

三种jar包

一般从官网上下载的jar包都会有三个包 .jar、sources.jar和javadoc.jar，那这三个分别有什么用呢？

- 1、.jar包大家应该都知道干嘛的，去官网下无非就为了这个。
- 2、sources.jar就是整个jar包的源码，当你用到jar包里的方法时，你想通过“ctrl+鼠标左键”看源码的话就要关联这个文件。
- 3、javadoc.jar就是整个jar包的帮助文档了。

概念

spring是轻量级的开源的JavaEE框架

轻量级是jar包比较少，总体外存少

目的

解决企业应用开发的复杂性

核心部分

IOC和Aop

IOC：控制反转，把创建对象过程交给Spring进行管理

Aop：面向切面，不修改源代码进行功能增强

特点

方便解耦，简化开发

Aop编程支持

方便程序测试

方便和其它框架进行整合

方便进行事物操作

降低API开发

入门案例

下载spring5，网址spring.io

Baidu 检测到当前网页不是中文网页，是否需要翻译成中文？ 翻译 不翻译 不再提示

spring

Why Spring Learn Projects Training Support Community

Spring make product

WHY SPRING QU

- Overview
- Spring Boot
- Spring Framework
- Spring Cloud
- Spring Cloud Data Flow
- Spring Data
- Spring Integration
- Spring Batch
- Spring Security
- View all projects
- DEVELOPMENT TOOLS
- Spring Tools 4

Baidu 检测到当前网页不是中文网页，是否需要翻译成中文？ 翻译 不翻译 不再提示

Spring Boot

Spring Framework

Spring Data >

Spring Cloud >

Spring Cloud Data Flow

Spring Security >

Spring for GraphQL

Spring Session >

Spring Integration

Spring HATEOAS

Spring REST Docs

Spring Batch

Spring AMQP

Spring CredHub

Spring Flo

Spring Framework 5.3.21

OVERVIEW LEARN SUPPORT

Documentation

Each **Spring project** has its own; it explains in great details how you can use **project features** and what you can achieve with them.

5.3.21 CURRENT GA	Reference Doc.	API Doc.
6.0.0-SNAPSHOT SNAPSHOT	Reference Doc.	API Doc.
6.0.0-M4 PRE	Reference Doc.	API Doc.
5.3.22-SNAPSHOT SNAPSHOT	Reference Doc.	API Doc.
5.2.23.BUILD-SNAPSHOT SNAPSHOT	Reference Doc.	API Doc.
5.2.22.RELEASE GA	Reference Doc.	API Doc.

snapshot是快照不稳定，ga是稳定的

Spring Framework

GitHub - spring-projects/spring-framework

github.com/spring-projects/spring-framework

Spring provides everything required beyond the Java programming language for creating enterprise applications for a wide range of scenarios and architectures. Please read the [Overview](#) section as reference for a more complete introduction.

Code of Conduct

This project is governed by the [Spring Code of Conduct](#). By participating, you are expected to uphold this code of conduct. Please report unacceptable behavior to spring-code-of-conduct@pivotal.io.

Access to Binaries

For access to artifacts or a distribution zip, see the [Spring Framework Artifacts](#) wiki page.

下载地址

<https://repo.spring.io/ui/native/libs-release/org/springframework/spring/>

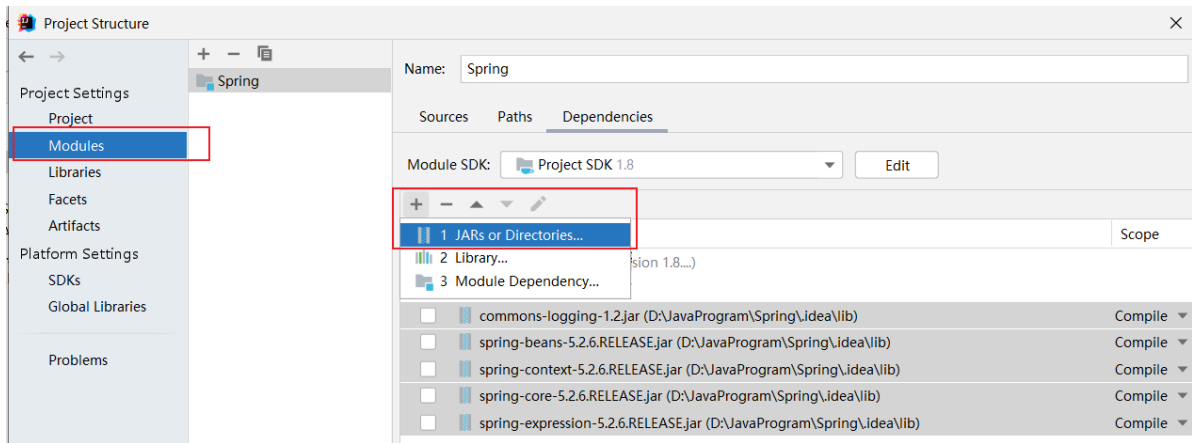
下载5.2.6

打开idea，创建普通项目

导入Spring相关的jar包

spring基本包，beans，context，core，expression

还需要一个额外的日志包 commons jar，链接为https://blog.csdn.net/qq_40092521/article/details/103893059



创建对象

创建Spring配置文件，在配置文件配置创建的对象

Spring配置文件使用xml格式

然后利用bean



id是名字，class是路径

```

8 public class test {
9     @Test
10    public void testAdd()
11    {
12        // 加载spring配置文件xml
13        ApplicationContext context=new ClassPathXmlApplicationContext("bean1.xml");//在src下就直接bean1.xml
14        // 获取配置创建文件的对象
15        User user = context.getBean("userName", User.class);//前面是xml中的id，后面是转换的类
16        System.out.println(user);
17        user.add();
18    }
19 }

```

IOC

inversion of control 控制反转，简称ioc

IOC底层原理：xml解析，工厂模式，反射

目的：降低代码之间的耦合度

过程：把对象创建和对象之间的调用过程交给Spring进行管理

原始方案：要修改所有的service

原始方式

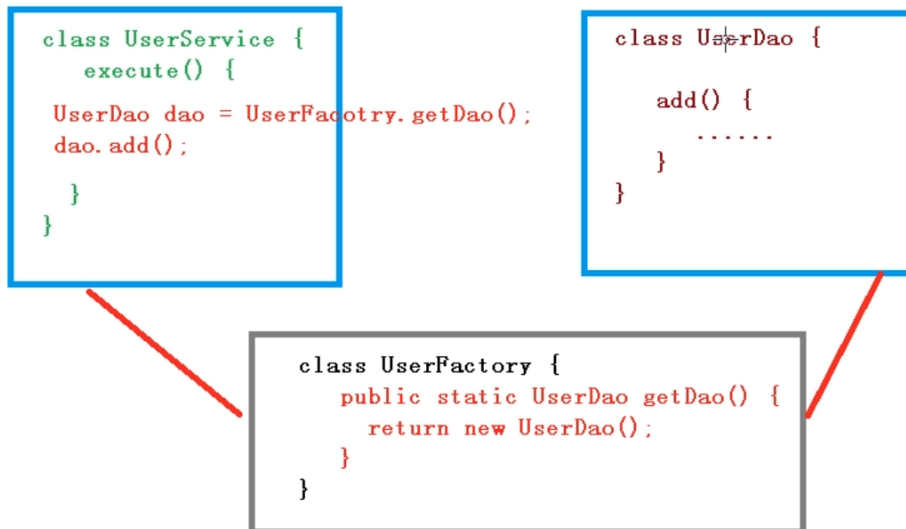
```
class UserService {  
    execute() {  
        UserDao dao = new UserDao();  
        dao.add();  
    }  
}
```

```
class UserDao {  
    add() {  
        ....  
    }  
}
```

耦合度太高了

工厂模式：修改只用修改工厂就好了

工厂模式



详细过程：

第一步：

xml配置文件，配置创建对象

第二步：有service类和dao类，创建工厂类

```
class UserFactory {  
    public static UserDao getDao() {  
        String classValue = class属性值; //xml解析  
        Class clazz = Class.forName(classValue) //通过反射创建对象  
        return (UserDAO) clazz.newInstance();  
    }  
}
```

