

Spring5 框架

Spring5 框架

第一章 Spring框架概述

1.1 基本说明

1.2 特点

1.3 入门案例 - 使用 Spring 创建对象

第二章 IOC 容器

2.1 IOC 概念和原理

2.1.1 什么是IOC

2.1.2 IOC底层原理

2.2 IOC 容器的实现方式

2.3 IOC 的操作 - Bean 管理

2.3.1 什么是 Bean 管理

2.3.2 Bean 管理操作的实现有两种方式

2.3 IOC 操作 Bean 管理 (基于xml)

2.3.1 基于 xml 方式创建对象

2.3.2 基于 xml 方式注入属性

DI

使用 xml 的方式 通过 set 方法注入属性

使用 xml 的方式 通过 有参构造函数 注入属性

p 名称空间注入(了解, 是对2.3.3的简化操作)

注入空值和特殊符号

注入外部bean

注入内部 bean

级联赋值

注入四种集合(数组, List, Set, Map)类型属性 - 基本使用

注入四种集合(数组, List, Set, Map)类型属性 - 进阶使用

2.3.3 FactoryBean - 工厂 bean

2.3.4 bean 的作用域

2.3.5 (重点) bean 的生命周期

基本五步

bean 后置处理器

2.3.6 自动装配

2.3.7 外部属性文件

2.4 IOC 操作 Bean 管理(基于注解)

2.4.1 Spring 针对 Bean 管理创建对象提供的注解

入门案例 - 基于注解方式完成对象创建

2.4.2 注解扫描

2.4.3 使用注解完成属性注入

使用的注解

@AutoWired和@Qualifier的使用

@Resource和@Value的使用

2.4.4 完全注解开发

第三章 AOP

3.1 概念

3.2 底层原理

3.2.1 实现 JDK 动态代理

3.3 操作术语

3.4 AOP操作 - 准备工作

3.5 AOP操作 - AspectJ注解

基本使用

细节补充

完全注解Aop

3.6 AOP操作 - AspectJ配置文件

第四章 JdbcTemplate

4.1 准备工作

4.2 添加修改删除

4.3 查询

4.3.1 查询某个特殊值

4.3.2 查询一个对象

4.3.3 查询多个对应构成的集合

4.4 批量操作

第五章 事务

5.1 复习

5.2 搭建 Spring 操作事务环境

5.3 Spring 事务管理

5.3.1 介绍

5.3.2 声明式事务管理(基于 XML)

基本使用

5.3.3 声明式事务管理(基于注解) - 推荐

基本使用

参数配置

完成注解开发

第六章 Spring5 新功能

第一章 Spring框架概述

1.1 基本说明

- Spring 是轻量级的开源的 JavaEE 框架
- 其目的主要是解决企业应用开发的复杂性
- Spring 有两个核心部分，IOC 和 Aop
 - a. IOC：控制反转，把创建对象的过程交给 Spring 进行管理
 - b. Aop：面向切面，不修改源代码进行功能增加

1.2 特点

1. 方便解耦，简化开发
2. Aop 编程支持
3. 方便程序测试
4. 方便和其他框架进行整合
5. 方便进行实物操作
6. 降低 API 开发难度

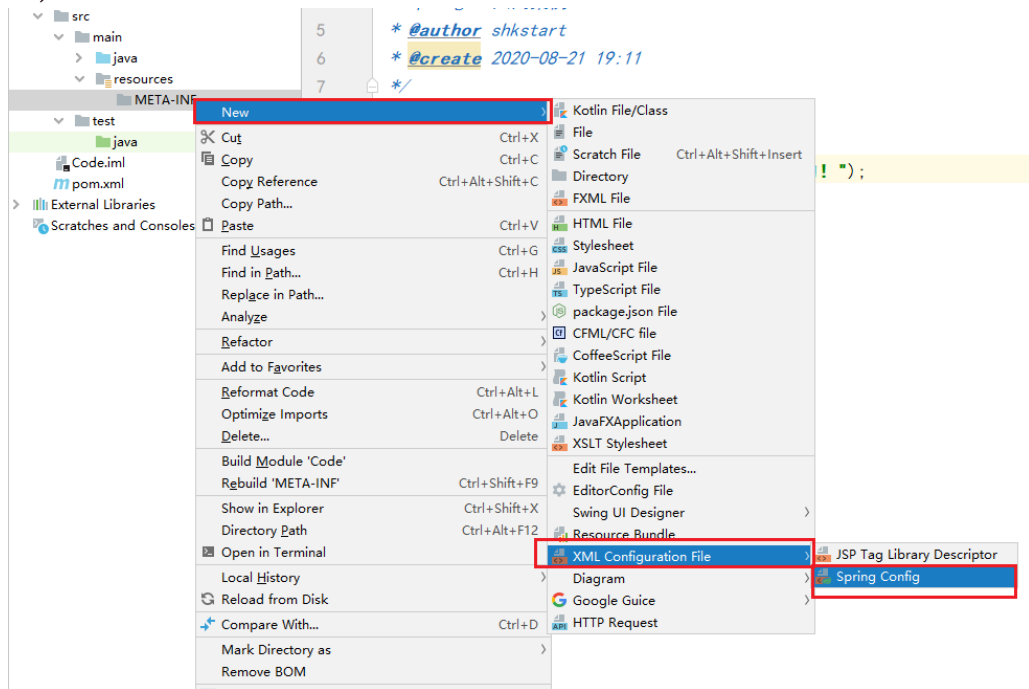
1.3 入门案例 - 使用 Spring 创建对象

1. 将以下代码添加到 Maven 工程的 pom.xml 文件内

```
<properties>
    <spring.version>5.2.6.RELEASE</spring.version>
</properties>
<dependencies>
    <!--spring context模块-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
</dependencies>
```

2. 编写类和对应的方法

3. 在 main/resources/ 目录下创建 xml 格式的 spring 配置文件(name自定义)



4. 配置 类的对象创建

```
<!--
这里 - 配置Main类对象的创建
id: 别名
class: 全类名
-->
<bean id="main" class="pers.dreamer07.code.Main">
</bean>
```

5. 在 pom.xml 下引入 junit 单元测试的依赖

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

6. 编写测试类和测试方法

```
@Test
public void test1(){
  /*
   * 1. 加载 spring 配合文件
```

```

    *      ClassPathXmlApplicationContext - 参考编译
    后 classes文件夹 的路径
    *      FileSystemXmlApplicationContext - 参考 当
    前工程/模块 路径
    * */
    //ApplicationContext context = new
    ClassPathXmlApplicationContext("META-
    INF/bean1.xml");
    ApplicationContext context = new
    FileSystemXmlApplicationContext("src/main/resources/
    META-INF/bean1.xml");

    //2. 读取配置文件中创建的对象
    Main main = context.getBean("main", Main.class);

    //3. 调用方法
    main.getEMT();
}

```

第二章 IOC 容器

2.1 IOC 概念和原理

2.1.1 什么是IOC

1. 控制反转，把 对象创建 和 对象调用 的过程统一交给 Spring 来进行管理
2. 使用 IOC 的目的就是为了 降低耦合度
3. 入门案例就是 IOC 实现

