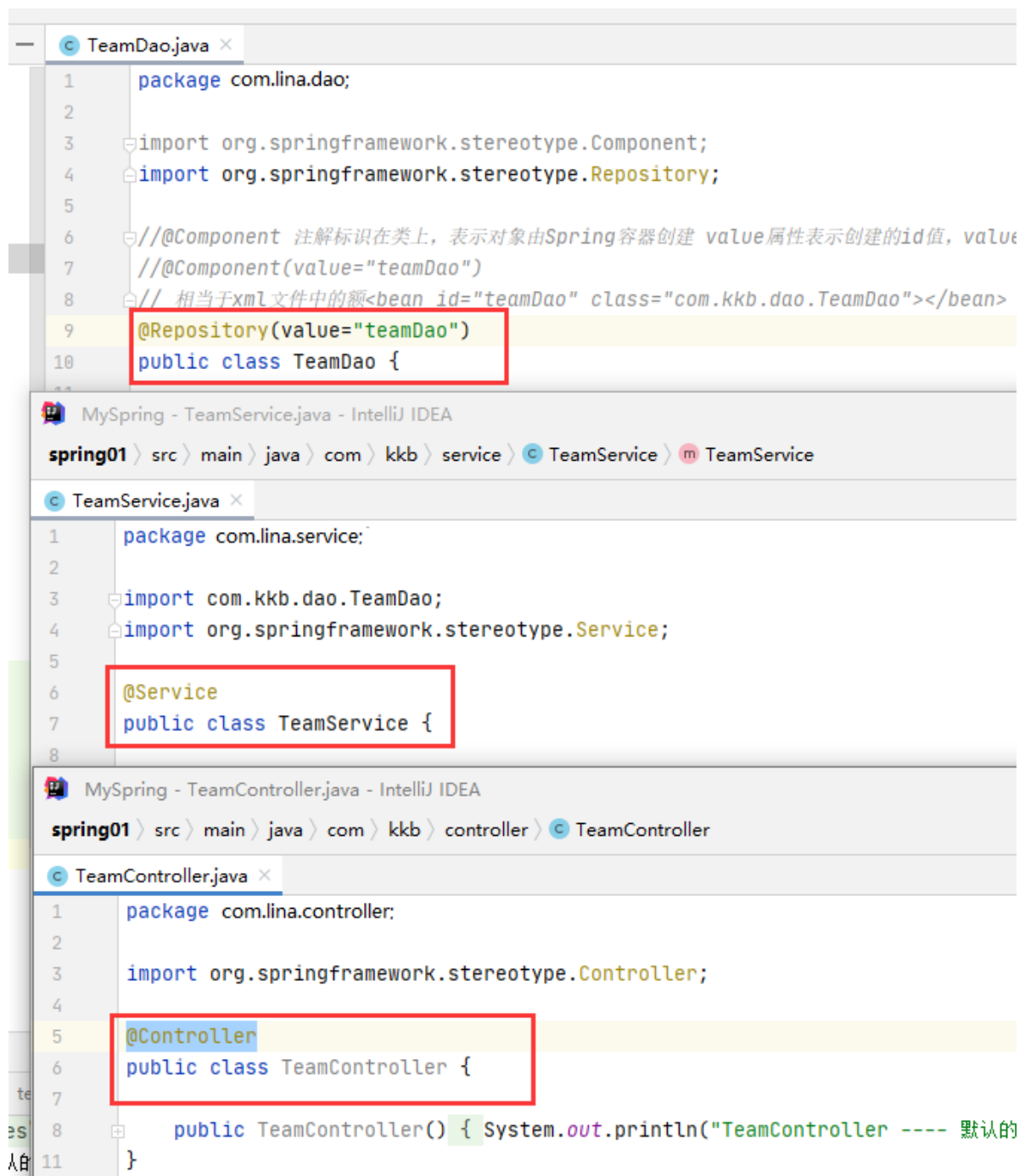
@Repository : 用于dao实现类的的注解

@Service: 用户service实现类的注解

@Controller: 用于controller实现类的注解

这三个注解与@Component 都可以创建对象，但这三个注解还有其他的含义，@Service创建业务层对象，业务层对象可以加入事务功能，@Controller 注解创建的对象可以作为处理器接收用户的请求。

@Repository, @Service, @Controller 是对@Component 注解的细化，标注不同层的对象。即持久层对象，业务层对象，控制层对象。



2.5.2 包扫描

需要在 Spring 配置文件中配置组件扫描器，用于在指定的基本包中扫描注解。**如果没有包扫描，添加的创建对象的注解不生效。**

如果要扫描的包有多个，可以有以下方式扫描：

- 1、使用多个context:component-scan指定不同的包路径


```
annotation.xml x
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="http://www.springframework.org/schema/beans
6                           http://www.springframework.org/schema/beans/spring-beans.xsd
7                           http://www.springframework.org/schema/context
8                           http://www.springframework.org/schema/context/spring-context.xsd">
9
10    <!-- context:component-scan 表示告知spring 要扫描的包
11         这些包以及子包当中的类上如果添加了@Component注解，这些添加了注解的类就交给了spring 容器创建对象
12         注意：在beans标签中添加 xmlns:context="http://www.springframework.org/schema/context"
13         http://www.springframework.org/schema/context
14         http://www.springframework.org/schema/context/spring-context.xsd
15    -->
16
17    <!-- 多个包的扫描：方式1 使用多个<context:component-scan 表明-->
18    <context:component-scan base-package="com.lina.dao" ></context:component-scan>
19    <context:component-scan base-package="com.lina.service" ></context:component-scan>
20    <context:component-scan base-package="com.lina.controller" ></context:component-scan>
```

2、指定 base-package 的值使用分隔符

分隔符可以使用逗号（，）分号（；）还可以使用空格，不建议使用空格。

<!-- 多个包的扫描：方式2： base-package 中直接声明要扫描的多个包，多个值用逗号、分号或者空格分割，但是空格不推荐-->

```
<context:component-scan base-
package="com.lina.dao,com.lina.service,com.lina.controller"></context:component-
scan>
```

3、base-package 是指定到父包名

base-package 的值是基本包，容器启动会扫描包及其子包中的注解，当然也会扫描到子包下级的子包。所以 base-package 可以指定一个父包就可以。

但不建议使用顶级的父包，扫描的路径比较多，导致容器启动时间变慢。指定到目标包和合适的。也就是注解所在包全路径。

<!-- 多个包的扫描：方式3： base-package 中直接声明要扫描的多个包的父包-->

```
<context:component-scan base-package="com.lina"></context:component-scan>
```

2.5.3 属性注入@Value

需要在属性上使用注解@Value，该注解的 value 属性用于指定要注入的值。使用该注解完成属性注入时，类中无需 setter。当然，若属性有 setter，则也可将其加到 setter 上。

```
@Component
public class Team {

    private Integer id;

    private String name;

    @Value("洛杉矶")
    private String location;

    @Value("1001")
    public void setId(Integer id) {
        this.id = id;
    }

    @Value("湖人队")
    public void setName(String name) {
        this.name = name;
    }

    public void setLocation(String location) {
        this.location = location;
    }
}
```

