

Spring

- 文件关系 ——> 环境搭建
- 类 关系 ——> XXX

概述

Spring官网

官网地址：<https://spring.io/projects/spring-framework#overview>

压缩包下载地址：<https://repo.spring.io/release/org/springframework/spring/>

<http://repo.springsource.org/libs-release-local/org/springframework/spring>

- dist : jar
- docs : 文档

源码地址：<https://github.com/spring-projects/spring-framework>

- jar
- javadoc文档
- sources源码

- docs : API 和 开发规范
- libs : jar包 和 源码
- schema : 约束

Spring是什么？

Spring 是 分层的 Java SE/EE应用 **full-stack** 轻量级开源框架，以 **IoC** (Inverse Of Control : 反转控制) 和 **AOP** (Aspect Oriented Programming : 面向切面编程) 为内核，提供了 **展现层**

SpringMVC 和 **持久层 Spring JDBC** 以及 业务层事务管理 等众多的 企业级应用技术，还能 整合 开源世界众多著名的 第三方框架和类库，逐渐成为使用最多的Java EE 企业应用开源框架。

核心解释

spring是一个开源框架。

spring是为了简化企业开发而生的，使得开发变得更加优雅和简洁。

spring是一个**IOC**和**AOP**的容器框架。

- IOC : 控制反转
- AOP : 面向切面编程
- 容器 : 包含并管理 应用对象 的生命周期，就好比用桶装水一样，spring就是桶，而对象就是水。容器就是放对象的地方。

Spring的发展历程

- 1997 年 IBM提出了EJB 的思想
- 1998 年 , SUN制定开发标准规范 EJB1.0
- 1999 年 , EJB1.1 发布
- 2001 年 , EJB2.0 发布
- 2003 年 , EJB2.1 发布
- 2006 年 , EJB3.0 发布
- **Rod Johnson (spring 之父)**
 - Expert One-to-One J2EE Design and Development(2002)
 - 阐述了 J2EE 使用EJB 开发设计的优点及解决方案
 - Expert One-to-One J2EE Development without EJB(2004)
 - 阐述了 J2EE 开发不使用 EJB的解决方式 (Spring 雏形)
- **2017 年 9 月份发布了 spring 的最新版本 0 spring 5.0 通用版 (GA)**

使用spring的优点

1. Spring通过**DI、AOP和消除样板式代码**来简化企业级Java开发
2. Spring框架之外还存在一个构建在核心框架之上的庞大**生态圈**，它将Spring扩展到不同的领域，如Web服务、REST、移动开发以及NoSQL
3. **低侵入式设计**，代码的污染极低
4. 独立于各种应用服务器，基于Spring框架的应用，可以真正实现**Write Once, Run Anywhere**的承诺
5. Spring的**IoC容器**降低了**业务对象替换**的复杂性，提高了组件之间的**解耦**
6. Spring的**AOP**支持允许将一些**通用任务**如安全、事务、日志等进行**集中式处理**，从而提供了**更好的复用**
7. Spring的**ORM和DAO**提供了与**第三方持久层**框架的良好整合，并简化了底层的数据库访问
8. Spring的高度开放性，并不强制应用完全依赖于Spring，开发者可**自由选用**Spring框架的**部分或全部**

方便解耦，简化开发

消除样板代码

Spring通过**DI、AOP和消除样板式代码**来简化企业级Java开发。

低侵入式设计

低侵入式设计，代码的污染极低

IoC降低对象依赖

通过 Spring提供的 **IoC容器**，可以将**对象间的依赖关系**交由 Spring进行控制，避免**硬编码**所造成**过度程序耦合**。用户也不必再为单例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。

Spring的**IoC容器**降低了**业务对象替换**的复杂性，提高了组件之间的**解耦**

AOP降低方法依赖

通过 Spring 的 **AOP** 功能，方便进行面向切面的编程，许多不容易用传统 OOP 实现的功能可以通过 AOP 轻松应付。

Spring 的 **AOP** 支持允许将一些 **通用任务** 如安全、事务、日志等进行 **集中式处理**，从而提供了 **更好的复用**

声明式事务的支持

可以将我们从单调烦闷的事务管理代码中解脱出来，通过 **声明式** 方式灵活的进行事务的管理，提高开发效率和质量。

方便程序的测试

可以用非容器依赖的编程方式进行几乎所有的测试工作，测试不再是昂贵的操作，而是随手可做的事情。

独立于各种应用服务器

基于 Spring 框架的应用，可以真正实现 **Write Once, Run Anywhere** 的承诺。

生态圈 支持各种框架

Spring 框架之外还存在一个构建在核心框架之上的庞大 **生态圈**，它将 Spring 扩展到不同的领域，如 Web 服务、REST、移动开发以及 NoSQL。