修改配置文件路径就可以了，降低耦合度

IOC接口

bean意思是让计算机自动生成类

两个接口

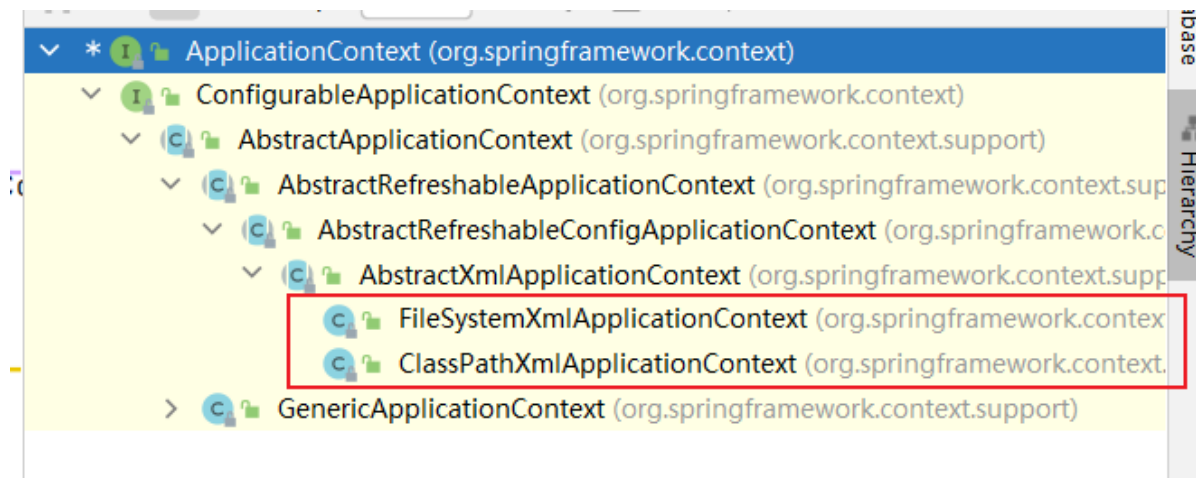
BeanFactory：IOC容器基本实现，是Spring内部的使用接口，不提供给开发人员

特点是加载配置文件时候不会创建对象，在获取对象（使用）才去创建对象

ApplicationContext: BeanFactory接口的子接口，提供更多强大的接口，一般由开发人员进行使用

特点是加载配置文件时候就会把在配置文对象进行创建

注意要有无参构造方法!!!!!! 否则实例化不了对象



file是绝对路径，class是相对路径

IOC操作Bean管理

什么是Bean管理?

Spring创建对象，Spring注入属性

操作有两种方式

基于xml配置文件方式实现

创建对象

```
<!--配置User对象创建-->
<bean id="userName" class="com.atguigu.spring5.User">
</bean>
```

在spring配置文件中，使用bean标签，标签里面添加对应属性，就可以实现对象创建

在bean标签有很多属性

id: 唯一标识

class: 类全路径（包类路径）

name: 可以有特殊符号，很少使用，了解一下

注意创建对象默认是执行无参数构造方法，java基础若写了有参构造方法，则没有默认无参构造方法，要自己写。

注入属性

DI

DI Dependency Injection：依赖注入，注入属性，
创建类，属性，方法

```
/*
演示使用set方法进行注入属性
*/
public class Book
{
    //创建属性
    private String bname;
    private String bauthor;

    //创建属性对应的set方法
    public void setName(String bname) {
        this.bname = bname;
    }
    public void setBauthor(String bauthor) {
        this.bauthor = bauthor;
    }
}
```

在spring配置文件配置对象创建，配置属性注入

第一种方式：使用set方式注入

set方法注入

```
<!-- set方法注入属性-->
<bean id="book" class="com.atguigu.spring5.Book">
    <!-- 使用property完成属性注入
        name:类里面属性的名称，value向属性注入的值
    -->
    <property name="bname" value="庄消津"></property>
    <property name="bauthor" value="作者"></property>
</bean>
```

使用p名称空间注入，可以简化基于xml配置方式

第一步添加p名称空间在配置文件中

第二步进行属性注入，在bean标签里面进行操作

```

<!-- set方法注入属性-->
<bean id="book" class="com.atguigu.spring5.Book">
    <!-- 使用property完成属性注入
        name:类里面属性的名称, value向属性注入的值
    -->
    <property name="bname" value="p庄涓津"></property>
    <property name="bauthor" value="作者"></property>
</bean>
<!-- p写法, set方法注入属性-->
<bean id="pbook" class="com.atguigu.spring5.Book" p:bname="庄涓津" p:bauthor="作者"></bean>

```

字面量：设置null值

```

<bean id="book" class="com.atguigu.spring5.Book">
    <property name="bname" value="庄涓津"></property>
    <!-- 设置空值-->
    <property name="bauthor" >
        <null/>
    </property>
</bean>

```

特殊符号或者null

属性值包含特殊符号<或者>

需要转义符号 < >都要加个;表示<和>

或者带特殊符号内容写到cdata里面去

```

<property name="bname" >
    <value>
        <![CDATA[
            <><><><>
        ]]>
    </value>
</property>

```

```

<property name="bname" value="&lt;&gt;"></property>

```

第二种方式：使用有参数构造进行注入

构造方法注入

如果方法重写了constructor-arg元素type属性默认数据类型是String，可以用type=某个基本数据类型来识别想要的方法重写

index或者name在构造方法没有歧义可不写，否则对于string int, int string这种参数前后不同

构造方法顺序是随意的，调用的困难是参数string int也可能是int string

属性名

```

<!--有参构造注入属性    -->
<bean id="orders" class="com.atguigu.spring5.Orders">
    <constructor-arg name="oname" value="电脑"></constructor-arg>
    <constructor-arg name="address" value="东风"></constructor-arg>
</bean>

```

下标

```

<!--有参构造注入属性    -->
<bean id="orders" class="com.atguigu.spring5.Orders">
    <constructor-arg index="0" value="电脑"></constructor-arg>
    <constructor-arg index="1" value="电脑"></constructor-arg>
</bean>

```

```

<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"
    xmlns:p="http://www.springframework.org/schema/p">

```

###

属性是对象

创建service类和dao类

```

package com.atguigu.spring5.dao;

public interface UserDao {
    public void update();
}

```

```

package com.atguigu.spring5.dao;

public class UserDaoImpl implements UserDao{
    @Override
    public void update() {
        System.out.println("dao update-----");
    }
}

```

```

package com.atguigu.spring5.service;

import com.atguigu.spring5.dao.UserDao;
import com.atguigu.spring5.dao.UserDaoImpl;

public class UserService {
    //创建UserDao类型属性,生成set方法
    private UserDao userDao;

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

    public void add()
    {

```



```

        System.out.println("service add-----");
        userDao.update();
    }
}

```

在service调用dao的方法

级联赋值

property引用外部bean

```

<!-- service和dao对象的创建-->
<bean id="userService" class="com.atguigu.spring5.service.UserService">
<!-- 注入userDao对象-->
<!-- 不能是value，得是ref，因为是引用数据类型-->
<!-- ref属性，创建userDao对象的bean标签id值-->

    <property name="userDao" ref="userDaoImpl"></property>或者是
    //前提是需要get方法获取userDao
    <property name="userDao.属性" value="?"></property>

</bean>

<!-- 得是userdao的实现类-->
<bean id="userDaoImpl" class="com.atguigu.spring5.dao.UserDaoImpl">
    <property name="?" value="?"></property>
</bean>

```

```

package com.atguigu.spring5.service;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.testng.annotations.Test;

public class testService {
    @Test
    public void test()
    {
        ApplicationContext context=new
        ClassPathXmlApplicationContext("bean2.xml");
        UserService userService = context.getBean("userService",
        UserService.class);
        userService.add();
    }
}

```

内部bean

引用bean可以写成内部bean

将上面改写成

外部bean比较清晰，大多时候都是写外部bean

还可以用或者

local属性的话是在当前的xml找，bean属性的是可以在不同的xml找，上面的等同于bean属性的作用

多个值

默认是string类型，若是要指定数据类型可以在元素

或者在array或者list，map中的value元素的属性为

前面注入属性是注入string或者对象，现在是注入数组类型属性，list集合类型属性，map集合类型属性，以上都是针对多个值

```
<!-- 集合类型属性的注入-->
<bean id="stu" class="com.atguigu.src2.Stu">
<!-- 数组类型属性注入-->
  <property name="course" >
<!-- array标签-->
    <array>
      <value>java课程</value>
      <value>数据库课程</value>
    </array>
  </property>
  <property name="list">
```

```

        <list>
            <value>张三</value>
            <value>李四</value>
        </list>
    </property>
    <property name="map">
<!--      map数据类型用map，不用value用entry表示key-value机制-->
        <map>
            <entry key="JAVA" value="java"></entry>
        </map>
    </property>
</bean>