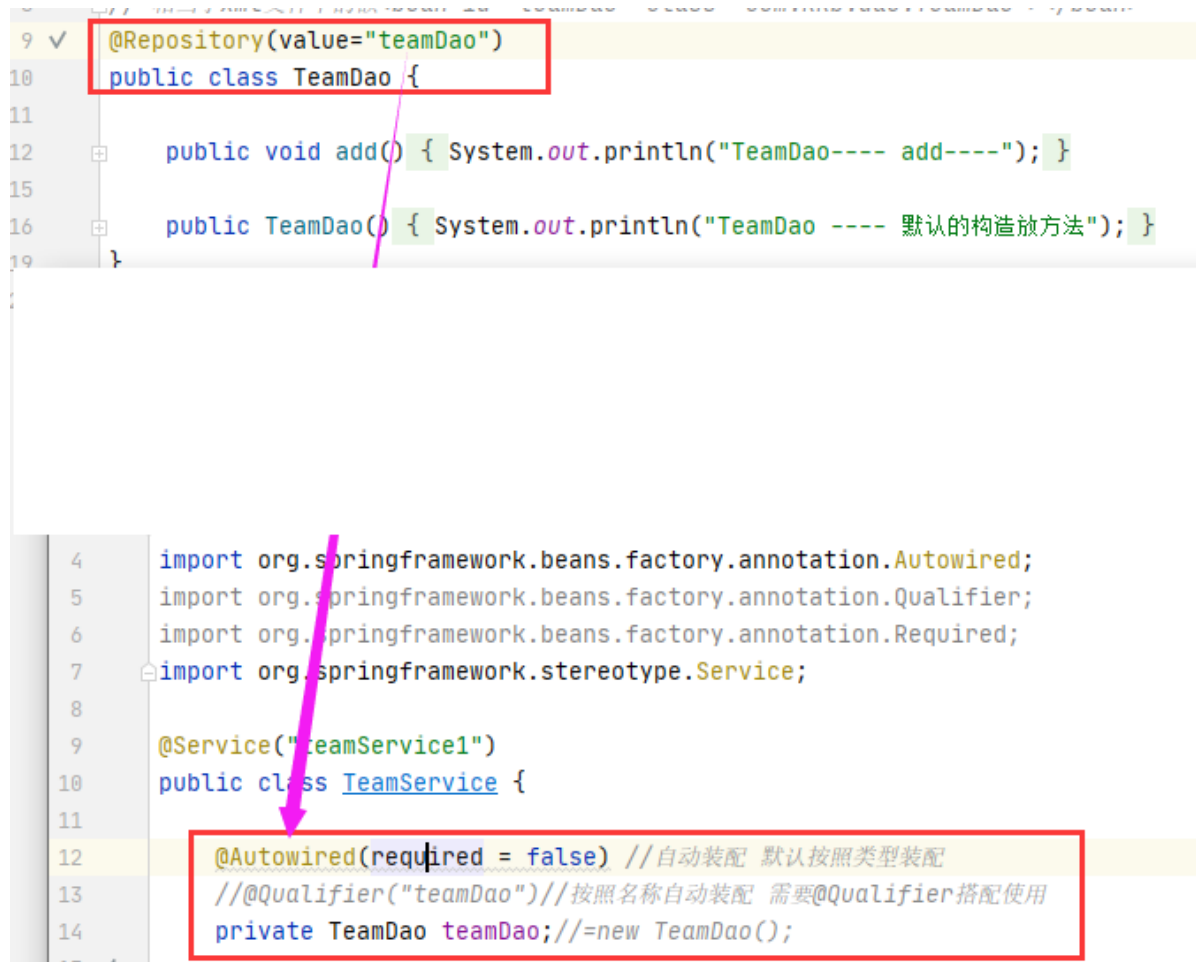
2.5.4 byType自动注入@Autowired

需要在引用属性上使用注解@Autowired，该注解默认使用按类型自动装配 Bean 的方式。使用该注解完成属性注入时，类中无需 setter。当然，若属性有 setter，则也可将其加到 setter 上。

2.5.5 byName自动注入@Autowired和@Qualifier

需要在引用属性上联合使用注解@Autowired 与@Qualifier。@Qualifier 的 value 属性用于指定要匹配的 Bean 的 id 值。类中无需 set 方法，也可加到 set 方法上。

@Autowired 还有一个属性 required，默认值为 true，表示当匹配失败后，会终止程序运行。若将其值设置为 false，则匹配失败，将被忽略，未匹配的属性值为 null。



2.5.6 自动注入@Resource

Spring提供了对jdk中@Resource注解的支持。@Resource 注解既可以按名称匹配Bean，也可以按类型匹配 Bean。默认是按名称注入。使用该注解，要求JDK 必须是 6 及以上版本。@Resource 可在属性上，也可在 set 方法上。

1、byType注入引用类型属性

@Resource 注解若不带任何参数，采用默认按名称的方式注入，按名称不能注入 bean，则会按照类型进行 Bean 的匹配注入。

2、byName注入引用类型属性

@Resource 注解指定其 name 属性，则 name 的值即为按照名称进行匹配的 Bean 的 id。

```

9  @Service("teamService1")
10 public class TeamService {
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
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
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

3、Spring核心之AOP

3.1 什么是AOP

AOP为Aspect Oriented Programming的缩写，意思为**面向切面编程**，是通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。

AOP的作用：不修改源码的情况下，程序运行期间对方法进行功能增强

好处：1、减少代码的重复，提高开发效率，便于维护。

2、专注核心业务的开发。

核心业务和服务性代码混合在一起

开发中：各自做自己擅长的事情，运行的时候将服务性代码织入到核心业务中。

通过spring工厂自动实现将服务性代码以切面的方式加入到核心业务代码中。

3.2 AOP的实现机制-动态代理

3.2.1 什么是代理模式

代理：自己不做，找人帮你做。

代理模式：在一个原有功能的基础上添加新的功能。

分类：静态代理和动态代理。

3.3 静态代理

3.3.1 原有方式：核心业务和服务方法都编写在一起

```

package com.lina.service;
public class TeamService {
    public void add(){
        try {
            System.out.println("开始事务");
            System.out.println("TeamService----- add-----");// 核心业务
            System.out.println("提交事务");
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("回滚事务");
        }
    }
}

```

3.3.2 基于类的静态代理

将服务性代码分离出来，核心业务--保存业务中只有保存功能

```

package com.lina.service;
public class TeamService {
    public void add(){
        System.out.println("TeamService----- add-----");// 核心业务
    }
}

```

```

package com.lina.staticproxy;

import com.lina.service.TeamService;