提供了对各种优秀框架（Struts、Hibernate、Hessian、Quartz 等）的直接支持，Spring 可以降低各种框架的使用难度。

Spring 的 **ORM** 和 **DAO** 提供了与 **第三方持久层** 框架的良好整合，并简化了底层的数据库访问。

降低 JavaEE API 的使用难度

Spring 对 **JavaEE API**（如 JDBC、JavaMail、远程调用等）进行了薄薄的 **封装层**，使这些 API 的使用难度大为降低。

高度开放性

Spring 的 **高度开放性**，并不强制应用完全依赖于 Spring，开发者可 **自由选用** Spring 框架的 **部分或全部**。

Java 源码是经典学习范例

Spring 的源代码设计精妙、结构清晰、匠心独用，处处体现着大师对 Java 设计模式灵活运用以及对 Java 技术的高深造诣。它的源代码无疑是 Java 技术的最佳实践的范例。

如何简化开发

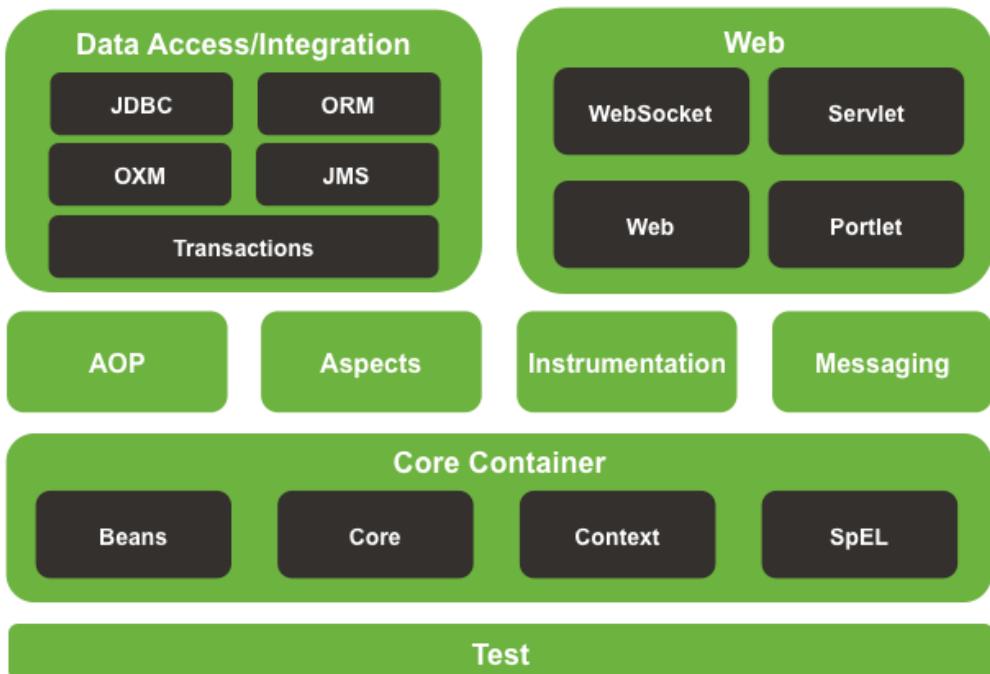
- 基于 POJO 的轻量级和最小侵入性编程
- 通过依赖注入和面向接口实现松耦合
- 基于切面和惯例进行声明式编程
- 通过切面和模板减少样板式代码

spring 的模块划分

- 绿色框 表示 模块。
- 黑色框 表示 具体依赖 jar 包。



Spring Framework Runtime



Test

Spring的单元测试模块

Core Container

核心容器模块，也就是IOC。Spring的核心之一，它的IOC部分。Spring的任何部分想要运行都必须有核心容器的支持。

- spring-beans-xxx.jar
- spring-core-xxx.jar
- spring-context-xxx.jar
- spring-expression-xxx.jar