```

对象数组

在集合里面设置对象类型值

例如一个学生有多门课则

```

public class Course {
    private String cname;

    public void setCname(String cname) {
        this.cname = cname;
    }
}

```

```

public class Stu {
    //一个学生有多门课
    private List<Course> courseList;

    public void setCourseList(List<Course> courseList) {
        this.courseList = courseList;
    }
}

```

```

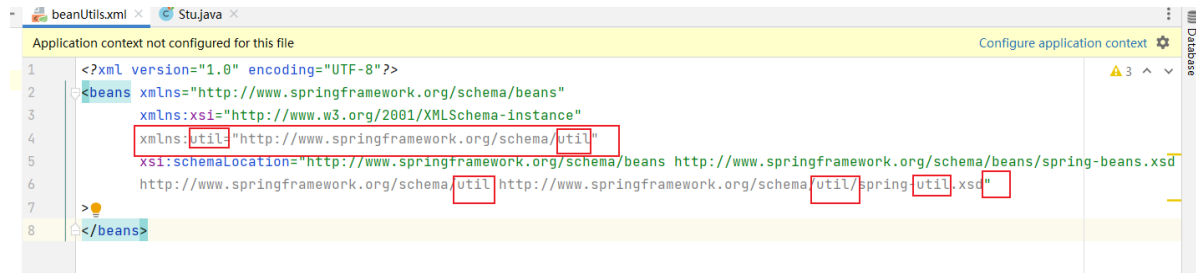
<bean id="stu" class="com.atguigu.src2.Stu">
<!--      注入list集合类型，值是对象-->
    <property name="courseList">
        <list>
<!--      利用ref标签-->
            <ref bean="course0"></ref>
            <ref bean="course1"></ref>
        </list>
    </property>
</bean>

<!--      创建多个course对象-->
<bean id="course0" class="com.atguigu.src2.Course">
    <property name="cname" value="spring4"></property>
</bean>
<bean id="course1" class="com.atguigu.src2.Course">
    <property name="cname" value="spring5"></property>
</bean>

```

集合通用方法util

在spring配置文件中引入名称空间util



写一条util

以及

把所有的beans改为util

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

使用util标签完成list集合注入提取

```
<!-- 提取list集合类型属性注入-->
<util:list id="list">
  <value>第一本书</value>
  <value>第二本书</value>
  <value>....</value>
</util:list>
```

对集合进行使用

```
<!-- 提取list集合类型属性注入使用-->
<bean id="book" class="com.atguigu.src2.Book">
  <property name="listBook" ref="list"></property>
</bean>
```

检查注入

设置方法注入的不足之一是无法确定一个属性将会被注入，意思就是说作用在set方法上

bean元素有属性dependency-check，有四种属性值，none不依赖检查

simple检查基本数据类型和集合类型

objects检查引用数据类型

all检查任意类型

没有注入会抛出异常UnsatisfiedDependencyException

不能检查是否为空，其实就是判断有没有new对象或者set属性

而既然Spring3中放弃使用了dependency-check属性一定就会有替代它的功能出现。

查了下资料，果然，在spring3中替代dependency-check有4条建议：

使用构造方法（使用构造方法注入替代setter注入）专门用来确认特定属性被初始化
用init方法初始化setter的属性

在需要强制进行初始化的setters上标注@Required

使用@Autowired-driven 注入也可以实现

两种Bean

普通Bean在Spring 配置文件定义bean类型就是返回类型

工厂Bean在Spring配置文件定义bean类型可以和返回类型不一样

创建类作为工厂Bean，实现接口FactoryBean

实现接口里面的方法，在实现方法定义返回bean类型

```
package com.atguigu.src3;

import org.springframework.beans.factory.FactoryBean;

public class facBean implements FactoryBean<course>{
    // 返回对象
    @Override
    public com.atguigu.src3.course getObject() throws Exception {
        return null;
    }

    //Bean类型,在? 修改
    @Override
    public Class<?> getObjectType() {
        return null;
    }

    @Override
    public boolean isSingleton() {
        // 设置返回的Bean是否为单例
        return FactoryBean.super.isSingleton();
    }
}
```

继承AbstractFactorBean也可以

```

public class facBean1 extends AbstractFactoryBean<course> {
    @Override
    public Class<?> getObjectType() {
        return null;
    }

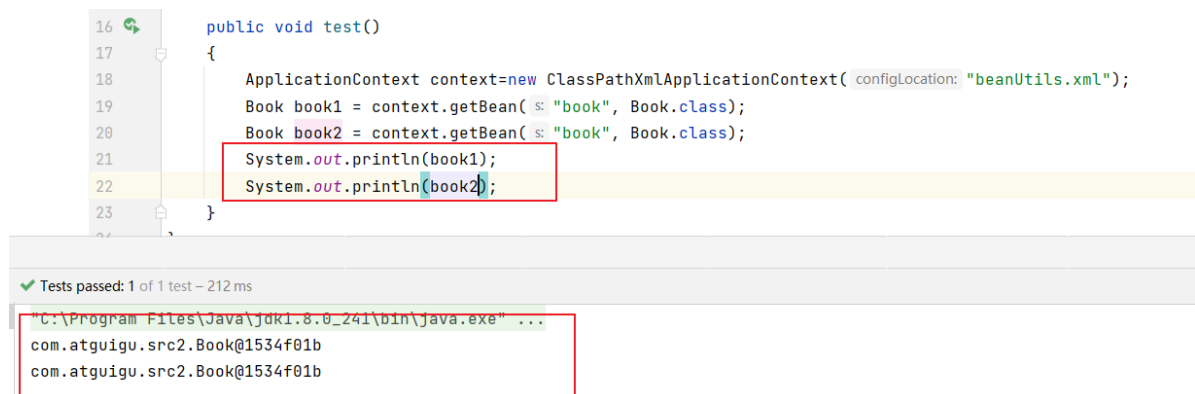
    @Override
    protected com.atguigu.src3.course createInstance() throws Exception {
        return null;
    }
}

```

bean作用域

在Spring里面，设置创建bean实例是单实例还是多实例

默认是单实例对象



```

public void test()
{
    ApplicationContext context=new ClassPathXmlApplicationContext( configLocation: "beanUtils.xml");
    Book book1 = context.getBean( s: "book", Book.class);
    Book book2 = context.getBean( s: "book", Book.class);
    System.out.println(book1);
    System.out.println(book2);
}

```

Tests passed: 1 of 1 test – 212 ms

```

C:\Program Files\Java\jdk1.8.0_241\bin\java.exe" ...
com.atguigu.src2.Book@1534f01b
com.atguigu.src2.Book@1534f01b

```

在spring配置文件bean标签里面有属性用于设置单实例还是多实例

scope属性值

默认值为singleton，表示单实例对象

prototype表示多实例对象

还有一定的区别

singleton时候，加载spring配置文件时候就会创建单实例对象

prototype时候，不是在加载spring配置文件时候就会创建多实例对象，

getBean才会创建对象

还有两个值作为了解即可，request和session

```

<bean id="book" class="com.atguigu.src2.Book" scope="prototype">
    <property name="listBook" ref="list"></property>
</bean>

```

bean生命周期

生命周期：从对象创建到销毁的过程

1首先通过构造器创建bean实例（无参构造）

2其次为bean的属性设置值和对其它bean引用（调用set方法）

3把bean的实例传递bean后置处理器的方法

4调用bean的初始化方法（需要进行配置）

5把bean的实例传递bean后置处理器的方法

6bean可以使用了（对象获得了）

7当容器关闭了，调用bean的销毁方法（需要进行配置销毁的方法）

首先通过构造器创建bean实例（无参构造）

```
public Orders() {  
    System.out.println("第一步,无参构造");  
}
```

其次为bean的属性设置值和对其它bean引用（调用set方法）

```
public void setName(String oname) {  
    this.oname = oname;  
    System.out.println("第二步,执行set方法");  
}
```

把bean的实例传递bean后置处理器的方法

```
public class MyBeanPost implements BeanPostProcessor {  
    //去BeanPostProcessor中复制两个方法  
    //    @Nullable  
    //    default Object postProcessBeforeInitialization(Object bean, String  
    beanName) throws BeansException {  
    //        return bean;  
    //    }  
    //  
    //    @Nullable  
    //    default Object postProcessAfterInitialization(Object bean, String beanName)  
    throws BeansException {  
    //        return bean;  
    //    }  
    //改成  
    @Override  
    public Object postProcessBeforeInitialization(Object bean, String beanName)  
    throws BeansException {  
        System.out.println("初始化之前");  
        return bean;  
    }  
  
    @Override  
    public Object postProcessAfterInitialization(Object bean, String beanName)  
    throws BeansException {  
        System.out.println("初始化之后");  
        return bean;  
    }  
}
```

```
}
```