2.1.2 IOC底层原理

- 主要使用的技术：xml解析，工厂模式，反射

- 图解 IOC 创建对象的过程

第一步 xml配置文件，配置创建的对象

```
<bean id="dao" class="com.atguigu.UserDao"></bean>
```

进一步降低耦合度

第二步 有service类和dao类，创建工厂类

I

```
class UserFactory {
    public static UserDao getDao() {
        String classValue = class属性值; //1 xml解析
        Class clazz = Class.forName(classValue); //2 通过反射创建对象
        return (UserDao)clazz.newInstance();
    }
}
```

2.2 IOC 容器的实现方式

- IOC 基于 IOC 容器完成，而 IOC 容器的底层就是对象工厂
- Spring 提供了 IOC 容器的两种实现方式(对应两个接口)

a. BeanFactory

- 开发不常用，IOC 容器的基本实现，是 Spring 内部的使用接口，
- 加载配置文件时不会创建对象，在是在使用 对应/获取的对象时再创建对象

b. ApplicationContext

- 开发常用，是 BeanFactory 的子接口，提供更多更强大的功能，一般由开发人员进行使用
- 加载配置文件时，会把文件中配置的对象进行创建
- 两个实现类

ClassPathXmlApplicationContext /
FileSystemXmlApplicationContext

2.3 IOC 的操作 - Bean 管理

2.3.1 什么是 Bean 管理

1. Spring 创建对象
2. Spring 向创建的对象中注入属性

2.3.2 Bean 管理操作的实现有两种方式

1. 基于xml
2. 基于依赖

2.3 IOC 操作 Bean 管理 (基于xml)

2.3.1 基于 xml 方式创建对象

1. 在 spring 的配置文件中 使用 `<bean>` 标签及其属性配置创建对象的信息
2. bean 标签的常用属性
 - id属性: 唯一标识
 - class属性: 创建对象对应类的全类名
3. 创建对象时, 默认执行无参构造方法

2.3.2 基于 xml 方式注入属性

DI

依赖注入, 就是注入属性; 是IOC中的一种具体实现; 但这种注入属性需要在创建对象的基础上完成

- Spring 中支持通过两种方式注入属性
 - a. 通过set方法注入属性
 - b. 通过对应的有参构造器注入属性

使用 xml 的方式 通过 set 方法注入属性

1. 创建类, 属性以及对应的 set() 方法
2. 在配置文件 xml 中使用 bean 标签配置要创建对象的类信息
3. 在 bean 标签中, 使用 property 标签实现注入属性(该标签会调用对应的 set方法)
 - 设置name属性: 指定要设置的对象属性
 - 设置value属性: 指定要设置的对象属性的属性值

4.

```
<!--举例-->
<bean name="book" class="pers.dreamer07.code.Book">
    <property name="name" value="喝茶部的养成">
</property>
    <property name="author" value="HTT"></property>
</bean>
```

5. 编写测试类和测试方法

使用 **xml** 的方式 通过 有参构造函数 注入属性

1. 创建类，属性以及对应的 `set()` 方法
2. 在配置文件 **xml** 中使用 **bean** 标签配置要创建对象的类信息
3. 在 **bean** 标签中，使用 **constructor-arg** 标签实现注入属性(该标签会自动填充为对应构造函数的参数值)
 - 设置 **name** / **index** 属性：根据构造函数的 形参名/形参的索引 (从0开始)
 - 设置 **value** 属性：对应的参数值

4. `<!--举例`
构造函数原型: `public Book(String name, String author, Double price)`
`-->`

```
<bean id="book" class="pers.dreamer07.code.Book">
    <constructor-arg name="name" value="喝茶部的养成">
</constructor-arg>
    <constructor-arg name="author" value="HTT">
</constructor-arg>
    <constructor-arg index="2" value="99.9">
</constructor-arg>
</bean>
```

5. 编写测试方法和测试类

p 名称空间注入(了解，是对2.3.3的简化操作)

1. 在配置文件中的 **beans** 标签中添加以下属性

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

2. 在 **bean** 标签内使用 **p** 名称空间注入属性(调用set方法)

```
<bean id="book" class="pers.dreamer07.code.Book"
p:name="喝茶部的养成" p:author="HTT" p:price="99.9">
</bean>
```


注入空值和特殊符号

- 注入空值，使用 `<null/>` 标签

```
<!--set方法-->
<property name="属性名">
    <null/> <!--注入空值-->
</property>
<!--构造方法-->
<constructor-arg name="属性名">
    <null/>
</constructor-arg>
```

- 注入特殊符号，使用 `<value>` 标签

```
<!--使用CDATA-->
<property name="属性名">
    <value>
        <![CDATA[属性值]]>
    </value>
</property>
```

注入外部bean

1. 设计两个类 `service` 类和 `dao` 类
2. 在 `service` 中调用 `dao` 里的方法
3. 在 `spring` 配置文件中进行以下配置

```
<bean id="userService"
class="pers.dreamer07.spring5.service.UserService">
    <!-- 使用ref属性指定外部bean的id -->
    <property name="userDAO" ref="userDAO">
</property>
</bean>
<!--外部bean-->
<bean id="userDAO"
class="pers.dreamer07.spring5.dao.UserDAO"></bean>
```

4. 设计测试类和方法

注入内部 bean

1. 设计具有一对多关系的两个类 **emp**(员工类) 和 **dept** (部门类)
2. 都重写 **toString** 方法
3. 在 **spring** 配置文件中进行以下配置

```
<bean id="emp"
class="pers.dreamer07.spring5.bean.Emp">
  <property name="name" value="Jack"></property>
  <property name="gender" value="男"></property>
  <property name="dept">
    <!--在内部配置对应的bean对象-->
    <bean id="dept"
class="pers.dreamer07.spring5.bean.Dept">
      <property name="dname" value="IT">
</property>
      </bean>
    </property>
  </bean>
```

4. 设计测试类和方法

- 根据类与类的关系不同，使用不同的方式注入**bean**能够体现它们之间的关系

级联赋值

当两个 **bean** 关联时，通过一个 **bean** 给另一个 **bean** 里的属性赋值

1. 设计具有一对多关系的两个类 **emp**(员工类) 和 **dept** (部门类)
2. 都重写 **toString** 方法
3. 可以通过两种方式完成
 - 3.1 注入外部**bean**的同时，通过外部**bean**完成对其自身的属性赋值

```

<bean id="emp"
class="pers.dreamer07.spring5.bean.Emp">
    <property name="name" value="Jack"></property>
    <property name="gender" value="男"></property>
    <property name="dept" ref="dept"></property>
</bean>
<bean id="dept"
class="pers.dreamer07.spring5.bean.Dept">
    <!--方式1: 注入外部bean的同时, 通过外部bean完成对其的属性
赋值-->
    <property name="dname" value="级联赋值1">
</property>
</bean>

```

3.2 注入外部bean时, 通过内部调用其属性完成赋值

注意: 内部级联赋值需要对应的bean属性具有get方法

```

<bean id="emp"
class="pers.dreamer07.spring5.bean.Emp">
    <property name="name" value="Jack"></property>
    <property name="gender" value="男"></property>
    <property name="dept" ref="dept"></property>
    <!--方式2: 注入外部bean时, 通过内部调用其属性完成赋值
        注意 - 内部级联赋值需要对应的bean属性具有get方法
    -->
    <property name="dept.dname" value="OHHH">
</property>
</bean>
<bean id="dept"
class="pers.dreamer07.spring5.bean.Dept"></bean>