AOP+Aspects

面向切面编程模块

Instrumentation

提供了class instrumentation支持和类加载器的实现来在特定的应用服务器上使用,几乎不用

Messaging

包括一系列的用来映射消息到方法的注解,几乎不用

Data Access/Integration

数据的获取/整合模块，包括了JDBC，ORM，OXM，JMS和事务模块

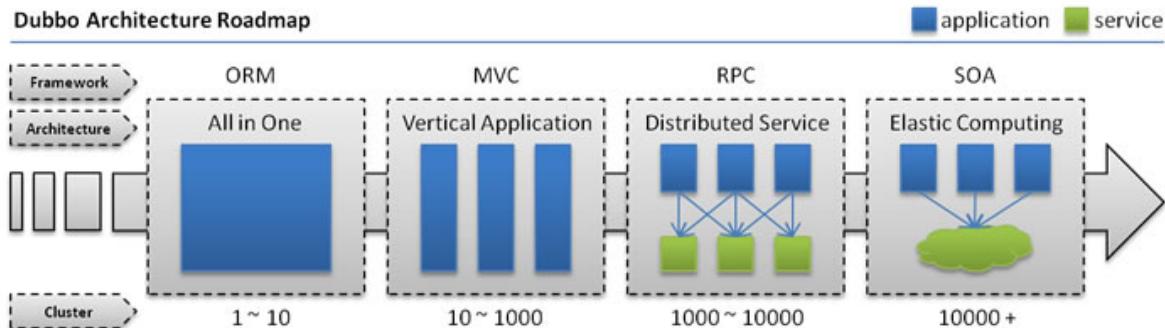
Web

提供面向web整合特性

框架演进

诞生背景

随着互联网的发展，网站应用的规模不断扩大，常规的垂直应用架构已无法应对，分布式服务架构以及流动计算架构势在必行，亟需一个治理系统确保架构有条不紊的演进。



(图片来源：<http://dubbo.apache.org/zh-cn/docs/user/preface/background.html>)

单一应用架构

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的数据访问框架(ORM，Object Relational Mapping对象关系映射)是关键。

垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小(这里说的应该是集群)，提升效率的方法之一是将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的Web框架(MVC)是关键。

之前的单一应用架构，项目或者说应用做好了，打个jar或war包，运行在服务器里，就ok了。

但是如果项目比较大，比如电商网站，包括用户模块、商品模块、购物车模块、搜索模块...，单一应用架构就是打到一个包里，这个jar|war包会变得非常大，里面包含非常多的业务逻辑，运行会变得比较麻烦。

将应用拆分，也就是不同的模块。一个jar包拆分成N多个jar包，或者说模块。这样一来，每个小的jar包启动时间短，不同的服务请求，就去不同的服务器里去执行。

分布式服务架构

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用及整合的分布式服务框架(RPC，用于不同服务器之间的通信)是关键。