/**
 * 基于类的静态代理：
 * 要求继承被代理的类
 * 弊端：每次只能代理一个类
 */
public class ProxyTeamService extends TeamService {

    public void add(){
        try {
            System.out.println("开始事务");
            super.add();//核心业务就是由被代理对象完成；其他服务功能由代理类完成
            System.out.println("提交事务");
        } catch (Exception e) {
            System.out.println("回滚事务");
        }
    }
}

```

```

public static void main(String[] args) {
    TeamService ser=new ProxyTeamService();
    ser.add();
}

```

弊端：代理类只能代理一个类

3.3.3 基于接口的静态代理

为核心业务（保存add）创建一个接口,通过接口暴露被代理的方法

要求：代理类和被代理类都实现了同一个接口

```
package com.lina.service;

/**
 * 接口定义核心方法
 */
public interface IService {
    void add();
}
```

```
package com.lina.service;
public class TeamService implements IService{
    @Override
    public void add(){
        System.out.println("TeamService---- add----");// 核心业务
    }
}

package com.lina.service;
public class UserService implements IService{
    @Override
    public void add() {
        System.out.println("UserService---- add-----");
    }
}
```

```
package com.lina.staticproxy;
import com.lina.service.IService;
/**
 * 基于接口的静态代理：
 * 代理类和被代理类实现同一个接口
 */
public class ProxyTranService implements IService {
    private IService service;//被代理的对象
    public ProxyTranService(IService service) {
        this.service = service;
    }
    @Override
    public void add() {
        try {
            System.out.println("开始事务");
            service.add();//核心业务就是由被代理对象完成；其他服务功能由代理类完成
            System.out.println("提交事务");
        }catch (Exception e){
            System.out.println("回滚事务");
        }
    }
}
```

```
package com.lina.staticproxy;
import com.lina.service.IService;
```

```

public class ProxyLogService implements IService {
    private IService service;//被代理对象
    public ProxyLogService(IService service) {
        this.service = service;
    }
    @Override
    public void add() {
        try {
            System.out.println("开始日志");
            service.add();//核心业务就是由被代理对象完成；其他服务功能由代理类完成
            System.out.println("结束日志");
        } catch (Exception e) {
            System.out.println("异常日志");
        }
    }
}

```

测试类:

```

public static void main(String[] args) {
    TeamService teamService=new TeamService();//被代理对象
    UserService userService=new UserService();//被代理对象

    ProxyTranService tranService=new ProxyTranService(userService);//事务代理
    对象--一级代理
    //tranService.add();//代理对象干活

    ProxyLogService logService=new ProxyLogService(tranService);//日志的代理对
    象--二级代理
    logService.add();
}

```

3.3.4 提取出切面代码，作为AOP接口

共有4个位置可以将切面代码编织进入核心业务代码中。

```

package com.lina.aop;

/**
 * 切面：服务代码，切入到核心代码中,切入到哪里，给了四个位置
 */
public interface AOP {
    void before();
    void after();
    void exception();
    void myFinally();
}

```

```

package com.lina.aop;
public class TranAOP implements AOP {
    @Override
    public void before() {
        System.out.println("事务----before");
    }
}

```

```

@Override
public void after() {
    System.out.println("事务----after");
}
@Override
public void exception() {
    System.out.println("事务----exception");
}
@Override
public void myFinally() {
    System.out.println("事务----myFinally");
}
}

```

```

package com.lina.aop;
public class LogAop implements AOP{
    @Override
    public void before() {
        System.out.println("日志----before");
    }
    @Override
    public void after() {
        System.out.println("日志----after");
    }
    @Override
    public void exception() {
        System.out.println("日志----exception");
    }
    @Override
    public void myFinally() {
        System.out.println("日志----myFinally");
    }
}

```

```

package com.lina.staticproxy;

import com.lina.aop.AOP;
import com.lina.service.IService;

public class ProxyAOPService implements IService {
    private IService service;//被代理对象
    private AOP aop;//要加入切面
    public ProxyAOPService(IService service, AOP aop) {
        this.service = service;
        this.aop = aop;
    }
    @Override
    public void add() {
        try {
            aop.before();
            service.add();//被代理对象干活
            aop.after();
        }catch (Exception e){
            aop.exception();
        }finally {
            aop.myFinally();
        }
    }
}

```



```

    }
}
}

```

```

@Test
public void test02(){
    IService teamService=new TeamService();//被代理对象--核心内容
    AOP logAop=new LogAop();//切面-服务内容
    AOP tranAop=new TranAOP();
    IService service=new ProxyAOPService(teamService,logAop); //代理对象--一级代理
    IService service2=new ProxyAOPService(service,tranAop);//代理对象--二级代理
    service2.add();
}

```

总结静态代理：

- 1) 可以做到在不修改目标对象的功能前提下，对目标对象功能扩展。
- 2) 缺点：
 - 因为代理对象，需要与目标对象实现一样的接口。所以会有很多代理类，类太多。
 - 一旦接口增加方法，目标对象与代理对象都要维护。

3.4 动态代理

静态代理：要求代理类一定存在，

动态代理：程序运行的时候，根据要被代理的对象动态生成代理类。

类型：1、基于JDK的动态代理

2、基于CGLIB的动态代理

3.4.1 基于JDK的动态代理

```

static Object newProxyInstance(ClassLoader loader, 类<?>[] interfaces, InvocationHandler h)
    返回指定接口的代理类的实例，该接口将方法调用分派给指定的调用处理程序。

```

3.4.1.1 直接编写测试类

```

/*newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)

```

ClassLoader：类加载器，因为动态代理类，借助别人的类加载器。一般使用被代理对象的类加载器。

Class<?>[] interfaces:接口类对象的集合，针对接口的代理，针对哪个接口做代理，一般使用的就是被代理对象的接口。

InvocationHandler：句柄，回调函数，编写代理的规则代码

```

public Object invoke(Object arg0, Method arg1, Object[] arg2)
Object arg0: 代理对象
Method arg1: 被代理的方法
Object[] arg2: 被代理方法被执行的时候的参数的数组
*/

```

```

package com.lina.dynamicproxy;

```

```

import com.lina.service.IService;
import com.lina.service.TeamService;

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

public class MyJDKProxy {
    public static void main(String[] args) {
        //目标对象--被代理对象
        TeamService teamService=new TeamService();
        //返回代理对象 调用JDK中Proxy类中的静态方法newProxyInstance获取动态代理类的实例
        IService proxyService= (IService) Proxy.newProxyInstance(
            teamService.getClass().getClassLoader(),
            teamService.getClass().getInterfaces(),
            new InvocationHandler() { //回调函数 编写代理规则
                @Override
                public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                    try {
                        System.out.println("开始事务");
                        Object invoke = method.invoke(teamService, args); //核
心业务

                        System.out.println("提交事务");
                        return invoke;
                    } catch (Exception e){
                        System.out.println("回滚事务");
                        e.printStackTrace();
                        throw e;
                    } finally {
                        System.out.println("finally-----");
                    }
                }
            }
        );
        //代理对象干活
        proxyService.add();
        System.out.println(teamService.getClass());
        System.out.println(proxyService.getClass()+"-----");
    }
}

```

3.4.1.2 结构化设计

方式1:

```

package com.lina.dynamicproxy;

import com.lina.aop.AOP;
import com.lina.service.IService;

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

public class ProxyHandler implements InvocationHandler {

```

```

private IService service;//目标对象
private AOP aop;//切面

public ProxyHandler(IService service, AOP aop) {
    this.service = service;
    this.aop = aop;
}

@Override
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    try {
        aop.before();
        Object invoke = method.invoke(service, args);//核心业务
        aop.after();
        return invoke;
    }catch (Exception e){
        aop.exception();
        e.printStackTrace();
        throw e;
    }finally {
        aop.myFinally();
    }
}
}

```

```

public static void main2(String[] args) {
    //目标对象--被代理对象
    TeamService teamService=new TeamService();
    //切面
    AOP tranAop=new TranAOP();
    //返回代理对象 基于JDK的动态代理
    IService proxyService= (IService) Proxy.newProxyInstance(
        teamService.getClass().getClassLoader(),
        teamService.getClass().getInterfaces(),
        new ProxyHandler(teamService,tranAop)
    );
    //代理对象干活
    proxyService.add();
    System.out.println(teamService.getClass());
    System.out.println(proxyService.getClass()+"-----");
}

```

方式2:

```

package com.lina.dynamicproxy;

import com.lina.aop.AOP;
import com.lina.service.IService;

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

public class ProxyFactory {

```

```

private IService service;//目标对象
private AOP aop;//切面

public ProxyFactory(IService service, AOP aop) {
    this.service = service;
    this.aop = aop;
}

/**
 * 获取动态代理的示例
 * @return
 */
public Object getProxyInstance() {
    return Proxy.newProxyInstance(
        service.getClass().getClassLoader(),
        service.getClass().getInterfaces(),
        new InvocationHandler() { //回调函数 编写代理规则
            @Override
            public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
                try {
                    aop.before();
                    Object invoke = method.invoke(service, args);//核心业
务

                    aop.after();
                    return invoke;
                } catch (Exception e) {
                    aop.exception();
                    e.printStackTrace();
                    throw e;
                } finally {
                    aop.myFinally();
                }
            }
        }
    );
}
}

```

```

public static void main(String[] args) {
    //目标对象--被代理对象
    TeamService teamService=new TeamService();
    //切面
    AOP tranAop=new TranAOP();
    AOP logAop=new LogAop();
    //获取代理对象
    IService service= (IService) new
ProxyFactory(teamService,tranAop).getProxyInstance();
    IService service1= (IService) new
ProxyFactory(service,logAop).getProxyInstance();
    service1.add();//核心业务+服务代码混合在一起的完整的业务方法
}

```

代理对象不需要实现接口，但是目标对象一定要实现接口；否则不能用JDK动态代理 如果想要功能扩展，但目标对象没有实现接口，怎样功能扩展？

子类的方式实现代理CGLIB。

3.4.2 基于CGLIB的动态代理

Cglib代理，也叫做子类代理。在内存中构建一个子类对象从而实现对目标对象功能的扩展。

- JDK的动态代理有一个限制，就是使用动态代理的对象必须实现一个或多个接口。如果想代理没有实现接口的类，就可以使用CGLIB实现。
- CGLIB是一个强大的高性能的代码生成包，它可以在运行期扩展Java类与实现Java接口。它广泛的被许多AOP的框架使用，例如Spring AOP和dynaop，为他们提供方法的interception。
- CGLIB包的底层是通过使用一个小而快的字节码处理框架ASM，来转换字节码并生成新的类。不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。

3.4.2.1 直接编写测试类

```
package com.lina.cglibproxy;
public class NBAService {

    public int add(String name,int id){
        System.out.println("NBAService----- add-----");
        return id;
    }
}
```

```
public static void main(String[] args) {
    //目标对象:没有接口
    NBAService nbaService=new NBAService();
    //创建代理对象: 选择cglib动态代理
    NBAService proxyService= (NBAService)
    Enhancer.create(nbaService.getClass(),
        new MethodInterceptor() { //回调函数编写代理规则
            @Override
            public Object intercept(Object o, Method method, Object[] objects,
                MethodProxy methodProxy) throws Throwable {
                try {
                    System.out.println("开始事务");
                    Object invoke = methodProxy.invokeSuper(o, objects); //核心
                    System.out.println("提交事务");
                    return invoke;
                } catch (Exception e) {
                    System.out.println("事务回滚");
                    throw e;
                } finally {
                    System.out.println("finally-----");
                }
            }
        });
    //代理对象干活
    int res=proxyService.add("huren",1001);
    System.out.println(res);
}
```

3.4.2.2 结构化设计方式

```
package com.lina.cglibproxy;

import com.lina.aop.AOP;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class CglibProxyFactory {
    //目标对象
    private NBAService nbaService;//没有实现接口的
    //切面
    private AOP aop;//切面

    /**
     * 创建代理对象
     * @param nbaService
     * @param aop
     * @return
     */
    public Object getProxyInstance(NBAService nbaService, AOP aop) {
        return Enhancer.create(nbaService.getClass(),
            new MethodInterceptor() {
                @Override
                public Object intercept(Object o, Method method, Object[]
objects, MethodProxy methodProxy) throws Throwable {
                    try {
                        aop.before();
                        Object o1 = methodProxy.invokeSuper(o, objects);
                        aop.after();
                        return o1;
                    } catch (Exception e) {
                        aop.exception();
                        throw e;
                    } finally {
                        system.out.println("finally-----");
                    }
                }
            }
        );
    }
}
```