```

注入四种集合(数组, List, Set, Map)类型属性 - 基本使用

1. 设计类以及对应的属性和set方法
2. 重写 toString 方法
3. 在 xml 中配置对应的属性

```

<!--配置需要创建的对象-->
<bean id="stu"
class="pers.dreamer07.spring5.coll_map.Stu">
    <!--数组属性-->
    <property name="arr">

```

```

<!--<list>标签和<array>标签都可以-->
<array>
    <value>砸我鲁多</value>
    <value>邪王真眼!! </value>
</array>
</property>
<!--list集合属性-->
<property name="list">
    <!--<list>标签和<array>标签都可以-->
    <list>
        <value>AAA</value>
        <value>BBB</value>
        <value>BBB</value>
    </list>
</property>
<!--set集合属性-->
<property name="set">
    <set>
        <value>CCC</value>
        <value>CCC</value>
    </set>
</property>
<!--map集合属性-->
<property name="map">
    <map>
        <!--使用entry标签, 设置key和value属性-->
        <entry key="E" value="EMT!"></entry>
        <entry key="M" value="EMT!!"></entry>
        <entry key="T" value="EMT!!!"></entry>
    </map>
</property>
</bean>

```

4. 设计测试类和方法

注入四种集合(数组, **List**, **Set**, **Map**)类型属性 - 进阶使用

- 集合元素类型为对象类型

```

<!--配置需要创建的对象-->
<bean id="stu"
class="pers.dreamer07.spring5.coll_map.Stu">
    <!--集合元素类型为对象类型-->

```

```
<property name="courses">
  <list>
    <!--使用ref注入外部bean-->
    <ref bean="course1"></ref>
    <ref bean="course2"></ref>
    <!--或使用内部bean注入-->
    <bean id="course3"
class="pers.dreamer07.spring5.coll_map.Course">
      <property name="name"
value="SpringMVC"></property>
    </bean>
  </list>
</property>
</bean>
<!--创建对应的外部bean-->
<bean id="course1"
class="pers.dreamer07.spring5.coll_map.Course">
  <property name="name" value="Spring5框架">
</property>
</bean>
<bean id="course2"
class="pers.dreamer07.spring5.coll_map.Course">
  <property name="name" value="MyBatis框架">
</property>
</bean>
```

- 把集合注入部分提取出来
 - a. 在 Spring 配置文件中引入名称空间 util

```
<?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:util="http://www.springframework.org/sc
hema/util"

xsi:schemaLocation="http://www.springframewor
k.org/schema/beans
http://www.springframework.org/schema/beans/s
pring-beans.xsd

http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/sp
ring-util.xsd ">
```

b. 使用 util 标签完成对 list 集合注入的提取

```
<!--将list集合注入提取出来-->
<util:list id="bookList">
    <value>SpringBoot</value>
    <value>SpringCloud</value>
    <value>Shiro</value>
</util:list>
<!--将外部list集合注入到属性中-->
<bean id="book"
class="pers.dreamer07.spring5.coll_map.Book">
    <!--使用ref引用-->
    <property name="names" ref="bookList">
</property>
</bean>
```

c. 设计测试类和方法

2.3.3 FactoryBean - 工厂 bean

Spring 中有两种类型 bean，一种是普通 bean，另外一个工厂 bean(FactoryBean)

- 普通 bean：配置文件中定义的 bean 类型(由 class 属性决定)和使用时返回值的类型相同
- 工厂 bean：配置文件中定义的 bean 类型可以和使用时返回值的类型不相同
- 实现步骤
 - a. 设计一个类，将该类作为工厂 bean，实现接口 `FactoryBean<T>`
 - 由 T 决定返回的数据类型
 - b. 实现接口里的方法，在实现的方法中定义返回的 bean 类型

```
@Override
//主要通过该方法返回对应数据类型的数据
public Person getObject() throws Exception {
    Person person = new Person();
    person.setName("EMT!!");
    return person;
}

@Override
public Class<?> getObjectType() {
    return null;
}
```

- c. 在配置文件中配置实现了接口的类

```
<bean id="myBean"
class="pers.dreamer07.spring.factorybean.MyBean"></bean>
```

- d. 设计测试类和方法，使用工厂 Bean 实现接口的类型 T 接收返回值

```
@Test
public void test(){
    ApplicationContext context = new
    ClassPathXmlApplicationContext("META-
    INF/bean1.xml");
    //配置的是 MyBean 类但可以使用 Person 类的对象
    来接收
    Person person = context.getBean("myBean",
    Person.class);
    System.out.println(person);
}
```

2.3.4 bean 的作用域

在 Spring 中，可以设置创建的 bean 实例是单实例(默认)还是多实例

1. 在 spring 配置文件对应的 bean 标签里，设置 scope 属性
2. scope 的取值
 - 默认值，singleton，代表单例对象
 - prototype，代表多实例对象
3. singleton 和 prototype 的区别
 - a. singleton 是单例对象，prototype 是多例对象
 - b. singleton 会在加载配置文件时创建对应的对象，prototype 会在调用对应的 getBean() 方法创建对应的对象
4. 设计测试类和方法，调用两次 getBean() 看是否为同一个实例(地址)

2.3.5 (重点) bean 的生命周期

通过 spring 创建的 bean 对象从创建到销毁的过程

bean : Spring IOC 容器所管理的对象

基本五步

1. 通过构造器创建 bean 实例(默认调用无参构造器)
2. 为 bean 的属性设置值或对其他 bean 的引用(调用 set 方法)
3. 调用 bean 的初始化方法(需要进行配置)
4. bean 的使用，获取 bean 的实例

5. 关闭容器时，调用 bean 的销毁方法(需要进行配置)

- Java源代码

```
public class Order {

    private String oname;

    //无参构造器
    public Order() {
        System.out.println("第一步 调用对应的构造器");
    }

    //通过set完成依赖注入
    public void setName(String oname) {
        System.out.println("第二步 调用对应属性的set方法");
        this.oname = oname;
    }

    //创建对应的初始化方法
    public void initMethod(){
        System.out.println("第三步 调用 bean 配置的初始化方法");
    }

    //创建对应的销毁方法
    public void destroyMethod(){
        System.out.println("第五步 调用 bean 配置的销毁方法");
    }
}
```

- xml 配置文件

```

<!--
    配置对应的bean实例
    init-method: 配置初始化方法
    destroy-method: 配置销毁方法
-->
<bean id="order"
class="pers.dreamer07.spring.life.Order" init-
method="initMethod" destroy-method="destroyMethod">
    <property name="oname" value="EMT!!"></property>
</bean>

```

- 测试类

```

public class LifeTest {

    @Test
    public void test(){
        //创建SpringIOC容器对象
        ApplicationContext context = new
        ClassPathXmlApplicationContext("META-
        INF/bean3.xml");
        //获取对应的bean实例
        Order order = context.getBean("order",
        Order.class);
        System.out.println("第四步，获取bean实例");
        System.out.println(order);

        /* 调用close()方法关闭对应的SpringIOC容器
        *      - 由于 ApplicationContext 中并没有对应的
        close()方法
        *      所以需要强转子类
        ClassPathXmlApplicationContext
        * */
        ((ClassPathXmlApplicationContext)
        context).close();
    }
}