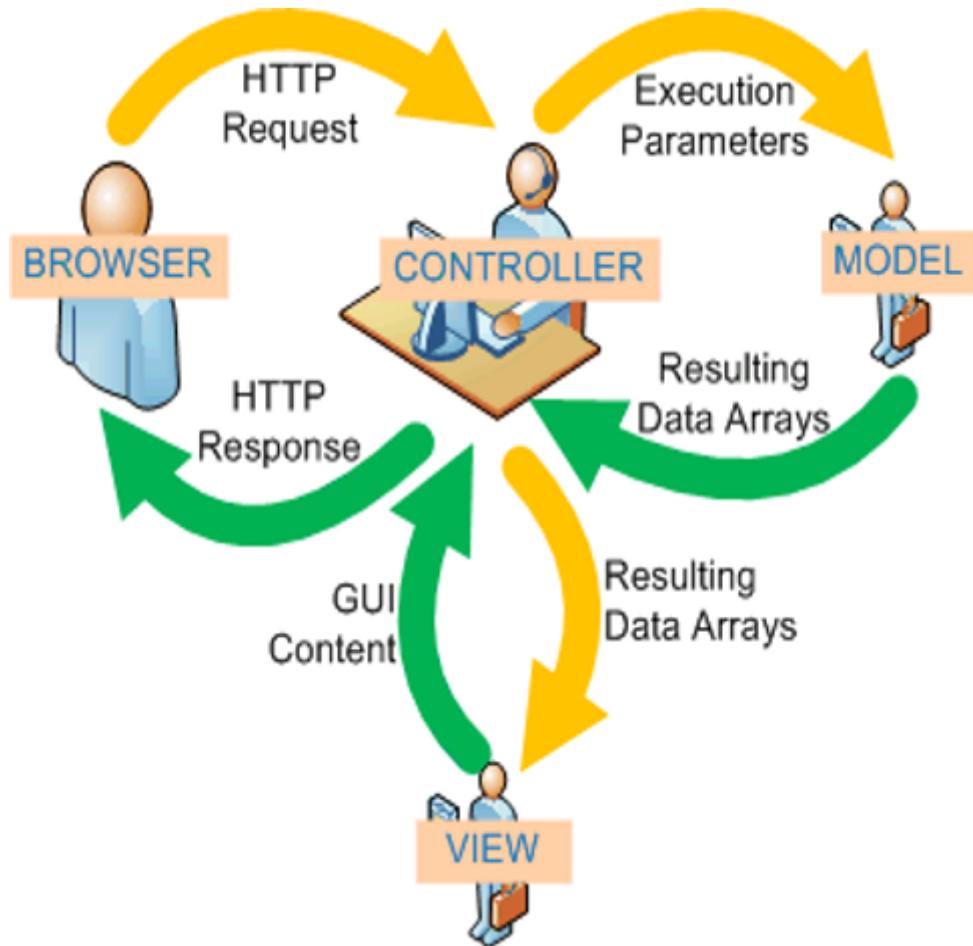
流动计算架构

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键。

Java主流框架演变

1. JSP+Servlet+JavaBean

2. MVC三层架构



3. 使用EJB进行应用的开发，但是EJB是重量级框架（在使用的时候，过多的接口和依赖，侵入性强），在使用上比较麻烦

4. Struts1/Struts2+Hibernate+Spring

5. SpringMVC+Mybatis+Spring

6. SpringBoot开发，约定大于配置

程序的耦合

程序间的依赖关系

- **类之间** 的依赖
- **方法间** 的依赖

耦合

耦合性(Coupling)，也叫耦合度，是对模块间 **关联程度**的度量。

耦合的强弱 取决于

- 模块间接口的复杂性
- 调用模块的方式
- 通过界面传送数据的多少

模块间的耦合度是 指模块之间的 依赖关系，包括

- 控制关系

- 调用关系
- 数据传递关系

模块间联系越多，其耦合性越强，同时表明其独立性越差(降低耦合性，可以提高其独立性)。耦合性存在于各个领域，而非软件设计中独有的，但是我们只讨论软件工程中的耦合。

在软件工程中，耦合指的就是对象之间的依赖性。对象之间的耦合越高，维护成本越高。因此对象的设计应使类和构件之间的耦合最小。软件设计中通常用耦合度和内聚度作为衡量模块独立程度的标准。划分模块的一个准则就是高内聚低耦合。

它有如下分类：

- **内容耦合**。内容耦合是最高程度的耦合，应该避免使用之。
 - 一个模块直接修改或操作另一个模块的数据
 - 一个模块不通过正常入口而转入另一个模块
- **公共耦合**。在具有大量公共耦合的结构中，确定究竟是哪个模块给全局变量赋了一个特定的值是十分困难的。
 - 两个或两个以上的模块共同引用一个全局数据项
- **外部耦合**。
 - 一组模块都访问同一全局简单变量而不是同一全局数据结构，而且不是通过参数表传递该全局变量的信息。
- **控制耦合**。
 - 一个模块通过接口向另一个模块传递一个控制信号，接受信号的模块根据信号值而进行适当的动作。
- **标记耦合**。
 - 若一个模块A通过接口向两个模块B和C传递一个公共参数，那么称模块B和C之间存在一个标记耦合。
- **数据耦合**。数据耦合是最低的一种耦合形式，系统中一般都存在这种类型的耦合，因为为了完成一些有意义的功能，往往需要将某些模块的输出数据作为另一些模块的输入数据。
 - 模块之间通过参数来传递数据
- **非直接耦合**。
 - 两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的。