调用bean的初始化方法（需要进行配置）

```
public void initMethod()
{
    System.out.println("第三步,执行初始化方法");
}
```

把bean的实例传递bean后置处理器的方法

```
<bean id="orders" class="com.atguigu.src4.Orders" init-method="initMethod"
destroy-method="destroyMethod"></bean>
<!-- 配置后置处理器-->
<bean id="myBeanPost" class="com.atguigu.src4.MyBeanPost"></bean>
```

```
<bean id="orders" class="com.atguigu.src4.Orders" init-method"initMethod" destroy-method"destroyMethod"></bean>
```

bean可以使用了（对象获得了）

```
//ApplicationContext没有close方法
ClassPathXmlApplicationContext context=new
ClassPathXmlApplicationContext("bean4.xml");
Orders orders = context.getBean("orders",Orders.class);
System.out.println("第四步,对象获取");
```

当容器关闭了，调用bean的销毁方法（需要进行配置销毁的方法）

```
//手动进行销毁
context.close();
```

自动装配

当一个Bean需要访问另一个Bean时，你可以显式指定引用装配它，但是，如果你的容器能够自动装配Bean，就可以免去手工配置的麻烦

在Spring框架xml配置中共有5种自动装配：

no：默认的方式是不进行自动装配的，通过手工设置ref属性来进行装配bean。

byName：通过bean的名称进行自动装配，如果一个bean的 property 与另一bean 的name 相同，就进行自动装配。

byType：通过参数的数据类型进行自动装配。

constructor：利用构造函数进行装配，并且构造函数的参数通过byType进行装配。复杂在于为每一个构造程序的每一个参数都找到一个类型兼容的Bean，然后将选择具有最多匹配参数的构造程序。注意最多！！

autodetect：自动探测，如果有构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。

根据指定装配规则（属性名称或者属性类型），Spring自动将配置的属性值进行注入

演示自动装配过程

```
<!-- 实现自动装配, bean标签属性autowire, autowire属性常用两个值, byName根据属性名称注入, byType根据属性类型注入-->
<!-- byName根据属性名称注入, 注入值bean的id和类属性名称一样-->
    <bean id="emp" class="com.atguigu.src5.Emp" autowire="byName"></bean>
    <bean id="dept" class="com.atguigu.src5.Dept"></bean>
<!-- 如果用byType的话-->
<!-- <bean id="dept" class="com.atguigu.src5.Dept"></bean>-->
<!-- <bean id="dept1" class="com.atguigu.src5.Dept"></bean>-->
<!-- 由于两个class一样就会报错-->
```

自动装配如果超过一个候选的自动装配Bean会抛出UnsatisfiedDependencyException异常, 如果为byName或者byType找不到Bean就属性保持为位置状态, 可能导致空指针异常, 如果希望自动装配无法装配的Bean时得到通知, 要检查注入dependency-check属性设置为objects或者all

外部属性文件

不用外部属性文件则

```
<!--直接配置连接池-->
<bean id="dataSource0" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
    <property name="url" value="jdbc:mysql://localhost:3306/library"></property>
    <property name="username" value="root"></property>
    <property name="password" value="mdjfbzyj515"></property>
    <property name="initialSize" value="10"></property>
    <property name="maxActive" value="10"></property>
</bean>
```

否则利用context命名空间

```
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">
```

以及

来自druid.properties文件

```
url=jdbc:mysql://localhost:3306/library?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
username=root
password=mdjfbzyj515
driverClassName=com.mysql.jdbc.Driver
initialSize=10
maxActive=10
```

```

<!-- 引入外部属性文件-->
<context:property-placeholder location="classpath:druid.properties">
</context:property-placeholder>
<!-- 利用key-value机制-->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="${driverClassName}">
</property>
    <property name="url" value="${url}"></property>
    <property name="username" value="${username}"></property>
    <property name="password" value="${password}"></property>
    <property name="initialSize" value="${initialSize}"></property>
    <property name="maxActive" value="${maxActive}"></property>
</bean>

```

基于注解方式

什么是注解？

注解是代码特殊标记，格式@注解名称（属性名称=属性值，.....）

注解的范围？

注解作用在类，方法，属性上面

注解的目的？

简化xml配置

过程

@Component业务特殊组件层

@Service用于服务层，处理业务逻辑

@Controller用于呈现层

@Repository用于持久层，数据库访问层

四个实现功能一样，用于不同开发程序层次

引入依赖操作要导jar包

此电脑 > Data (D:) > Spring > spring-framework-5.2.6.RELEASE > libs			
名称	修改日期	类型	大小
spring-aop-5.2.6.RELEASE.jar	2020/4/28 8:14	Executable Jar File	364 KB

开启组件扫描

会对com.atguigu以下所有目录进行扫描

```

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"

<!--      开启组件扫描-->
<!--      如果扫描多个包使用逗号隔开
           或者写上层目录
-->
<context:component-scan base-package="com.atguigu"></context:component-scan>

```

过滤

有一定过滤，将使用默认过滤关闭，将包含过滤类型为注解annotation（还有exclude-filter不包含过滤），表达式为某一个注解

属性值annotation是注解类型

assignable接口

regex正则表达式

aspectj是AspectJ切入点表达式匹配类

```

<!--      对包含某个注解才进行扫描，如对Component-->
<context:component-scan base-package="com.atguigu" use-default-
filters="false">
    <context:include-filter type="annotation"
expression="org.springframework.stereotype.Component"/>
</context:component-scan>

```

类注解

```

import org.springframework.stereotype.Component;

@Component(value = "user") //等价于bean中的id，不用class，不写括号及里面内容默认为类名，
开头小写
public class User {
    public void add()
    {
        System.out.println("User add");
    }
    @Test
    public void test()
    {
        ApplicationContext context=new
ClassPathXmlApplicationContext("bean7.xml");
        User user = context.getBean("user", User.class);
        user.add();
    }
}

```

属性注解

@Value是赋值有多种用法上查



如果value的占位符名字需要变化, 防止大量修改代码, 可以考虑自定义注解

查找bean时的过滤流程:

@Value ---> 如果没有@Value, 根据type查找 ---> isAutowireCandidate -----> generic ---> qualifier
---> @Primary --> @Priority --> byName (dependencyName)

1. resolveDependency()方法源码解析

两种情况,

情况一: 走缓存

情况二: 没缓存

1) 如果有@Value注解, 就解析返回获取到的值。@value返回的可能是字符串, 也可能是对象。是各字符串时, 先解析占位符, 再解析Spring EL表达式("#{}").如果是字符串后, 解析的结果, 会根据TypeConverter再解析, 如果类型匹配或者可以转换, 就返回成功, 否则会报错。

2) 先bytype 后byname, 中间有6步过滤

findAutowireCandidates 返回的map中, value可能是bean对象 (已实例化), 也可能是bean的class对象。

2. findAutowireCandidates()方法源码解析

1) 如果找到了多个bean, 如果有bean有@Primary注解, 就返回这个, 如果有多个Primary, 就报错。如果没有primay,

2) 就看有没有@Priority, 有的话返回优先级最高的。数字越小, 优先级越高。 @Priority注解是增加在类上的, 如果是一个类注册了多个实例, 那这些实例的优先级都是一样的。如果是一个接口的多个实现分别增加优先级是可以的。

3) 否则就根据bean.getDependencyName (属性字段的名称或者set参数的名称来确定的) 来确定。

如何根据类型找到所有bean的名字:

找到bean的名字

bean名字对应的beandefinition是否是支持自动注入, 可以自动注入的才能被筛选出来

@Autowired 根据属性类型进行自动装配，一个接口可以有多个实现类，所以有时需要属性名称来进行注入

@Qualifier 根据属性名称进行注入，要跟@Autowired一起使用

@Resource 根据属性类型或者属性名称进行注入

//javax.annotation.Resource, Resource不是spring包中的，是java扩展包的

针对基本数据类型使用的注解String, long

@Value (value="?.") //可以给其赋值

```
public interface UserDao {  
    public void add();  
}
```