```

bean 后置处理器

1. 通过构造器创建 bean 实例(默认调用无参构造器)
2. 为 bean 的属性设置值或对其他 bean 的引用(调用 set 方法)

3. 调用对应后置处理器的 `postProcessBeforeInitialization()` 方法
 4. 调用 `bean` 的初始化方法(需要进行配置)
 5. 调用对应后置处理器的 `postProcessAfterInitialization()` 方法
 6. `bean` 的使用, 获取 `bean` 的实例
 7. 关闭容器时, 调用 `bean` 的销毁方法(需要进行配置)
- (实现)

- Java主程序源代码不变
- 测试类不变
- 后置处理器.java

```
//1. 实现BeanPostProcessor接口
public class OrderProc implements BeanPostProcessor
{
    //2. 重写其中的两个方法
    @Override
    public Object
    postProcessBeforeInitialization(Object bean, String
    beanName) throws BeansException {
        System.out.println("会在 bean 配置的初始化方法前
        执行");
        return bean; //返回bean
    }

    @Override
    public Object
    postProcessAfterInitialization(Object bean, String
    beanName) throws BeansException {
        System.out.println("会在 bean 配置的初始化方法后
        执行");
        return bean; //返回bean
    }
}
```

- xml文件, 额外配置以下代码

```

<!--配置 bean 后置处理器
会对当前xml配置文件中的所有bean都起作用
正常引入即可，spring会将实现了BeanPostProcessor的bean
对象作为后置处理器
-->
<bean id="orderProc"
class="pers.dreamer07.spring.life.OrderProc"></bean>

```

2.3.6 自动装配

根据指定装配规则(属性名称/属性类型)，Spring 自动将匹配的属性值进行注入

1. 根据 属性名称 完成自动装配
2. 根据 属性类型 完成自动装配

3.


```

<!--bean 自动装配 - 使用 bean 标签的属性autowire实现
- byName: 根据属性名完成自动装配，要求 对应的bean
实例id 和 属性名相同
- byType: 根据属性的类型完成自动装配，由class决定，且在同一个xml文件中不能配置两个相同class的两个bean
-->
<bean id="emp"
class="pers.dreamer07.spring.autowire.Emp"
autowire="byType"></bean>
<bean id="dept"
class="pers.dreamer07.spring.autowire.Dept"></bean>
<!-- 使用byType时会报错 -->
<!-- <bean id="dept1"
class="pers.dreamer07.spring.autowire.Dept"></bean>
-->

```

2.3.7 外部属性文件

在 Spring 中，我们可以将一些常用的固定值保存在外部文件(比如连接数据库的地址等等)，然后通过指定的方法完成属性注入

(配置Druid数据库连接池)

1. 创建 properties 格式的外部属性文件

```
#只做引入，不做测试
emt.driverClass=com.mysql.cj.jdbc.Driver
emt.url=jdbc:mysql://localhost:3306/test
emt.userName=root
emt.password=root
```

2. 引入 context 容器

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

xmlns:context="http://www.springframework.org/schema
/context"

xsi:schemaLocation="http://www.springframework.org/s
chema/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">
</beans>
```

3. 在 spring 配置文件中使用以下标签引入外部属性文件

```
<!--引入外部属性文件-->
<context:property-placeholder
location="jdbc.properties"/>
<!--使用外部属性文件配置连接池-->
<bean id="dataSource"
class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName"
value="${emt.driverClass}"></property>
    <property name="url" value="${emt.url}">
</property>
    <property name="username"
value="${emt.userName}"></property>
    <property name="password"
value="${emt.password}"></property>
</bean>
```

2.4 IOC 操作 Bean 管理(基于注解)

2.4.1 Spring 针对 Bean 管理创建对象提供的注解

1. @Component(普通)
2. @Service(业务逻辑层)
3. @Controller(web层)
4. @Repository(dao层)

- 上面的四个注解功能相同，都可以用来创建 bean 实例，但建议根据分类来使用，便于开发

入门案例 - 基于注解方式完成对象创建

1. 额外引入 spring-aop 的 jar 包 / 依赖

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.2.6.RELEASE</version>
</dependency>
```

2. 引入 context 容器

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

xmlns:context="http://www.springframework.org/schema
/context"

xsi:schemaLocation="http://www.springframework.org/s
chema/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">
</beans>
```

3. 开启注解扫描

```
<!--开启注解扫描 - 通过其 base-package 属性指定要扫描的包
    1. 多个包之间可以使用, 隔开
    2. 可以使用它们共同的父包
-->
<context:component-scan base-
package="pers.dreamer07.spring"></context:component-
scan>
```

4. 使用注解创建 bean 实例

```

/*
 * 1. 使用四个注解中的任意一个都可
 * 2. 都可以指定value值, 该value值就是 <bean id="...">
    </bean> 中的id属性
 * 3. value的默认值为 当前类名 但是首字母小写
 * */
@Service(value = "userService")
public class UserService {
    public void add(){
        System.out.println("service add()...");
    }
}

```

5. 设计测试类和方法

2.4.2 注解扫描

开启注解扫描, *Spring* 会根据指定的位置自动扫描其中所根据注解创建的 *bean*

(示例1)

```

<!--
base-package: 自动要扫描的包
use-default-filters="false": 不使用默认的过滤器, 会使用自定义配置的
-->
<context:component-scan base-package="com.atguigu" use-
default-filters="false">
    <!--
        context:include-filter: 自定义配置过滤器, 可以配置多个
        type: 类型, 这里是注解
        experssion: 扫描指定的注解类型, 这里是只扫描@Controller类型
    的
    -->
    <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
</context:component-scan>

```

(示例2)


```

<context:component-scan base-package="com.atguigu">
    <!--
        context:exclude-filter: 不扫描过滤器，根据属性不扫描某些类
        type: 类型，这里是注解
        experssion: 扫描指定不扫描的注解类型，这里是不扫描
        @Controller类型的
    -->
    <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
</context:component-scan>

```

2.4.3 使用注解完成属性注入

使用的注解

1. @Autowired: 根据属性类型进行自动装配
2. @Qualifier: 根据属性名称进行注入
3. @Resource: 可以根据类型注入，也可以根据名称进行注入
4. @Value: 注入普通类型属性

@AutoWired和@Qualifier的使用

1. 设计两个类 service 类和 dao 类，在 service 中调用 dao 的方法
2. 使用注解，创建两个类的 bean 实例
3. 在 service 中对应的 dao 属性上添加@Autowired注解

```

@Autowired //根据类型自动注入，不需要写set方法
private BookDAO bookDAO;

```

- @Qualifier是依赖于@Autowired的，需要写在其下面