```
public static void main(String[] args) {
    //目标对象:没有接口
    NBAService nbaService=new NBAService();
    //创建切面
    AOP tranAop=new TranAOP();
    //创建代理对象: 选择cglib动态代理
    NBAService proxyInstance = (NBAService) new
CglibProxyFactory().getProxyInstance(nbaService, tranAop);
    int res=proxyInstance.add("huren",1001);
    System.out.println(res);
}
```

3.5 SpringAOP

3.5.1 Spring 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。

我们先来介绍AOP的相关术语：

- **Target(目标对象)**

要被增强的对象，一般是业务逻辑类的对象。--TeamService

- **Proxy (代理)**

一个类被 AOP 织入增强后，就产生一个结果代理类。--ProxyTeamService--动态生成

- **Aspect(切面)**

表示增强的功能，就是一些代码完成的某个功能，非业务功能。是切入点和通知的结合。--服务性代码：日志 权限 事务

- **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（一般是类中的业务方法）,因为Spring只支持方法类型的连接点。--add()

- **Pointcut(切入点)**

切入点指声明的一个或多个连接点的集合。通过切入点指定一组方法。

被标记为 final 的方法是不能作为连接点与切入点的。因为最终的是不能被修改的，不能被增强的。

- **Advice(通知/增强)**

所谓通知是指拦截到 Joinpoint 之后所要做的事情就是通知。通知定义了增强代码切入到目标代码的时间点，是目标方法执行之前执行，还是之后执行等。通知类型不同，切入时间不同。通知的类型：前置通知,后置通知,异常通知,最终通知,环绕通知。

切入点定义切入的位置，通知定义切入的时间。

- **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。

切面的三个关键因素：

- 1、切面的功能--切面能干啥
- 2、切面的执行位置--使用Pointcut表示切面执行的位置
- 3、切面的执行时间--使用Advice表示时间，在目标方法之前还是之后执行。

3.5.2 AspectJ 对 AOP 的实现

对于 AOP 这种编程思想，很多框架都进行了实现。Spring 就是其中之一，可以完成面向切面编程。AspectJ 也实现了 AOP 的功能，且其实现方式更为简捷而且还支持注解式开发。所以，Spring 又将 AspectJ 的对于 AOP 的实现也引入到了自己的框架中。

在 Spring 中使用 AOP 开发时，一般使用 AspectJ 的实现方式

AspectJ 是一个优秀面向切面的框架，它扩展了 Java 语言，提供了强大的切面实现。

3.5.2.1 AspectJ的通知类型

AspectJ 中常用的通知有5种类型：

1. 前置通知
2. 后置通知
3. 环绕通知
4. 异常通知
5. 最终通知

3.5.2.2 AspectJ的切入点表达式

AspectJ 定义了专门的表达式用于指定切入点。

表达式的原型如下：

```
execution(modifiers-pattern? ret-type-pattern  
declaring-type-pattern?name-pattern(param-pattern)  
throws-pattern?)
```

说明：

modifiers-pattern 访问权限类型

ret-type-pattern 返回值类型

declaring-type-pattern 包名类名

name-pattern(param-pattern) 方法名(参数类型和参数个数)

throws-pattern 抛出异常类型

? 表示可选的部分

以上表达式共 4 个部分。execution(访问权限 方法返回值 方法声明(参数) 异常类型)

切入点表达式要匹配的对象就是目标方法的方法名。所以，execution 表达式中就是方法的签名。

PS:表达式中黑色文字表示可省略部分，各部分间用空格分开。在其中可以使用以下符号：

符号	意义
*	0-多个任意字符
..	用在方法参数中，表示任意个参数；用在包名后，表示当前及其子包路径
+	用在类名后，表示当前及其子类；用在接口后，表示当前接口及其实现类

示例：

```
execution(* com.lina.service.*.*(..))
```

指定切入点为：定义在 service 包里的任意类的任意方法。

```
execution(* com.lina.service...*.*(..))
```

指定切入点为：定义在 service 包或者子包里的任意类的任意方法。“..”出现在类名中时，后面必须跟“*”，表示包、子包下的所有类。

```
execution(* com.lina.service.IUserService+.*(..))
```

指定切入点为：IUserService 若为接口，则为接口中的任意方法及其所有实现类中的任意方法；若为类，则为该类及其子类中的任意方法。

3.5.3 注解方式实现AOP

开发阶段：关注核心业务和AOP代码

运行阶段：Spring框架会在运行的时候将核心业务和AOP代码通过动态代理的方式编织在一起

代理方式的选择：是否实现了接口：有接口就选择JDK动态代理；没有就选择CGLIB动态代理。

1、创建项目引入依赖

```
<dependencies>
    <!--spring 核心依赖-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.13.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>5.2.13.RELEASE</version>
    </dependency>
    <!--测试依赖-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
<build>
    <plugins>
        <!--编译插件-->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.0</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

2、创建spring配置文件引入约束

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd">
    <!--在beans标签中 引入AOP和context约束-->
</beans>
```

3、创建核心业务类

```
public interface IService {
    void add(int id,String name);
    boolean update(int num);
}
```

```
package com.lina.service;
import org.springframework.stereotype.Service;
/**
 * 核心业务类
 */
@Service
public class TeamService implements IService{
    @Override
    public void add(int id, String name) {
        System.out.println("TeamService---- add----");
    }

    @Override
    public boolean update(int num) {
        System.out.println("TeamService---- update----");
        //int res=10/0;
        if(num>666){
            return true;
        }
        return false;
    }
}
```

```
package com.lina.service;
import org.springframework.stereotype.Service;
/**
 * 核心业务类
 */
@Service("nbaService")
public class NBAService implements IService{
    @Override
    public void add(int id, String name) {
        System.out.println("NBAService---- add----");
    }
}
```

```

    }

    @Override
    public boolean update(int num) {
        System.out.println("NBAService---- update----");
        //int res=10/0;
        if(num>666){
            return true;
        }
        return false;
    }
}

```

4、定义切面类

```

package com.lina.aop;

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

/**
 * 切面类
 */
@Component //切面对象的创建权限依然交给spring容器
@Aspect //aspectj 框架的注解 标识当前类是一个切面类
public class MyAspect{
    /**
     * 当较多的通知增强方法使用相同的 execution 切入点表达式时，编写、维护均较为麻烦。
     * AspectJ 提供了@Pointcut 注解，用于定义 execution 切入点表达式。
     * 其用法是，将@Pointcut 注解在一个方法之上，以后所有的 execution 的 value 属性值均
     * 可使用该方法名作为切入点。代表的就是@Pointcut 定义的切入点。
     * 这个使用@Pointcut 注解方法一般使用 private 的标识方法，即没有实际作用的方法。
     */
    @Pointcut("execution(* com.lina.service..*.*(..))")
    private void pointCut(){

    }

    @Pointcut("execution(* com.lina.service..*.add*(..))")
    private void pointCut2(){

    }

    /**
     * 声明前置通知
     * @param jp
     */
    @Before("pointCut()")
    public void before(JoinPoint jp){
        System.out.println("前置通知：在目标方法执行之前被调用的通知");
        String name = jp.getSignature().getName();
        System.out.println("拦截的方法名称: "+name);
        Object[] args = jp.getArgs();
        System.out.println("方法的参数格式: "+args.length);
        System.out.println("方法参数列表: ");
    }
}

```

```

        for (Object arg : args) {
            System.out.println("\t"+arg);
        }
    }

    /**
     * AfterReturning 注解声明后置通知
     * value: 表示切入点表达式
     * returning 属性表示 返回的结果, 如果需要的话可以在后置通知的方法中修改结果
     */
    @AfterReturning(value = "pointCut2()", returning = "result")
    public Object afterReturn(Object result){
        if(result!=null){
            boolean res=(boolean)result;
            if(res){
                result=false;
            }
        }
        System.out.println("后置通知: 在目标方法执行之后被调用的通知,result="+result);
        return result;
    }

    /**
     * Around 注解声明环绕通知
     * ProceedingJoinPoint 中的proceed方法表示目标方法被执行
     */
    @Around(value = "pointCut()")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("环绕方法---目标方法的执行之前");
        Object proceed = pjp.proceed();
        System.out.println("环绕方法---目标方法的执行之后");
        return proceed;
    }