第一步把service和dao对象创建，在service和dao添加创建对象注解

第二步在service注入dao对象，在service类添加dao类型属性，在属性上使用注解

```
@Service(value = "userDaoImp1")  
public class UserDaoImp implements UserDao{  
    @Override  
    public void add() {  
        System.out.println("UserDaoImp add");  
    }  
}
```

```
@Service  
public class UserService {  
  
    @Autowired  
    //不需要set方法,而且这个接口会去实现userDao=new UserDaoImp,但如果多个实现类的话需要具体到名称  
    @Qualifier(value = "userDaoImp1")  
    private UserDao userDao;  
  
    // @Resource 根据类型注入  
    //根据名称注入  
    @Resource(name = "userDaoImp1")  
    private UserDao userDaoNext;  
  
    @Value(value = "Hello world")  
    private String str;  
  
    public void excute()  
    {  
        System.out.println("service add "+str);  
        userDao.add();  
    }  
    @Test  
    public void test()
```

```

{
    ApplicationContext context=new
    ClassPathXmlApplicationContext("bean8.xml");
    UserService userService = context.getBean("userService",
    UserService.class);
    userService.excute();
}
}

```

检查注入

@Required 应用于 bean 属性 setter 方法。此注解仅指示必须在配置时使用 bean 定义中的显式属性值或使用自动装配填充受影响的 bean 属性。如果尚未填充受影响的 bean 属性，则容器将抛出 BeanInitializationException。

注意：

\1. 这里只能在setter方法上加@Required

\2. 如果任何带有@Required的属性未设置的话 将会抛出BeanInitializationException异常

@Required和@Autowired的区别：

	@Required	@Autowired
区别	1.@Required作用在Setter方法上（用于检查一个Bean的属性的值在配置期间是否被赋予或设置(populated)) 2.@Required作用在Setter方法上就 必须赋值 ，否则容器就会抛出一个 BeanInitializationException 异常。	1.@Autowired 可以作用在 Setter 方法中，属性，构造函数中 2.可以使用 @Autowired 的 (required=false) 选项关闭默认行为。也就是被标注的属性不会被赋值
联系	1.@Required作用在Setter方法上需要生成Setter方法	1.@Autowired 作用在 Setter 方法 也需要生成Setter方法 2.@Autowired 作用在 属性上 ，则可以省略Setter方法，根据Bean类型来注入

完全注解

意思就是不用到配置文件

创建配置类，替代xml配置文件

```

@Configuration //作为配置类,替代xml配置文件
@ComponentScan(basePackages = "com.atguigu") //扫描范围
public class SpringConfiguration {
}

```

加载配置类有一定区别

```

public class Test {
    @org.testng.annotations.Test
    public void test()
    {
        //加载配置类
        ApplicationContext context=new
        AnnotationConfigApplicationContext(SpringConfiguration.class);
    }
}

```

AOP

详细学习代码

https://blog.csdn.net/zidieg/article/details/121889930?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522166055723816782391841295%2522%252C%2522scm%2522%253A%25220140713.130102334..%2522%257D&request_id=166055723816782391841295&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~baidu_landing_v2~default-1-121889930-null-null.142^v40^new_blog_pos_by_title,185^v2^control&utm_term=spring%E7%9A%84aop%E7%AC%94%E8%AE%B0&spm=1018.2226.3001.4187

对方法的操作

https://blog.csdn.net/feiying0canglang/article/details/120711774?ops_request_misc=&request_id=&biz_id=102&utm_term=proceedjoinpoint&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-4-120711774.142^v40^new_blog_pos_by_title,185^v2^control&spm=1018.2226.3001.4187

什么是AOP?

Aspect Oriented Programming面向切面编程

面向切面编程，利用AOP可以对业务逻辑的各个部分进行隔离，即不通过修改源代码方式添加新的功能。

AOP底层使用动态代理

两种情况实现AOP

第一种情况是有接口情况，使用JDK动态代理

创建接口实现类代理对象，增强类的方法

动态代理
有接口

```

interface UserDao {
    public void login();
}

```

```

class UserDaoImpl implements UserDao {
    public void login() {
        //登录实现过程
    }
}

```

JDK动态代理

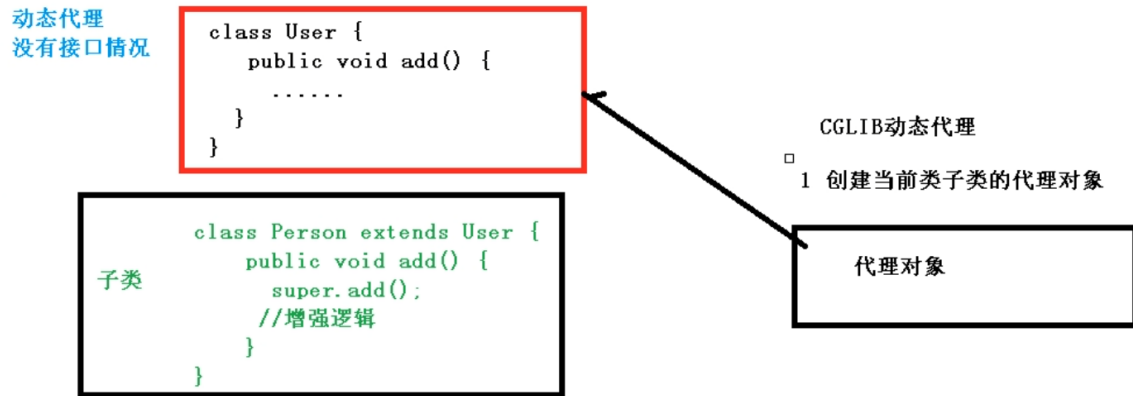
1 创建UserDao接口实现类代理对象

代理对象



第二种情况没有接口情况，使用CGLIB动态代理

创建子类的代理对象，增强类的方法



JDK动态代理代码示例

创建接口，定义方法

```
public interface UserDao {  
    public int add(int a,int b);  
    public String update(String str);  
}
```

创建接口实现类，定义方法

```
public class UserDaoImp implements UserDao{  
    @Override  
    public int add(int a, int b) {  
        System.out.println("add方法执行");  
        return a+b;  
    }  
  
    @Override  
    public String update(String str) {  
        System.out.println("update方法执行");  
        return str;  
    }  
}
```

Proxy代理类的方法即newProxyInstance即new代理实例， 返回的是指定接口的代理类的实例

该方法有三个参数 (ClassLoader loader,类<?>[] interfaces,InvocationHandler h)

第一个参数，main方法所在类的类加载器

第二个参数，类数组存放的是接口的类

第三个参数，实现接口InvocationHandler动态代理，即创建代理 对象，写增强的方法

```
//jdk代理  
public class JdkProxy {  
    public static void main(String[] args) {  
        //接口的类的数组  
        Class[] interfaces={UserDao.class};  
        //创建接口实现类代理对象
```



```

//      Proxy.newProxyInstance(JdkProxy.class.getClassLoader(), interfaces, new
InvocationHandler() { //匿名内部类
//          @Override
//          public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
//              return null;
//          }
//      })
//代理类
    UserDaoImp userDaoImp=new UserDaoImp();
    UserDao o = (UserDao)
Proxy.newProxyInstance(JdkProxy.class.getClassLoader(), interfaces, new
UserDaoProxy(userDaoImp));
    System.out.println(o.add(1,2));
}
}

//创建代理对象，接口了动态代理
class UserDaoProxy implements InvocationHandler
{
    //先传入接口实现类
    private Object object;
    //构造方法初始化
    public UserDaoProxy(Object object)
    {
        this.object=object;
    }
    //实行方法，用来增强的逻辑

    @Override
        //          代理对象          当前的方法          当前方法的参数
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {

            //方法之前做的处理
            System.out.println("方法执行前输出名字以及参数"+method.getName()+" "+
Arrays.toString(args));
            //执行的方法          代理类    参数
            Object invoke = method.invoke(object, args);
            //方法之后做的处理
            System.out.println("方法之后执行");
            return invoke;
        }
}
}

```

操作术语

连接点

类里面可以被增强的方法叫做连接点

切入点

实际被真正增强的方法叫做切入点

通知（增强）

实际增强的逻辑部分称为通知或者增强

通知有多种类型

前置通知 前

后置通知 后（返回通知AfterReturning） 有异常不执行，是返回才执行

环绕通知 前后

异常通知 catch

最终通知 finally （After）

切面

把通知应用到切入点过程，也就是动作过程

AOP操作（准备）

Spring框架一般都是基于AspectJ实现AOP操作

什么是AspectJ?