总结：耦合是影响软件复杂程度和设计质量的一个重要因素，在设计上我们应采用以下原则：

如果模块间必须存在耦合，就尽量使用**数据耦合，少用控制耦合，限制公共耦合的范围，尽量避免使用内容耦合**。

内聚与耦合

内聚标志一个模块内各个元素彼此结合的紧密程度，它是**信息隐蔽和局部化**概念的自然扩展。内聚是从功能角度来度量模块内的联系，一个好的内聚模块应当**恰好做一件事**。它描述的是模块内的功能联系。

耦合是软件结构中各模块之间相互连接的一种度量，耦合强弱取决于模块间接口的复杂程度、进入或访问一个模块的点以及通过接口的数据。程序讲究的是低耦合，高内聚。**就是同一个模块内的各个元素之间要高度紧密，但是各个模块之间的相互依存度却要不那么紧密。**

内聚和耦合是密切相关的，同其他模块存在高耦合的模块意味着低内聚，而高内聚的模块意味着该模块同其他模块之间是低耦合。在进行软件设计时，应力争做到高内聚，低耦合。

解耦

降低程序间的依赖关系。

- 反射(字符串) + 配置文件
- 工厂模式：
 - 在实际开发中我们可以把三层的对象都使用配置文件配置起来。
 - 当启动服务器应用加载的时候，让一个类中的方法通过读取配置文件，把这些对象创建出来并存起来。
 - 在接下来的使用的时候，直接拿过来用就好了。那么，这个读取配置文件，创建和获取三层对象的类就是**工厂**。

实际开发中

- 编译期不依赖，运行时才依赖。

```
1 | DriverManager.registerDriver(new com.mysql.jdbc.Driver());//new 拴合严重
2 | Class.forName("com.mysql.jdbc.Driver");//反射
```

解耦思路：

- 使用反射来创建对象，而避免使用new关键字
- 通过读取配置文件来获取要创建的对象全限定类名

工厂模式解耦

- 在实际开发中我们可以把三层的对象都使用配置文件配置起来。
- 当启动服务器应用加载的时候，让一个类中的方法通过读取配置文件，把这些对象反射创建出来并存起来。
- 在接下来的使用的时候，直接拿过来用就好了。那么，这个读取配置文件，创建和获取三层对象的类就是**工厂**。

类解耦：

- Bean：可重用组件
- JavaBean：用java语言编写的可重用组件

示例

- 需要一个配置文件来配置我们的service和dao
 - 配置的内容：唯一标识 = 全限定类名(key=value)
 - xml或properties
- 通过读取配置文件中配置的内容，反射创建对象

bean.properties

```
1 | accountService=com.learn.service.impl.AccountServiceImpl  
2 | accountDao=com.learn.dao.impl.AccountDaoImpl
```

BeanFactory

一个创建Bean对象的工厂，它就是创建我们的service和dao对象的。

```
1 | public class BeanFactory {  
2 |     //定义一个Properties对象  
3 |     private static Properties props;  
4 |  
5 |     //定义一个Map,用于存放我们要创建的对象,我们把它称之为容器  
6 |     private static Map<String, Object> beans;  
7 |  
8 |     //使用静态代码块为Properties对象赋值  
9 |     static {  
10 |         try {  
11 |             //实例化对象  
12 |             props = new Properties();  
13 |             //获取properties文件的流对象。  
14 |             //不能new FileInputStream(), 因为web工程部署后src路径用不了。绝对路径  
肯定不行, cdef盘  
15 |             //所以通过类加载器来获取。  
16 |             //在resources下创建的文件, 最后会成为类的根路径下的一个文件  
17 |             InputStream in =  
18 |                 BeanFactory.class.getClassLoader().getResourceAsStream("bean.properties");  
19 |             props.load(in);  
20 |             //实例化容器  
21 |             beans = new HashMap<String, Object>();  
22 |             //取出配置文件中所有的key  
23 |             Enumeration<String> keys = props.keys();  
24 |             //遍历枚举  
25 |             while (keys.hasMoreElements()) {  
26 |                 //取出每个key  
27 |                 String key = keys.nextElement().toString();  
28 |                 //根据key获取value  
29 |                 String beanPath = props.getProperty(key);  
30 |                 //反射创建对象  
31 |                 Object value = Class.forName(beanPath).newInstance();  
32 |                 //把key和value存入容器中  
33 |                 beans.put(key, value);  
34 |             }  
35 |         } catch (Exception e) {  
36 |             throw new ExceptionInInitializerError("初始化properties失败");  
37 |         }  
38 |     }  
39 |  
40 |     //多例  
41 |     public static Object getBean(String beanName) {  
42 |         Object bean;  
43 |         try {  
44 |             String beanPath = props.getProperty(beanName);  
45 |             bean = Class.forName(beanPath).newInstance();  
46 |         } catch (Exception e) {  
47 |             e.printStackTrace();  
48 |         }  
49 |         return bean;
```

```
49     }
50
51     //单例
52     public static Object getBean(String beanName) {
53         return beans.getBean(beanName);
54     }
55 }
```