```

@Autowired //根据类型自动注入,不需要写set方法
/*
 * 根据属性名称进行注入，该注解依赖于@Autowired
 * 需要指定 value 属性为要注入的 bean 实例的value值
(就是之前的id)
 * */
@Qualifier(value = "bookDAOImpl")
private BookDAO bookDAO;

```

- 注意：当根据类型自动装配时，如果该属性是接口，且有多个实现类，这时就需要@Qualifier找到指定的实现类对应的 bean 实例

4. 设计测试类和测试方法

@Resource和@Value的使用

```
/*@Resource的两种使用方式，但由于不是 spring 官方的，所以不推荐使用*/  
1. @Resource //根据类型自动注入  
2. @Resource(name="value") //根据名称自动注入  
/*@value的使用，为普通数据类型属性自动注入*/  
1. @value(value="123")  
   private int num;
```

2.4.4 完全注解开发

1. 使用配置类替换配置文件

```
//1. 添加Configuration表示为配置类  
@Configuration  
//2. 开启注解扫描  
@ComponentScan(basePackages =  
{"pers.dreamer07.spring"},useDefaultFilters = false)  
public class SpringConfig {  
}
```

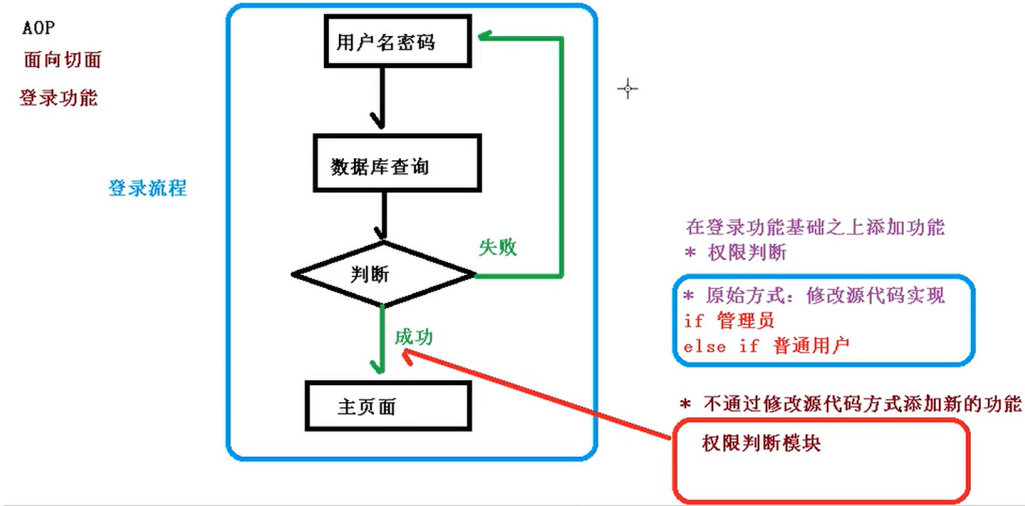
2. 重新设计测试类

```
@Test  
public void test(){  
    //加载配置类  
    ApplicationContext context = new  
AnnotationConfigApplicationContext(SpringConfig.class  
s);  
    BookService bookService =  
context.getBean("bookService", BookService.class);  
    bookService.add();  
}
```

第三章 AOP

3.1 概念

- 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，从而使得业务逻辑各部分之间的耦合度降低，提供程序的可重用性，同时提高开发的效率
- 简单描述：在需要修改/添加新功能时，可以不通过修改源代码的方式，实现修改/添加新功能
- 当 登录功能 需要添加 权限判断 功能

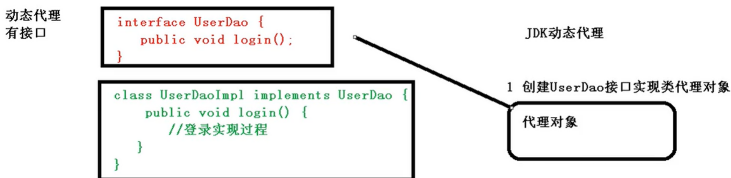


3.2 底层原理

- 动态代理

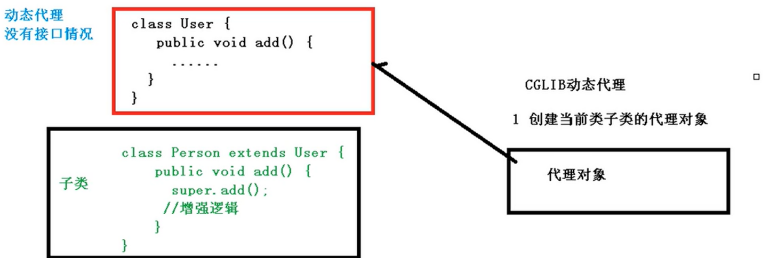
a. 有接口的情况，使用 JDK 动态代理

- 创建接口实现类的代理对象，增强类中的方法



b. 没有接口的情况，使用 CGLIB 动态代理

- 创建当前类子类到的代理对象，增强类中的方法



3.2.1 实现 JDK 动态代理

1. 创建一个接口和一个对应的实现类 (接口)

```
public interface UserDao {  
    public int update(int id,int value);  
  
    public String queryNameById(int id);  
}
```

(实现类)

```
public class UserDaoImpl implements UserDao{  
    @Override  
    public int update(int id, int value) {  
        return id + value;  
    }  
  
    @Override  
    public String queryNameById(int id) {  
        return id + " EMT!!";  
    }  
}
```

2. 创建一个类实现InvocationHandler接口，用来完成增强代码的逻辑

```
public class UserDaoHandler implements  
InvocationHandler {  
    //通过有参构造函数获取被代理类的对象，以便后续执行其中的原  
    方法  
    private Object obj;  
    public UserDaoHandler(Object obj) {  
        this.obj = obj;  
    }  
  
    /**  
     * 该方法会在帮助真实代理类对象实现方法增强，在调用代理类对  
     象的方法前都会调用该方法  
     * @param proxy 真实的代理类对象  
     * @param method 要执行的方法  
     * @param args 执行方法需要的参数  
     * @return  
     * @throws Throwable  
     */  
}
```

```

@Override
public Object invoke(Object proxy, Method
method, Object[] args) throws Throwable {
    //根据不同的方法增加不同的逻辑
    Object res = null;
    if("update".equals(method.getName())){
        System.out.println("修改修改修改!!");
    }else
    if("queryNameById".equals(method.getName())){
        System.out.println("找找找");
    }
    //在完成增加逻辑的同时，还需要执行原方法
    res = method.invoke(obj, args);
    //返回执行的返回值
    return res;
}
}

```

3. 创建一个代理工厂，用来创建代理类

```

public class ProxyFactory {

    /**
     * 传入需要增强方法的对象
     * @param obj
     * @return
     */
    public static Object getProxyInstance(Object
obj){
        InvocationHandler invocationHandler = new
        UserDaoHandler(obj);
        return
        Proxy.newProxyInstance(obj.getClass().getClassLoader
(),obj.getClass().getInterfaces(),invocationHandler)
        ;
    };
}

```

4. 设计测试类和方法