    /**
     * AfterThrowing 注解声明异常通知方法
     * value: 表示切入点表达式
     * returning 属性表示 返回的结果, 如果需要的话可以在后置通知的方法中修改结果
     */
    @AfterThrowing(value = "pointCut()", throwing = "ex")
    public void exception(JoinPoint jp, Throwable ex){
        //一般会把异常发生的时间、位置、原有都记录下来
        System.out.println("异常通知: 在目标方法执行出现异常的时候才会别调用的通知, 否则不
        执行");
        System.out.println(jp.getSignature()+"方法出现异常, 异常信息
        是: "+ex.getMessage());
    }

    /**
     * After 注解声明为最终通知
     */
    @After( "pointCut()")
    public void myFinally(){
        System.out.println("最终通知: 无论是否出现异常都是最后被调用的通知");
    }
}

```

5、业务类和切面类添加注解

```
/**
 * 核心业务类
 */
@Service
public class TeamService implements IService{

}

/**
 * 核心业务类
 */
@Service("nbaService")
public class NBAService implements IService{

}

/**
 * 切面类
 */
@Component //对象的创建也要交给spring容器
@Aspect //指定当前类是切面类
public class MyAspect {

}
```

在定义好切面 Aspect 后，需要通知 Spring 容器，让容器生成“目标类+ 切面”的代理对象。这个代理是由容器自动生成的。只需要在 Spring 配置文件中注册一个基于 aspectj 的自动代理生成器，其就会自动扫描到@Aspect 注解，并按通知类型与切入点，将其织入，并生成代理。

5、spring.xml配置文件中开启包扫描和 注册aspectj的自动代理

```
<!--包扫描-->
<context:component-scan base-package="com.lina.service,com.lina.aop"/>
<!--开启注解AOP的使用-->
<aop:aspectj-autoproxy proxy-target-class="true"/>
<!--aop:aspectj-autoproxy的底层是由 AnnotationAwareAspectJAutoProxyCreator 实现的，
是基于 AspectJ 的注解适配自动代理生成器。
其工作原理是，aop:aspectj-autoproxy通过扫描找到@Aspect 定义的切面类，再由切面类根据切入点找到目标类的目标方法，再由通知类型找到切入的时间点。-->
```

6.测试类：

```
package com.lina.test;

import com.lina.service.NBAService;
import com.lina.service.TeamService;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Test01 {

    @Test
    public void test01(){
        ApplicationContext ac=new ClassPathXmlApplicationContext("spring.xml");
        TeamService teamService = (TeamService) ac.getBean("teamService");
        teamService.add(1001,"湖人队");
        System.out.println("-----");
        boolean update = teamService.update(888);
    }
}
```

```

        System.out.println("update 结果="+update);
        System.out.println("~~~~~");
        NBAService nbaService = (NBAService) ac.getBean("nbaService");
        nbaService.add(1002,"热火");
        System.out.println("-----");
        boolean update2 = teamService.update(888);
        System.out.println("update 结果="+update2);
    }
}

```

3.5.4 XML方式实现AOP

```

package com.lina.aop;

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

/**
 * 切面类
 */
@Component //切面对象的创建权限依然交给spring容器
@Aspect //aspectj 框架的注解 标识当前类是一个切面类
public class MyAOP {

    public void before(JoinPoint jp){
        System.out.println("AOP前置通知: 在目标方法执行之前被调用的通知");
    }

    public void afterReturn(Object result){
        System.out.println("AOP后置通知: 在目标方法执行之后被调用的通知,result="+result);
    }

    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("AOP环绕方法---目标方法的执行之前");
        Object proceed = pjp.proceed();
        System.out.println("AOP环绕方法---目标方法的执行之后");
        return proceed;
    }

    public void exception(JoinPoint jp,Throwable ex){
        //一般会把异常发生的时间、位置、原有都记录下来
        System.out.println("AOP异常通知: 在目标方法执行出现异常的时候才会别调用的通知, 否则不执行");
        System.out.println(jp.getSignature()+"方法出现异常, 异常信息是: "+ex.getMessage());
    }

    public void myFinally(){
        System.out.println("AOP最终通知: 无论是否出现异常都是最后被调用的通知");
    }
}

```

```

<aop:config>
    <!--声明切入点的表达式,可以声明多个-->
    <aop:pointcut id="pt1" expression="execution(* com.lina.service...add*
(..))"/>
    <aop:pointcut id="pt2" expression="execution(*
com.lina.service...update*(..))"/>
    <aop:pointcut id="pt3" expression="execution(* com.lina.service...del*
(..))"/>
    <aop:pointcut id="pt4" expression="execution(*
com.lina.service...insert*(..))"/>
    <aop:aspect ref="myAOP">
        <aop:before method="before" pointcut="execution(*
com.lina.service...*(..))"></aop:before>
        <aop:after-returning method="afterReturn" pointcut-ref="pt2"
returning="result"></aop:after-returning>
        <aop:after-throwing method="exception" pointcut-ref="pt1"
throwing="ex"></aop:after-throwing>
        <aop:after method="myFinally" pointcut-ref="pt1"></aop:after>
        <aop:around method="around" pointcut-ref="pt2"></aop:around>
    </aop:aspect>

```

4、Spring整合JDBC

4.1 使用spring-jdbc操作数据库

主要内容:学习使用JdbcTemplate API和 如何使用Spring管理 JdbcTemplate

1、创建项目引入依赖

```

<dependencies>
    <!--spring 核心依赖-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.13.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.13.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.23</version>
    </dependency>
    <dependency>
        <groupId>com.mchange</groupId>
        <artifactId>c3p0</artifactId>
        <version>0.9.5.2</version>
    </dependency>
    <!--测试依赖-->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>

```

```

        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
<build>
    <plugins>
        <!--编译插件-->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.0</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>

```

2、测试连接

```

package com.lina.test;

import com.mchange.v2.c3p0.ComboPooledDataSource;

import org.junit.Test;
import org.springframework.jdbc.core.JdbcTemplate;

import java.beans.PropertyVetoException;

public class Test01 {

    @Test
    public void test01() throws PropertyVetoException {
        //创建数据源
        ComboPooledDataSource dataSource=new ComboPooledDataSource();
        dataSource.setDriverClass("com.mysql.cj.jdbc.Driver");
        dataSource.setJdbcUrl("jdbc:mysql://127.0.0.1:3306/springJDBC?
serverTimezone=Asia/Shanghai&characterEncoding=utf8&useUnicode=true&useSSL=false"
);
        dataSource.setUser("root");
        dataSource.setPassword("root");
        //使用JdbcTemplate
        JdbcTemplate template=new JdbcTemplate(dataSource);
        String sql="insert into team(tname,location) value (?,?)";
        int update = template.update(sql,"勇士","金州");
        System.out.println("插入数据的结果: "+update);
    }
}

```

4.2. Spring管理JdbcTemplate

spring整合jdbc的时候我们都是直接让我的dao继承Spring提供的JdbcDaoSupport类，该类中提供了JdbcTemplate模板可用。

示例：