AspectJ不是Spring组成部分，独立于AOP框架，一般把AspectJ和Spring框架一起使用，进行AOP操作

基于AspectJ实现AOP操作

引入依赖

← → ↶ ↷ 此电脑 > Data (D:) > Spring > spring-framework-5.2.6.RELEASE > libs				
名称	修改日期	类型	大小	
spring-aop-5.2.6.RELEASE-sources.jar	2020/4/28 8:20	Executable Jar File	353 KB	
spring-aspects-5.2.6.RELEASE.jar	2020/4/28 8:15	Executable Jar File	47 KB	

切入点表达式

作用：切入点表达式作用，知道对哪一个类里面的哪个方法进行增强

语法：

execution（权限修饰符 返回类型 类全路径.方法名称 （参数列表））

举例1：对com.atguigu.dao.BookDao类里面的add进行增强

execution（* com.atguigu.dao.BookDao.add(.....) // *表示任意修饰符，返回类型可以省略

对com.atguigu.dao.BookDao类所有方法进行增强

execution（* com.atguigu.dao.BookDao.*(.....) // *表示所有方法

对com.atguigu.dao包中所有类的所有方法进行增强

execution（* com.atguigu.dao.* . *(.....)

基于注解方式实现（主要）

1创建类，在类里面定义方法

2创建增强类（编写增强逻辑）

3进行通知的配置

4在spring配置文件中，开启注解扫描（xml配置）

```
xmlns:context="http://www.springframework.org/schema/context"
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd

<!--      开启注解扫描-->
<context:component-scan base-package="com.atguigu.Aop2"></context:component-
scan>
```

5使用注解创建User和Userproxy对象

```
@Component
public class User {}
```

```
//增强User
@Component
public class UserProxy {
```

6在增强类上面添加注解@Aspect

```
//增强User
@Component
@Aspect //生成代理对象
public class UserProxy {
```

7在spring配置文件中开启生成代理对象（xml配置）

```
xmlns:aop="http://www.springframework.org/schema/aop"
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd

<!--      开启Aspect生成代理对象-->
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

配置不同类型通知：在增强类里面，在作为通知方法上面添加通知类型注解，使用切入点表达式配置

```
//增强User
@Component
@Aspect //生成代理对象
public class UserProxy {
```

```

//前置通知
//@Before注解表示作为前置通知
@Before(value = "execution(* com.atguigu.Aop2.User.add(..))")//两个点表示参数
public void before()
{
    System.out.println("before-----");
}

//后置通知
@After(value = "execution(* com.atguigu.Aop2.User.add(..))")
public void after(){
    System.out.println("after-----");
}

//异常通知
@AfterThrowing(value = "execution(* com.atguigu.Aop2.User.add(..))")
public void afterThrowing (){
    System.out.println("afterThrowing-----");
}

//最终通知，注意，最终通知和后置通知的区别：最终通知，不管异常与否，都执行；而后置通知在异常
//时不执行。
@AfterReturning(value = "execution(* com.atguigu.Aop2.User.add(..))")
public void afterReturning (){
    System.out.println("afterReturning---");
}

//环绕通知，方法之前之后都会通知
@Around(value = "execution(* com.atguigu.Aop2.User.add(..))")
public void around (ProceedingJoinPoint proceedingJoinPoint) throws
Throwable //进行加入点
{
    System.out.println("环绕前");

    proceedingJoinPoint.proceed();

    System.out.println("环绕后");
}
}

```

```

环绕前
before-----
add-----
环绕后
after-----
afterReturning---

```

```

public void add()
{
    System.out.println("add-----");
    int a=10/0;
}

```

```

环绕前
before-----
add-----
after-----

```

```
afterThrowing-----
```

上面注解中的value值一样，进行化简

```
//相同切入点抽取
@Pointcut(value = "execution(* com.atguigu.Aop2.User.add(..))")
public void pointDemo() {

}

//前置通知
//@Before注解表示作为前置通知
@Before(value = "pointDemo()")//两个点表示参数
public void before()
{
    System.out.println("before-----");
}

//通知中参数value的值写成方法名称
```

多个增强类对同一个方法进行增强

在增强类上面添加注解@Order（数字类型值）数字类型值越小优先级越高 N 0 1 2 3...

```
@Component
@Aspect
@Order(value = 0)
public class UserProxyPro {
    @Before(value = "execution(* com.atguigu.Aop2.User.add(..))")
    public void before()
    {
        System.out.println("UserProxyPro.before");
    }
}

UserProxyPro.before
环绕前
before-----
add-----
环绕后
after-----
afterReturning----
```

基于xml配置文件实现

创建增强类和被增强类，创建方法

```
public class Book {
    public void add()
    {
        System.out.println("book.add()");
    }
}
```

```
public class BookProxy {
    public void before()
    {
        System.out.println("beforebook.add");
    }
}
```

在spring配置文件中创建两个类对象

```
<!-- 创建对象-->
<bean id="book" class="com.atguigu.Aop3.Book"></bean>
<bean id="bookProxy" class="com.atguigu.Aop3.BookProxy"></bean>
```

在spring配置文件中配置切入点

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

<!-- 配置aop增强-->
<aop:config>
<!-- 切入点-->
<aop:pointcut id="p" expression="execution(*
com.atguigu.Aop3.Book.add(..))"/>
<!-- 配置切面,也就是动作-->
<aop:aspect ref="bookProxy">
<!-- 增强作用在具体的方法上-->
<aop:before method="before" pointcut-ref="p"></aop:before>
</aop:aspect>
</aop:config>
```

Bean配置有其它xml

等同于完全注解开发的@Import({?.class,...})

完全注解开发

```
//完全注解配置
@Configuration
//组件扫描
@ComponentScan(basePackages = "com.atguigu.Aop3")
//aspect生成代理对象
@EnableAspectJAutoProxy(proxyTargetClass = true) //默认是false
public class Scan {
}
```

@Bean可以生成Bean对象，有属性name没有name就默认返回值为对象的方法名为Bean名字，属性名为Bean名字。

JdbcTemplate

是什么？

数据库控制器模板

连接错误解决

https://blog.csdn.net/gg_21479345/article/details/89424743?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522165743469816782246456224%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request_id=165743469816782246456224&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-6-89424743-null-null.142^v32^experiment_2_v1,185^v2^control&utm_term=com.alibaba.druid.pool.DruidDataSource%20error&spm=1018.2226.3001.4187

```
url=jdbc:mysql://localhost:3306/book?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
jdbc.username=root
password=mdjfbzyj515
driverClassName=com.mysql.jdbc.Driver
initialSize=10
maxActive=10
```

```
12 <!-- 利用key-value机制-->
13 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
14 <property name="driverClassName" value="${driverClassName}"></property>
15 <property name="url" value="${url}"></property>
16 <property name="username" value="${jdbc.username}"></property>
17 <property name="password" value="${password}"></property>
18 <property name="initialSize" value="${initialSize}"></property>
19 <property name="maxActive" value="${maxActive}"></property>
20 </bean>
21 <!-- JdbcTemplate对象-->
```

Spring框架对JDBC进行封装，使用jdbcTemplate方便实现对数据库操作

准备工作

1引入相关jar包

druid, mysql, spring中jdbc以及tx（事务）的jar包

 druid-1.1.9.jar	2018/7/1 14:34	Executable Jar File	2,653 KB
 mysql-connector-java-5.1.7-bin.jar	2018/12/24 21:35	Executable Jar File	694 KB

整合template和mybatis需要orm的jar包

2在spring配置文件配置数据库连接池

```
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">

    <!-- 引入外部属性文件-->
    <context:property-placeholder location="classpath:druid.properties">
</context:property-placeholder>
<!-- 利用key-value机制-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${driverClassName}"></property>
        <property name="url" value="${url}"></property>
        <property name="username" value="${username}"></property>
        <property name="password" value="${password}"></property>
        <property name="initialSize" value="${initialSize}"></property>
        <property name="maxActive" value="${maxActive}"></property>
    </bean>
```

druid.properties文件

```
url=jdbc:mysql://localhost:3306/library?useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai
username=root
password=mdjfbzyj515
driverClassName=com.mysql.jdbc.Driver
initialSize=10
maxActive=10
```

3配置JdbcTemplate对象，注入DataSource

```
<!-- JdbcTemplate对象-->
<bean id="JdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <!-- 注入dataSource-->
    <property name="dataSource" ref="dataSource0"></property>
</bean>
```

配置文件

```
<!-- 组件扫描-->
<context:component-scan base-package="com.atguigu.Aop4"></context:component-scan>
```

创建service类，创建dao类，在dao注入jdbcTemplate对象

```
@Repository
public class BookDaoImpl implements BookDao{

    //注入JdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;
}

@org.springframework.stereotype.Service
```



```
public class Service {
    //注入dao
    @Autowired
    private BookDao bookDao;
}
```