```
public class ProxyTest {  
    @Test  
    public void test(){  
        //调用代理工厂获取增加后的代理类对象  
        UserDao userDao = (UserDao)  
        ProxyFactory.getProxyInstance(new UserDaoImpl());  
        //调用对应的方法  
        int update = userDao.update(20, 999);  
        System.out.println(update);  
    }  
}
```

3.3 操作术语

1. 连接点：类中可以增强的方法，又可以称为连接点
2. 切入点：实际上被增强的方法，称为切入点
3. 通知(增强)
 - 增强的逻辑部分称为通知(增强)
 - 通知的分类
 - i. 前置通知：在原方法前执行
 - ii. 后置通知：返回值后执行(有异常时不执行)
 - iii. 环绕通知：(较特殊，具体看实例)
 - iv. 异常通知：抛出异常时执行
 - v. 最终通知：在原方法后执行(有异常时仍然执行)
4. 切面：将通知应用到切入点的过程

3.4 AOP操作 - 准备工作

1. Spring 空间一般都是基于 AspectJ 实现AOP 操作
 - Aspectj：一个独立的 AOP 框架，但一般和 Spring 框架一起使用。进行 AOP 操作
2. 基于 AspectJ 实现 AOP 操作
 - 基于 xml 配置文件实现
 - 基于注解方式实现(使用)
3. 在项目中引入 AOP 相关依赖

```
<!--spring aspects模块-->  
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-aspects</artifactId>
```

```
<version>${spring.version}</version>
</dependency>
<!-- AspectJ Runtime(jrt) -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>${aspectj.version}</version>
</dependency>
<!-- cglib -->
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>3.2.4</version>
</dependency>
```

4. 切入点表达式

- 作用：知道哪个类里面的哪个方法进行增强
- 语法结构：execution([权限修饰符][返回类型][全类名].[方法名称][参数列表])
- 举例

```
/* 代表任意权限
//返回类型可以省略
execution(*
pers.dreamer07.spring.dao.BookDAO.add(...))
```

3.5 AOP操作 - AspectJ注解

基本使用

1. 设计被增强的类和方法
2. 设计增强类和以及对应方法(编写增强逻辑)
 - 在增强类里面，设计不同的方法代表不同的通知类型
3. 进行通知的配置
 - a. 在 spring 配置文件/配置类中，开启注解扫描
 - b. 使用注解创建 增强类 和 被增强类 的 bean 实例
 - c. 在增强类上添加 @Aspect
 - d. 在 spring 配置文件中开启生成代理对象(需要先引入 aop 容器)

```
<beans
xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

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

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

</beans>
```

4. 配置不同类型的通知

- 在增强类里，在作为通知方法上面添加 对应的通知类型注解，使用切入点表达式配置


```
@Before(value = "execution(*
pers.dreamer07.spring.aopanno.User.add(..))")
//添加 前置通知的类型注解，再使用切入点表达式进行配置
public void before(){
    System.out.println("前置通知");
}
```

5. 设计测试类和测试方法

```
@Test
public void test(){
    ApplicationContext context = new
    ClassPathXmlApplicationContext("META-
    INF/bean1.xml");
    User user = context.getBean("user", User.class);
    user.add();
}
```

6. 补充 - 其他通知类型的注解

```
//后置通知(在返回值后执行，抛出异常不执行)
@AfterReturning(value = "execution(*
pers.dreamer07.spring.aopanno.User.add(..))")
public void afterReturning(){
    System.out.println("后置通知");
}

//异常通知(抛出异常时执行)
@AfterThrowing(value = "execution(*
pers.dreamer07.spring.aopanno.User.add(..))")
public void afterThrowing(){
    System.out.println("异常通知");
}

//最终通知(原方法和环绕通知执行后执行，抛出异常执行)
@After(value = "execution(*
pers.dreamer07.spring.aopanno.User.add(..))")
public void after(){
    System.out.println("最终通知");
}

//环绕通知(原方法抛出异常后，该内部定义的后面的通知就不会执行)
```

```
@Around(value = "execution(*
pers.dreamer07.spring.aopanno.User.add(..))")
public void around(ProceedingJoinPoint
proceedingJoinPoint) throws Throwable { //需要在参数中
获取ProceedingJoinPoint实例对象
    //在方法执行前执行 - 在前置通知前执行
    System.out.println("在方法执行前执行");

    //执行被增强的方法
    proceedingJoinPoint.proceed();

    //在方法执行后执行 - 在后置和最终通知前执行
    System.out.println("在方法执行后执行");
}
```

细节补充

1. 抽取公共的切入点部分

```
@Pointcut(value = "execution(*
pers.dreamer07.spring.aopanno.User.add(..))") //提取
公共的切入点
public void pointName(){};

//value值可以直接通过调用 Pointcut 注解的方法获取
@Before(value = "pointName()")
public void before(){
    System.out.println("前置通知");
}
```

2. 当多个增强类对同一个方法进行增强时，可以设置增强类的优先级

- 在增强类上添加 @Order(数值型数据) - 数字越小，优先级越高

```

@Component
@Aspect
@Order(1) //通过@Order设置的数值型数据，数据越小，
          优先级越高
public class PersonProxy {

    @Before(value = "execution(*
pers.dreamer07.spring.aopanno.User.add(..))")
    public void before(){
        System.out.println("Person before");
    }
}

```

完全注解Aop

```

@Configuration //标识配置类
@ComponentScan(basePackages = {"pers.dreamer07.spring"}) //
注解扫描
@EnableAspectJAutoProxy(proxyTargetClass = true) //开启
AspectJ生成代理对象
public class SpringConfig {
}

```

3.6 AOP操作 - AspectJ配置文件

1. 设计增强类和被增强类
2. 在 spring 配置文件中创建两个类的 bean 实例
3. 在 spring 配置文件中配置 aop 增强

```

<!-- 配置aop增强 -->
<aop:config>
    <!-- 配置切入点
            id = 切入点标识
            expression = 切入点表达式
        -->
    <aop:pointcut id="p" expression="execution(*
pers.dreamer07.spring.aopxml.Book.buy(..))"/>

    <!-- 配置切面
            ref = 增强类的bean实例

```

```

-->
<aop:aspect ref="bookProxy">
    <!-- 将增强类中'通知'应用在指定的切入点上
        aop:before = 标识是前置方法
        method = 增强类中定义的'通知'
        pointcut-ref = 指定的切入点
-->
    <aop:before method="before" pointcut-
ref="p"/>
</aop:aspect>
</aop:config>

```

4. 设计测试类和方法

第四章 JdbcTemplate

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

4.1 准备工作

1. 导入相关依赖

```

<!--Spring JdbcTemplate-->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- Mysql 驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.18</version>
</dependency>
<!-- 如果需要整合其他 ORM 框架，就需要导入该依赖 -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${spring.version}</version>
</dependency>
<!-- Druid 数据库连接池 -->

```

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.22</version>
</dependency>
```

2. 创建 properties 配置文件

```
emt.username=root
emt.password=Dreamer07
emt.driverClassName=com.mysql.cj.jdbc.Driver
emt.url=jdbc:mysql://localhost:8080/spring_study
```

3. 在 spring 配置文件中配置数据库连接池的属性

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

       xmlns:context="http://www.springframework.org/schema/context"
       "

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

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

  <!-- 引入外部文件 -->
  <context:property-placeholder
    location="jdbc.properties"/>

  <!-- 使用外部文件中的属性配置Druid连接池对象 -->
  <bean id="dataSource"
    class="com.alibaba.druid.pool.DruidDataSource">
    <property name="username" value="${emt.username}">
  </property>
    <property name="password" value="${emt.password}">
  </property>
```