```

package com.lina.dao;

import com.lina.pojo.Team;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.support.JdbcDaoSupport;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import java.util.Map;

public class TeamDao extends JdbcDaoSupport {

    public Map<String, Object> getMany(){
        String sql="select max(tid) as max,min(tid) as min from team";
        return this.getJdbcTemplate().queryForMap(sql);
    }

    public int getCount(){
        String sql="select count(tid) from team";
        //如果查询的列只有唯一一列，queryForObject（sql语句，唯一列的数据类型）
        return this.getJdbcTemplate().queryForObject(sql,Integer.class);
    }
    /**
     * 自己封装处理结果的方法
     * @param resultSet
     * @return
     * @throws SQLException
     */
    public Team handlResult(ResultSet resultSet) throws SQLException {
        Team team=new Team();
        team.settid(resultSet.getInt("tid"));
        team.settName(resultSet.getString("tname"));
        team.setLocation(resultSet.getString("location"));
        return team;
    }

    public List<Team> findByAll() {
        String sql = "select * from team";
        return this.getJdbcTemplate().query(sql, new RowMapper<Team>() {
            @Override
            public Team mapRow(ResultSet resultSet, int i) throws SQLException {
                /*Team team=new Team();
                team.settid(resultSet.getInt("tid"));
                team.settName(resultSet.getString("tname"));
                team.setLocation(resultSet.getString("location"));
                return team;*/
                return handlResult(resultSet);
            }
        });
    }

    public Team findById(int id){
        String sql="select * from team where tid=?";
        return this.getJdbcTemplate().queryForObject(sql, new Object[]{id}, new
RowMapper<Team>() {

```

```

        @Override
        public Team mapRow(ResultSet resultSet, int i) throws SQLException {
            /*Team team=new Team();
            team.settid(resultSet.getInt("tid"));
            team.settName(resultSet.getString("tname"));
            team.setLocation(resultSet.getString("location"));
            return team;*/
            return handleResult(resultSet);
        }
    });
}

    public int insert(Team team){
        String sql="INSERT INTO `springjdbc`.`team` ( `tname`, `location`)
VALUES (?,?)";
        return
this.getJdbcTemplate().update(sql,team.gettName(),team.getLocation());
    }

    public int update(Team team){
        String sql="UPDATE `springjdbc`.`team` SET `tname` = ?, `location` = ?
WHERE `tid` = ?";
        return
this.getJdbcTemplate().update(sql,team.gettName(),team.getLocation(),team.gettid
());
    }

    public int del(int id){
        String sql="delete from team where tid=?";
        return this.getJdbcTemplate().update(sql,id);
    }
}

```

spring的配置文件application.xml中需要创建数据源和给TeamDao中的jdbcTemplate赋值

```

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

    <!--创建数据源-->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="com.mysql.cj.jdbc.Driver">
</property>
        <property name="jdbcUrl"
value="jdbc:mysql://192.168.58.128:3306/springJDBC?
serverTimezone=UTC&characterEncoding=utf8&useUnicode=true&useSSL=false"></property>
        <property name="user" value="root"></property>
        <property name="password" value="root"></property>
    </bean>

    <!--创建JdbcTemplate-->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <!--需要注入数据源-->
        <property name="dataSource" ref="dataSource"></property>

```

```

</bean>
<bean id="teamDao" class="com.lina.dao.TeamDao">
    <!--需要JdbcTemplate-->
    <property name="jdbcTemplate" ref="jdbcTemplate"></property>
</bean>
</beans>

```

测试:

```

package com.lina.test;

import com.lina.dao.TeamDao;
import com.lina.pojo.Team;
import com.mchange.v2.c3p0.ComboPooledDataSource;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;

import java.beans.PropertyVetoException;
import java.util.List;
import java.util.Map;

public class Test01 {
    ApplicationContext ac=new ClassPathXmlApplicationContext("spring.xml");
    @Test
    public void testGetMany() {
        TeamDao teamDao = (TeamDao) ac.getBean("teamDao");
        Map<String, Object> many = teamDao.getMany();
        for (String s : many.keySet()) {
            System.out.println(s+"-----"+many.get(s));
        }
    }
    @Test
    public void testGetCount() {
        TeamDao teamDao = (TeamDao) ac.getBean("teamDao");
        int count = teamDao.getCount();
        System.out.println("count="+count);
    }
    @Test
    public void testFindAll(){
        TeamDao teamDao= (TeamDao) ac.getBean("teamDao");
        List<Team> byAll = teamDao.findByAll();
        for (Team team : byAll) {
            System.out.println(team);
        }
    }
    @Test
    public void testFindById(){
        TeamDao teamDao= (TeamDao) ac.getBean("teamDao");
        Team team = teamDao.findById(1);
        System.out.println(team);
    }
    @Test
    public void test04(){

```

```

TeamDao teamDao= (TeamDao) ac.getBean("teamDao");

int insert = teamDao.del(4);
System.out.println("删除数据的结果是: "+insert);
}
@Test
public void test03(){
    ApplicationContext ac=new ClassPathXmlApplicationContext("spring.xml");
    TeamDao teamDao= (TeamDao) ac.getBean("teamDao");
    Team team=new Team();
    team.settId(2);
    team.settName("勇士");
    team.setLocation("金州");
    int insert = teamDao.update(team);
    System.out.println("修改数据的结果是: "+insert);
}
@Test
public void test02(){
    ApplicationContext ac=new ClassPathXmlApplicationContext("spring.xml");
    TeamDao teamDao= (TeamDao) ac.getBean("teamDao");
    Team team=new Team();
    team.settName("快船");
    team.setLocation("洛杉矶");
    int insert = teamDao.insert(team);
    System.out.println("插入数据的结果是: "+insert);
}