对数据库进行增删改查

```
@Repository
public class BookDaoImpl implements BookDao{

    //注入JdbcTemplate
    @Autowired
    private JdbcTemplate jdbcTemplate;

    //增删改操作是类似的
    @Override
    public void add(Book book) {
        String sql="insert into book values(?)";
        //第二个参数是class对象，也就是返回的class对象数组
        int update = jdbcTemplate.update(sql, book.getBookName());
        System.out.println("操作成功:"+update);
    }
}
```

```
@Nullable
public <T> T queryForObject(String sql, Class<T> requiredType) throws DataAccessException {
    return this.queryForObject(sql, this.getSingleColumnRowMapper(requiredType));
}
```

```
//查询返回基本数据类型
@Override
public int findCount() {
    String sql="select count(*) from book";
    //查询返回对象 第二个参数class对象
    Integer integer = jdbcTemplate.queryForObject(sql, Integer.class);
    return integer;
}

//查询返回引用数据类型
@Override
public Book findOne() {
    String sql="select * from book where bookname=?";
    Book book = jdbcTemplate.queryForObject(sql, new BeanPropertyRowMapper<Book>
(Book.class), "庄涓涓");
    return book;
}
```

```
m queryForObject(String sql, RowMapper<T> rowMapper, Object... args) T
```

第一个参数是sql语句

第二个参数是RowMapper，是接口，返回不同数据类型，使用这个接口里面实现类完成数据封装，用BeanPropertyRowMapper类new对象作为第二个参数

第三个参数，sql语句值

注意是query!!!! 不是queryForObject

```
//查询返回集合
@Override
public List<Book> findBookList() {
    String sql="select * from book";
    List<Book> query = jdbcTemplate.query(sql, new BeanPropertyRowMapper<Book>
(Book.class));
    return query;
}
```

批量操作

```
public int[] batchUpdate(String sql, List<Object[]> batchArgs) throws DataAccessException {
    return this.batchUpdate(sql, batchArgs, new int[0]);
}
```

第二个参数：List集合，批量添加多条记录数据

batch批量

```
@Override
public void updates(List<Object[]> batch) {
    //数组中每一个元素是来填充占位符
    String sql="insert into book values(?)";
    int[] ints = jdbcTemplate.batchUpdate(sql, batch);//遍历循环sql
    System.out.println(Arrays.toString(ints));
}
```

批量添加和批量修改以及批量删除是一样的，因为会遍历循环sql语句

事物操作

什么是事物？

数据库操作最基本单元，逻辑上一组操作，要么都成功，如果有一个失败就都失败

事物四个特征ACID

原子性一致性隔离性持久性

java环境中三个层

web 视图

service 业务操作

dao 数据库操作，不写业务

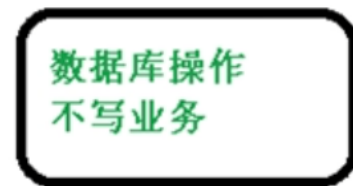
Service



创建转账的方法

(1) 调用dao两个的方法

Dao



创建两个方法

(1) 少钱的方法

(2) 多钱的方法

过程

开启事务

进行业务操作

没有发生异常，提交事务。出现异常，事务回滚

事务添加到JavaEE三层结构里面Service层（业务逻辑层）

事务管理操作两种方式：编程式事务管理和声明式事务管理（使用）

用声明式事务管理

在Spring进行声明式事务管理，底层使用AOP原理，提供一个接口PlatformTransactionManager，代表事务管理器，这个接口针对不同的框架提供不同的实现类！！

针对数据库是用DataSourceTransactionManager数据库事务管理

基于注解方式（使用）

先要有数据库连接池

```
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">
<!--引入外部属性文件要用到标签context-->
<context:property-placeholder location="classpath:druid.properties">
</context:property-placeholder>
<!-- 利用key-value机制-->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
<property name="url" value="${url}"></property>
<property name="driverClassName" value="${driverClassName}"></property>
<property name="username" value="${jdbc.username}"></property>
<property name="password" value="${password}"></property>
<property name="initialSize" value="${initialSize}"></property>
<property name="maxActive" value="${maxActive}"></property>
</bean>
```

在Spring配置文件，配置事务管理器

```

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

```

在spring配置文件开启，开启事务注解

首先要在spring配置文件引入名称空间tx

```

xmlns:tx="http://www.springframework.org/schema/tx"
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd

```

开启事物注解

```

<!-- 开启事务注解-->
<!-- 要指定事务管理器-->
    <tx:annotation-driven transaction-manager="transactionManager">
</tx:annotation-driven>

```

在service类上面（获取service类里面方法上面）添加事物注解

@Transactional注解可以添加到类或者方法上面，范围不同

完全注解声明式事务管理

```

@Configuration
@ComponentScan(basePackages = "com.atguigu.Aop6")//开启组件扫描
@EnableTransactionManagement //开启事务
public class configure {
    //创建数据库连接池
    @Bean
    public DruidDataSource getDruidDataSource()
    {
        DruidDataSource dataSource=new DruidDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver"); //驱动
        dataSource.setUrl("jdbc:mysql://localhost:3306/book?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=Asia/Shanghai"); //路径
        dataSource.setUsername("root");
        dataSource.setPassword("mdjfbzyj515");
        return dataSource;
    }

    //创建JdbcTemplate对象
    @Bean
    public JdbcTemplate getJdbcTemplate(DataSource dataSource)
    {
        //到IOC容器中根究类型找到dataSource
        JdbcTemplate jdbcTemplat=new JdbcTemplate();
        //注入dataBase
        jdbcTemplat.setDataSource(dataSource);
        return jdbcTemplat;
    }

    //创建事务管理器

```

```

@Bean
public DataSourceTransactionManager
getDataSourceTransactionManager(DataSource dataSource)
{
    DataSourceTransactionManager dataSourceTransactionManager=new
DataSourceTransactionManager();
    dataSourceTransactionManager.setDataSource(dataSource);
    return dataSourceTransactionManager;
}
}

```

详谈事务@Transational

排他锁(Exclusive Locks, 简称 X 锁)

共享锁(Share Locks,简称s锁)

在Read Uncommitted级别下, 读操作不加S锁;

在Read Committed级别下, 读操作需要加S锁, 但是在语句执行完以后释放S锁;

在Repeatable Read级别下, 读操作需要加S锁, 但是在事务提交之前并不释放S锁, 也就是必须等待事务执行完毕以后才释放S锁。

在Serialize级别下, 会在Repeatable Read级别的基础上, 添加一个范围锁。保证一个事务内的两次查询结果完全一样, 而不会出现第一次查询结果是第二次查询结果的子集。

在使用可重复读隔离级别时, 一个事务中所有被读取过的行上都会被加上S锁, 直到该事务被提交或回滚, 行上的锁才会被释放。这样可以保证在一个事务中即使多次读取同一行, 得到的值也不会改变。

关于锁的问题

针对不同的锁, 有了不同的场景, 因此有了不同的隔离级别

https://blog.csdn.net/feichitianxia/article/details/112506680?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522166005930316781667844895%2522%252C%2522scm%2522%253A%25220140713.130102334.%2522%257D&request_id=166005930316781667844895&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-112506680-null-null.142^v40^new_blog_pos_by_title,185^v2^control&utm_term=%E9%9A%94%E7%A6%BB%E7%BA%A7%E5%88%AB%E5%92%8C%E9%94%81%E7%9A%84%E5%85%B3%E7%B3%BB&spm=1018.2226.3001.4187

事务就是一系列操作不行就回滚当没发生。

@Transational失效有几种情况: 数据库不支持事务操作, 方法不是public (基于aop实现事务, 所以要public) 用final和static修饰也不行, 手动抛出异常, 未被spring管理, 多线程调用, 方法内部调用