```

        <property name="driverClassName"
value="${emt.driverClassName}"></property>
        <property name="url" value="${emt.url}"></property>
    </bean>
</beans>

```

4. 在 spring 配置文件中配置 JdbcTemplate 对象，注入 DataSource

```

<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
    <!-- 通过其set方法完成数据库连接池的注入 -->
    <property name="dataSource" ref="dataSource"></property>
</bean>

```

5. 开启注解扫描，

6. 在 baseDAO 中注入 JdbcTemplate 对象

```

@Repository
public class BaseDao<E> {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    private Class<E> clazz;

    /**
     * 通过无参构造函数初始化Class<E>
     * 具体实现请看 JDBC
     * 这里之所以放在无参构造函数中是因为Spring在创建bean实例时，并不是
     通过子类对象调用的
     * 会导致Type type是Object类型，最后报错，
     * 但又不能返回静态代码块中(因为只执行一次)，所以放在无参构造函数中
     * 使Spring在创建bean实例时不会报错，当创建子类对象调用父类构造器时
     又能获取指定的泛型参数
     */
    public BaseDao(){
        Type type = this.getClass().getGenericSuperclass();
        if(!type.getTypeName().contains("Object")){
            ParameterizedType pType = (ParameterizedType)
type;

            Type[] types = pType.getActualTypeArguments();
            clazz = (Class<E>) types[0];

```

```
        }  
    }  
}
```

4.2 添加修改删除

从操作上来说都是一样的，只需要返回影响的行数即可

```
public int update(String sql, Object... args){  
    return jdbcTemplate.update(sql, args);  
}
```

4.3 查询

4.3.1 查询某个特殊值

```
public <T> T querySpecial(String sql, Class<T>  
clazz, Object... args){  
    /*  
     * public <T> T queryForObject(String sql, Class<T>  
requiredType, Object... args)  
     * 第一个参数sql: sql语句  
     * 第二个参数requiredType: 返回值的数据类型  
     * 第三个参数args: 需要填充占位符的数据  
     * */  
    return jdbcTemplate.queryForObject(sql, clazz, args);  
};
```

4.3.2 查询一个对象

```

public E query(String sql, Object... args) {
    /**
     * <T> T queryForObject(String sql, RowMapper<T>
     * rowMapper, Object... args)
     * 第一个参数sql: sql语句
     * 第三个参数rowMapper: 和Apache-DBUtils中的
     * ResultSetHandler类一样, 根据查询结果的返回值类型, 选择相对应的实现类
     * 第三个参数args: 需要填充占位符的数据
     */
    return jdbcTemplate.queryForObject(sql, new
    BeanPropertyRowMapper<E>(clazz), args);
}

```

4.3.3 查询多个对应构成的集合

```

/**
 * 查询所有记录
 * @param sql
 * @param args
 * @return
 */
public List<E> queryList(String sql, Object... args) {
    /**
     * public <T> List<T> query(String sql, RowMapper<T>
     * rowMapper, @Nullable Object... args)
     * 第一个参数sql: sql语句
     * 第三个参数rowMapper: 和Apache-DBUtils中的
     * ResultSetHandler类一样, 根据查询结果的返回值类型, 选择相对应的实现类
     * 第三个参数args: 需要填充占位符的数据
     */
    return jdbcTemplate.query(sql, new
    BeanPropertyRowMapper<E>(clazz), args);
}

```

4.4 批量操作

- 这里以 批量添加 为例

BaseDao.java


```
/**
 * 批量操作
 * @param sql
 * @param list
 */
public void addBatch(String sql, List<Object[]> list){
    /**
     * public int[] batchUpdate(String sql, List<Object[]>
    batchArgs)
     * 第二个参数 batchArgs: 辅助保存需要添加的数据
     */
    jdbcTemplate.batchUpdate(sql, list);
}
```

测试方法

```
@Test //批量添加
public void testAddBatch(){
    //将需要填充到sql语句的占位符的数据保存到数组中
    Object[] vals1 = new Object[]{"EMM!!", "OKK"};
    Object[] vals2 = new Object[]{"EMP!!", "OKK"};
    Object[] vals3 = new Object[]{"EMH!!", "OKK"};

    //将数组保存到List集合中
    ArrayList<Object[]> list = new ArrayList<>();
    list.add(vals1);
    list.add(vals2);
    list.add(vals3);

    //调用方法
    userService.addUseBatch(list);
}
```

- 批量修改和批量删除的操作都一样，都是调用 `batchUpdate()` 方法
需要注意的是 `Object[]` 数组中保存数据的顺序，需要和 `sql` 填充占位符的数据顺序相同

第五章 事务

5.1 复习

- 概念：事务时数据库操作的最基本单元，由一条/多条sql语句组成，要么都成功，要么都不成功
- **ACID属性**
 - a. 原子性(Atomicity) - 事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生
 - b. 一致性(Consistency) - 事务必须使数据库从一个一致性状态变换到另一个一致性状态
 - c. 隔离性(Isolation) - 一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰
 - d. 持久性(Durability) - 一个事务的提交，对数据库的改变就是永久性的

5.2 搭建 Spring 操作事务环境

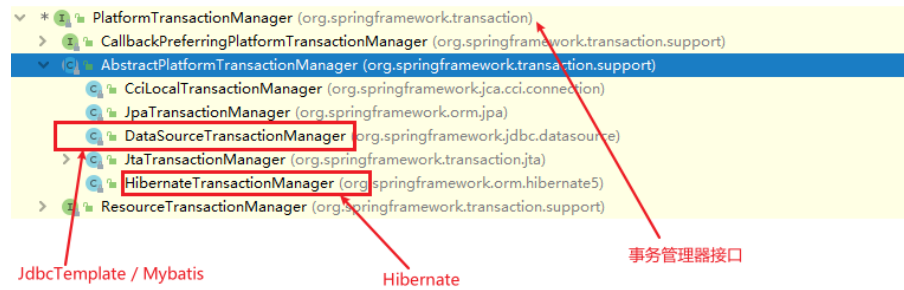
1. 准备相关数据库和数据表
2. 创建外部文件 `jdbc.properties`
3. 创建 **Spring** 配置文件，开启注解扫描，获取外部文件中的数据配置数据库连接池，以及创建 `JdbcTemplate` 的 `bean` 实例和输入连接池依赖
4. 设计对应的 **ORM** 类
5. 设计对应的 `dao` 和 `service`
6. 设计测试方法和测试类

5.3 Spring 事务管理

5.3.1 介绍

1. 建议添加事务到 **JavaEE** 三层架构的 **Service** 层(业务逻辑层)
2. 在 **Spring** 进行事务管理操作有两种方式
 - 编程式事务管理(使用代码完成，比较臃肿)
 - 声明式事务管理(推荐)
3. 声明式事务管理的实现
 - a. 基于注解方式
 - b. 基于 `xml` 配置文件方式
4. 声明式事务管理的底层，就是 **AOP**
5. **Spring** 事务管理的 **API**

- Spring 中提供了一个接口，代表事务管理器，针对不同的框架有不同的实现类



5.3.2 声明式事务管理(基于 XML)

基本使用

1. 配置事务管理器
2. 配置通知