@Test
public void test01() throws PropertyVetoException {
    //创建数据源
    ComboPooledDataSource dataSource=new ComboPooledDataSource();
    dataSource.setDriverClass("com.mysql.cj.jdbc.Driver");
    dataSource.setJdbcUrl("jdbc:mysql://127.0.0.1:3306/springJDBC?
serverTimezone=Asia/Shanghai&characterEncoding=utf8&useUnicode=true&useSSL=false"
);
    dataSource.setUser("root");
    dataSource.setPassword("root");
    //使用JdbcTemplate
    JdbcTemplate template=new JdbcTemplate(dataSource);
    String sql="INSERT INTO `springjdbc`.`team` ( `tname`, `location`)
VALUES (?,?)";
    int update = template.update(sql,"勇士","金州");
    System.out.println("插入数据的结果是: "+update);
}
}

```

5、Spring事务管理

事务原本是数据库中的概念，在jdbc中我们将事务放在了数据访问层。但在实际开发中，一般将事务提升到业务层，即 Service 层。这样做是为了能够使用事务的特性来管理具体的业务。

5.1 Spring事务管理API

Spring 的事务管理，主要用到两个事务相关的接口。

5.1.1 事务管理器接口

事务管理器是 PlatformTransactionManager 接口对象。其主要用于完成事务的提交、回滚，及获取事务的状态信息。

Method Summary	
All Methods	Instance Methods
Abstract Methods	
Modifier and Type	Method and Description
void	<code>commit(TransactionStatus status)</code> Commit the given transaction, with regard to its status.
<code>TransactionStatus</code>	<code>getTransaction(TransactionDefinition definition)</code> Return a currently active transaction or create a new one, according to the definition.
void	<code>rollback(TransactionStatus status)</code> Perform a rollback of the given transaction.

PlatformTransactionManager 接口常用的实现类：

DataSourceTransactionManager：使用 JDBC 或 MyBatis 进行数据库操作时使用。

Spring的回滚方式

Spring 事务的默认回滚方式是：发生运行时异常和 error 时回滚，发生受查(编译)异常时提交。不过，对于受查异常，程序员也可以手工设置其回滚方式。

5.1.2 事务定义接口

事务定义接口 TransactionDefinition 中定义了事务描述相关的三类常量：事务隔离级别、事务传播行为、事务默认超时时限，及对它们的操作。

Field Summary	
Fields	
Modifier and Type	Field and Description
static int	<code>ISOLATION_DEFAULT</code> Use the default isolation level of the underlying database.
static int	<code>ISOLATION_READ_COMMITTED</code> Indicates that dirty reads are prevented; non-repeatable reads are allowed.
static int	<code>ISOLATION_READ_UNCOMMITTED</code> Indicates that dirty reads, non-repeatable reads and phantom reads are allowed.
static int	<code>ISOLATION_REPEATABLE_READ</code> Indicates that dirty reads and phantom reads are prevented; only one read of a particular row is allowed.

5.1.2.1 事务隔离级别常量

这些常量均是以 ISOLATION_开头。即形如 ISOLATION_XXX。

➢ DEFAULT：采用 DB 默认的事务隔离级别。MySQL 的默认为 REPEATABLE_READ；Oracle默认为 READ_COMMITTED。

➢ READ_UNCOMMITTED：读未提交。未解决任何并发问题。

- READ_COMMITTED：读已提交。解决脏读，存在不可重复读与幻读。
- REPEATABLE_READ：可重复读。解决脏读、不可重复读，存在幻读
- SERIALIZABLE：串行化。不存在并发问题。

5.1.2.2 事务传播行为常量

所谓事务传播行为是指，处于不同事务中的方法在相互调用时，执行期间事务的维护情况。如，A 事务中的方法 doSome()调用 B 事务中的方法 doOther()，在调用执行期间事务的维护情况，就称为事务传播行为。事务传播行为是加在方法上的。

事务传播行为常量都是以 PROPAGATION_ 开头，形如 PROPAGATION_XXX。

Propagation.REQUIRED

当前没有事务的时候，就会创建一个新的事务；如果当前有事务，就直接加入该事务，比较常用的设置 Propagation.SUPPORTS

支持当前事务，如果当前有事务，就直接加入该事务；当前没有事务的时候，就以非事务方式执行

Propagation.MANDATORY

支持当前事务，如果当前有事务，就直接加入该事务；当前没有事务的时候，就抛出异常

Propagation.REQUIRES_NEW

创建新事务，无论当前是否有事务都会创建新的

PROPAGATION_NESTED

PROPAGATION_NEVER

PROPAGATION_NOT_SUPPORTED

5.1.2.3 默认事务超时时限

常量 TIMEOUT_DEFAULT 定义了事务底层默认的超时时限，sql 语句的执行时长。

注意，事务的超时时限起作用的条件比较多，且超时的时间计算点较复杂。所以，该值一般就使用默认值即可。

5.2 声明式事务控制

Spring提供的对事务的管理，就叫做声明式事务管理。

如果用户需要使用spring的声明式事务管理，在配置文件中配置即可：不想使用的时候直接移除配置。这种方式实现了对事务控制的最大程度的解耦。

声明式事务管理，核心实现就是基于AOP。

Spring中提供了对事务的管理。开发者只需要按照spring的方式去做就行。
事务必须在service层统一控制。

事务的粗细粒度：

- 细粒度：对方法中的某几行的代码进行开启提交回滚；
- 粗粒度：对整个方法进行开启提交回滚；

Spring中的aop只能对方法进行拦截，所有我们也就针对方法进行事务的控制。

如果只有单条的查询语句，可以省略事务；如果一次执行的是多条查询语句，例如统计结果、报表查询。必须开启事务。

5.3 基于注解的事务

```
package com.lina.service;
import com.lina.dao.TeamDao;
import com.lina.pojo.Team;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@Service
public class TeamService {

    @Autowired
    private TeamDao teamDao;

    /**
     * @Transactional 属性 说明:
     * readOnly:是否只读
     *
     * rollbackFor={Exception.class}: 遇到什么异常会回滚
     *
     * propagation事务的传播:
     * Propagation.REQUIRED:当前没有事务的时候,就会创建一个新的事务;如果当前有事务,就直接加入该事务,比较常用的设置
     * Propagation.SUPPORTS:支持当前事务,如果当前有事务,就直接加入该事务;当前没有事务的时候,就以非事务方式执行
     * Propagation.MANDATORY:支持当前事务,如果当前有事务,就直接加入该事务;当前没有事务的时候,就抛出异常
     * Propagation.REQUIRES_NEW:创建新事务,无论当前是否有事务都会创建新的
     *
     * isolation=Isolation.DEFAULT: 事务的隔离级别: 默认是数据库的隔离级别
     *
     */
    @Transactional(propagation = Propagation.REQUIRED,rollbackFor = {Exception.class})
    public int insert(Team team){
        int num1=teamDao.insert(team);
        System.out.println("第一条执行结果: num1="+num1);
        System.out.println(10/0);
        int num2=teamDao.insert(team);
        System.out.println("第二条执行结果: num2="+num2);
        return num2+num1;
    }
}
```

```
<!--配置文件中添加context约束-->
<context:component-scan base-package="com.lina"/>
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>
<tx:annotation-driven transaction-manager="transactionManager" />
```

```

@Test
    public void test01(){
        ApplicationContext ac=new
ClassPathXmlApplicationContext("application.xml");
        TeamService teamService = (TeamService) ac.getBean("teamService");
        int num=teamService.insert(new Team("凯尔特人","波士顿"));
        System.out.println(num);
    }

```

5.4 基于XML的事务

添加依赖

```

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.6</version>
</dependency>

```

```

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

<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="insert*" propagation="REQUIRED"/>
        <tx:method name="add*" propagation="REQUIRED"/>
        <tx:method name="find*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
</tx:advice>

<aop:config>
    <aop:pointcut id="pt" expression="execution(* com.lina.service...*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="pt" />
</aop:config>

```