@Component业务特殊组件层

@Service用于服务层, 处理业务逻辑

@Controller用于呈现层

@Repository用于持久层, 数据库访问层

四个实现功能一样, 用于不同开发程序层次

手续: 先创建事务管理, 然后开启事务管理, 最后才能用@Transational

如果注解添加到类所有方法都添加事务, 也可以只添加方法

声明事物管理参数配置

propagation

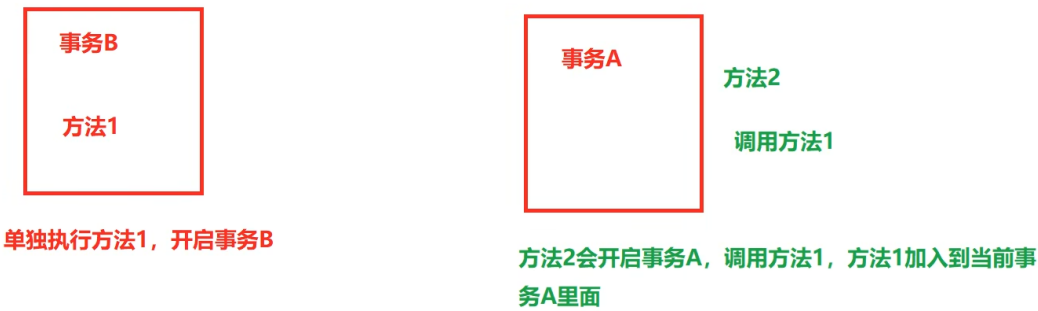
propagation传播 事物传播行为

当一个事务方法被另一个事务方法调用时候，这个事务方法如何进行

7种传播行为

required必备的

REQUIRED	如果有事务在运行，当前的方法就在这个事务内运行，否则，就启动一个新的事务，并在自己的事务内运行
----------	---



required_NEW

REQUIRED_NEW	当前的方法必须启动新事务，并在它自己的事务内运行。如果有事务正在运行，应该将它挂起
--------------	---



supports

....

isolation

事务隔离级别

事务有隔离性，多事务操作之间不会产生影响，不考虑隔离性会产生很多问题

有三个问题脏读，不可重复读，虚（幻）读

脏读：一个未提交事务读取到另一个未提交事务的数据，（另一个只会改了而未提交事务的数据没改）

不可重复读：一个未提交事务读取到另一个提交事务修改数据。未提交事物读取的数据数据两次不一样

虚读：一个未提交事务读取到另一提交事务添加数据（未提交事务两次读取到的数据数量不一致）

通过设置事务隔离级别解决以上三个问题

mysql默认可重复读

	脏读	不可重复读	幻读
READ UNCOMMITTED (读未提交)	有	有	有
READ COMMITTED (读已提交)	无	有	有
REPEATABLE READ (可重复读)	无	无	有
SERIALIZABLE (串行化)	无	无	无

```
@Transactional(isolation = Isolation.READ_COMMITTED)
public void accountMoney()
```

timeout

超时时间

事务需要在一定时间进行提交，如果不提交进行回滚，默认-1，设置时间是以秒为单位

readOnly

读：查询操作

写：添加修改删除操作

默认值为false，表示增删改查都可以

设置readOnly为true只能是查询

rollbackFor

设置出现哪些异常进行回滚

noRollbackFor

设置出现哪些异常进行回滚

基于xml方式

在spring配置文件中配置：

1配置事务管理器

配置通知

配置切入点和切面

```
<!-- 配置通知 -->
<tx:advice id="txadvice">
<!-- 配置事务参数 -->
<tx:attributes>
<!-- 指定哪种规则的方法上面添加事务，*表示以account开头命名方法都可以 -->
<tx:method name="account*" />
```

```

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

<!-- 配置切入点和切面-->
<aop:config>
<!-- 配置切入点-->
    <aop:pointcut id="pt" expression="execution(*
com.atguigu.Aop6.PeopleService.*(..))"/>
<!-- 配置切面，注意不是aspect! ! ! -->
    <aop:advisor advice-ref="txadvice" pointcut-ref="pt"></aop:advisor>
</aop:config>

```

Spring框架新功能

整个Spring框架的代码基于Java8，运行时兼容JDK9，许多不建议的类和方法在代码库删除

核心特性，自带了通用的日志封装，移除Log4jConfigListener，但是Spring5框架整合Log4j2，官方建议使用Log4j2。

要引入jar包

此电脑 > Data (D:) > Spring > spring5所需资料 > 资料 > 资料 > 4 log4j

名称	修改日期	类型	大小
log4j2.xml	2020/5/18 22:30	XML 文档	2 KB
log4j-api-2.11.2.jar	2019/2/19 13:39	Executable Jar File	261 KB
log4j-core-2.11.2.jar	2020/5/18 22:23	Executable Jar File	1,592 KB
log4j-slf4j-impl-2.11.2.jar	2020/5/18 22:36	Executable Jar File	23 KB
slf4j-api-1.7.30.jar	2020/2/13 15:51	Executable Jar File	41 KB

创建log4j2.xml配置文件（名字是固定的！）

```

<?xml version="1.0" encoding="UTF-8"?>
<!--日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL
-->
<!--Configuration后面的status用于设置log4j2自身内部的信息输出，可以不设置，当设置成trace
时，可以看到log4j2内部各种详细输出-->
<configuration status="INFO">
    <!--先定义所有的appender-->
    <appenders>
        <!--输出日志信息到控制台-->
        <console name="Console" target="SYSTEM_OUT">
            <!--控制日志输出的格式-->
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level
%logger{36} - %msg%n"/>
        </console>
    </appenders>
    <!--然后定义logger，只有定义了logger并引入的appender，appender才会生效-->
    <!--root: 用于指定项目的根日志，如果没有单独指定Logger，则会使用root作为默认的日志输出-->
    <loggers>
        <root level="info">
            <appender-ref ref="Console"/>
        </root>
    </loggers>
</configuration>

```


支持@Nullable注解，可以使用在方法，属性，参数，表示方法返回，属性值，参数值为空

(2) 注解用在方法上面，方法返回值可以为空

```
@Nullable  
String getId();
```

(3) 注解使用在方法参数里面，方法参数可以为空

```
public <T> void registerBean(@Nullable String beanName,  
    this.reader.registerBean(beanClass, beanName, suppli  
}
```

(4) 注解使用在属性上面，属性值可以为空

```
@Nullable  
private String bookName;
```

支持函数式风格

如果自己new对象spring中的IOC是不存在该对象所以要在IOC进行注册

```
//函数式风格创建对象，交给Spring进行管理  
//创建GenericApplicationContext对象  
GenericApplicationContext context=new GenericApplicationContext();  
context.refresh(); //刷新一下  
// 调用context的方法对象注册  
// context.registerBean(User.class,()->new User());  
// 获取在spring注册的对象  
// User bean = (User) context.getBean("com.atguigu.Aop7.User");  
context.registerBean("userName",User.class,()->new User());  
User userName = (User) context.getBean("userName");  
System.out.println(userName);
```

Spring5支持整合JUnit5

整合JUnit4

第一步引入Spring相关针对测试依赖

此电脑 > Data (D:) > Spring > spring-framework-5.2.6.RELEASE > libs						在 libs
2022-06	名称	修改日期	类型	大小		
Spring	spring-test-5.2.6.RELEASE.jar	2020/4/28 8:15	Executable Jar File	669 KB		

第二步创建测试类

```
import org.junit.Test;
```

```
@RunWith(SpringJUnit4ClassRunner.class) //单元测试框架  
@ContextConfiguration("classpath:Aop8.xml") //加载配置文件  
public class Juit4test {
```

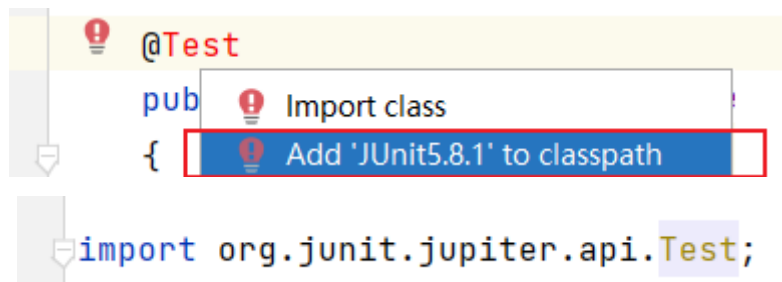
```

@Autowired
private UserService userService;
@Test
public void test()
{
    //不需要
    // ApplicationContext context=new ClassPathXmlApplicationContext();
    // UserService userService = context.getBean("userService",
userService.class);
    System.out.println(userService.getUserDaoImp());
}
}

```

整合JUnit5

引入unit5jar包



```

@ExtendWith(SpringExtension.class)
@ContextConfiguration("classpath:Aop8.xml")
public class Junit5test {
    @Autowired
    private UserService userService;
    @Test
    public void test()
    {
        System.out.println(userService.getUserDaoImp());
    }
}

```

复合注解

```
//@ExtendWith(SpringExtension.class)单元测试框架
//@ContextConfiguration("classpath:Aop8.xml")加载配置文件
//复合注解
@SpringJUnitConfig(locations = "classpath:Aop8.xml")
public class Junit5test {
    @Autowired
    private UserService userService;
    @Test
    public void test()
    {
        System.out.println(userService.getUserDaoImp());
    }
}
```