```

<!-- 配置通知 -->
<tx:advice id="txadvice">
    <!-- 配置事务参数 -->
    <tx:attributes>
        <!--
        tx:method = 指定方法
        name = 符合name属性值规则的方法将被添加上该事务
            - 可以使用通配符，比如: name=transfer*
        propagation = 事务管理的参数配置
        -->
        <tx:method name="transferMoney"
propagation="REQUIRED"/>

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

```

3. 配置切入点和切面

```
<!-- 配置切入点和切面 -->
<aop:config>
    <!-- 配置切入点 -->
    <aop:pointcut id="pt" expression="execution(*
pers.dreamer07.spring.service.AccountService.*
(..))"/>

    <!-- 配置切面 -->
    <aop:advisor advice-ref="txadvice" pointcut-
ref="pt"></aop:advisor>
</aop:config>
```

5.3.3 声明式事务管理(基于注解) - 推荐

基本使用

1. 在配置文件中配置对应的事务管理器的 bean 实例

```
<!-- 配置事务管理器的bean实例 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSource
eTransactionManager">
    <!-- 注入数据库连接池(数据源) -->
    <property name="dataSource" ref="dataSource">
</property>
</bean>
```

2. 引入 tx 名称空间

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

xmlns:context="http://www.springframework.org/schema
/context"

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

xsi:schemaLocation="http://www.springframework.org/s
chema/beans
http://www.springframework.org/schema/beans/spring-
beans.xsd

    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring
-context.xsd

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

3. 通过 事务管理器 在 xml 中开启事务注解

```
<!-- 开启事务注解,配置对应的事务管理器 -->
<tx:annotation-driven transaction-
manager="transactionManager"></tx:annotation-driven>
```

4. 在 service 类 / service 类中的方法上面添加 @Transactional 注解

- 添加到类上，代表该类中的所有方法都添加事务
- 添加到方法上，代表该方法添加事务
- 注意：Transactional 注解中的 transactionManager 属性默认值为 transactionManager，如果配置的事务管理器 id 不同 / 有多个事务管理器时，需要手动指定

参数配置

- 在 @Transactional 中可以配置以下参数

Propagation	propagation()	default org.springframework.transaction.annotation.Propagation
Isolation	isolation()	default org.springframework.transaction.annotation.Isolation.DEFAULT
int	timeout()	default -1
boolean	readOnly()	default false
Class<? extends Throwable>	rollbackFor()	default {}
String[]	rollbackForClassName()	default {}
Class<? extends Throwable>	noRollbackFor()	default {}

1. propagation: 事务传播行为

- 事务传播行为: 多事务方法之间进行调用, 在这个过程中事务是如何进行管理的
- 事务方法: 对数据库表数据进行变化的操作(查找不算)
- Spring 框架针对事务传播行为定义了七种模式(默认情况下是第一种)

传播属性	描述
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, 并在它自己的事务内运行
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	如果有事务在运行, 当前的方法就在这个事务内运行. 否则它可以不运行在事务中.
NOT_SUPPORTED	当前的方法不应该运行在事务中. 如果有运行的事务, 将它挂起
MANDATORY	当前的方法必须运行在事务内部, 如果没有正在运行的事务, 就抛出异常
NEVER	当前的方法不应该运行在事务中. 如果有运行的事务, 就抛出异常
NESTED	如果有事务在运行, 当前的方法就应该在这个事务的嵌套事务内运行. 否则, 就启动一个新的事务, 并在它自己的事务内运行.

事务传播行为

事务方法: 对数据库表数据进行变化的操作

```
@Transactional
public void add() {
    //调用update方法
    update();
}
```

```
public void update() {
}
```

Spring框架事务传播行为有7种

REQUIRED 如果add方法本身有事务, 调用update方法之后, update使用当前add方法里面事务
如果add方法本身没有事务, 调用update方法之后, 创建新事务

REQUIRED_NEW 使用add方法调用update方法, 如果add无论是否有事务, 都创建新的事务

2. ioslation: 事务隔离级别

- 在不考虑隔离性的情况下, 并发事务之间可能会产生三个问题
 - 脏读: 一个未提交的事务读取到了另一个未提交事务的数据
 - 不可重复读: 一个未提交的事务读取到了另一个已提交事务的修改数据
 - 幻(虚)读: 一个未提交的事务读取到了另一个已提交事务的提交数据

- 针对不同的情况，设置对应的隔离级别

	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 mysql8中已修改
SERIALIZABLE (串行化)	无	无	无

- 详情还是请看 Mysql/学习笔记

3. timeout: 超时时间

- 当事务操作超过了指定的时间还未提交时，就进行回滚操作
- 默认值是-1(永不超时)，设置时间以秒为单位进行

4. readOnly: 是否只读

- 默认是为false，表示可以进行 增删查改
- 如果修改为 true，表示只能进行 查询操作

5. rollbackFor: 回滚

- 设置当出现了哪些异常才进行事务回滚

6. noRollbackFor: 不回滚

- 设置当出现了哪些异常不进行事务回滚

完成注解开发

```

@Configuration //配置类
@ComponentScan(basePackages = "pers.dreamer07.spring.*") //
开启注解扫描
@EnableTransactionManagement //开启事务注解
public class TxConfig {
    /*
     * 使用 @Bean 注解创建需要的 bean 实例
     * 该注解下的方法就是需要创建 bean 实例所对应的类型
     * 格式如下:
     * @Bean
     * public 类型 get类型(可以接受IOC容器已创建的bean实例){
     *     return xxxxxxxx;
     * }
     */
    @Bean //数据库连接池
    public DruidDataSource getDruidDataSource(){
        FileReader fr = null;
    }
}

```

```
        DruidDataSource druidDataSource = new
        DruidDataSource();
        try {
            //加载配置文件
            fr = new
        FileReader("src/main/resources/jdbc.properties");
            Properties prop = new Properties();
            prop.load(fr);

            //配置数据库连接池

            druidDataSource.setDriverClassName(prop.getProperty("emt.dr
            iverClassName"));

            druidDataSource.setUsername(prop.getProperty("emt.username"
            ));

            druidDataSource.setPassword(prop.getProperty("emt.password"
            ));

            druidDataSource.setUrl(prop.getProperty("emt.url"));
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                try {
                    if (fr != null) {
                        fr.close();
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            return druidDataSource;
        }

        @Bean //JdbcTemplate
        /*
        * DruidDataSource dataSource - 根据类型可以从 IOC 容器中获取
        对应的 数据源
        * */
```



```
public JdbcTemplate getJdbcTemplate(DataSource
dataSource){
    JdbcTemplate jdbcTemplate = new JdbcTemplate();
    jdbcTemplate.setDataSource(dataSource);
    return jdbcTemplate;
}

@Bean //事务管理器
public DataSourceTransactionManager
getDataSourceTransactionManager(DataSource dataSource){
    DataSourceTransactionManager transactionManager =
new DataSourceTransactionManager();
    transactionManager.setDataSource(dataSource);
    return transactionManager;
}
}
```

第六章 Spring5 新功能