test

```
1 public static void main(String[] args) {
2     AccountService as =
3     (AccountService) BeanFactory.getBean("accountService");
4     as.xxx();
5 }
```

控制反转

工厂模式解耦：

- 在实际开发中我们可以把三层的对象都使用配置文件配置起来。
- 当启动服务器应用加载的时候，让一个类中的方法通过读取配置文件，把这些对象创建出来并存起来。
- 在接下来的使用的时候，直接拿过来用就好了。那么，这个读取配置文件，创建和获取三层对象的类就是工厂。

存哪？

- 由于我们是很多对象，肯定要找个集合来存。这时候有 Map 和 List 供选择。到底选 Map 还是 List 就看我们有没有查找需求。有查找需求，选 Map。
- 在应用加载时，创建一个 Map，用于存放三层对象。我们把这个 map 称之为 容器。

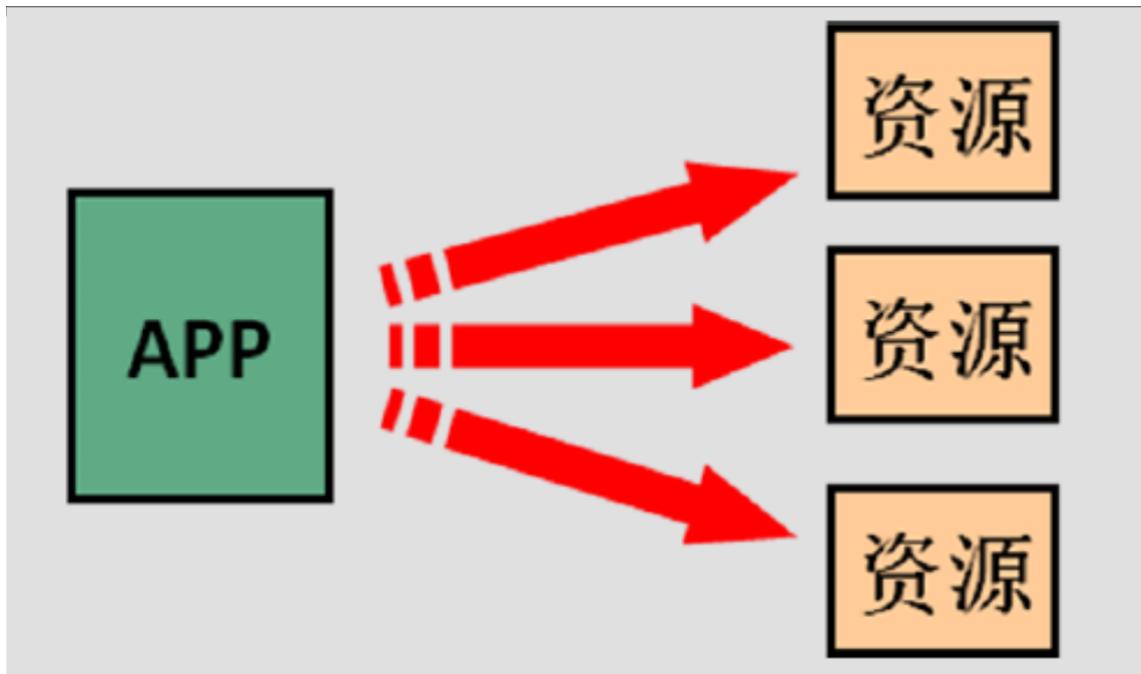
工厂？

- 工厂就是负责给我们从容器中获取指定对象的类。
- 创建和获取三层对象的类。

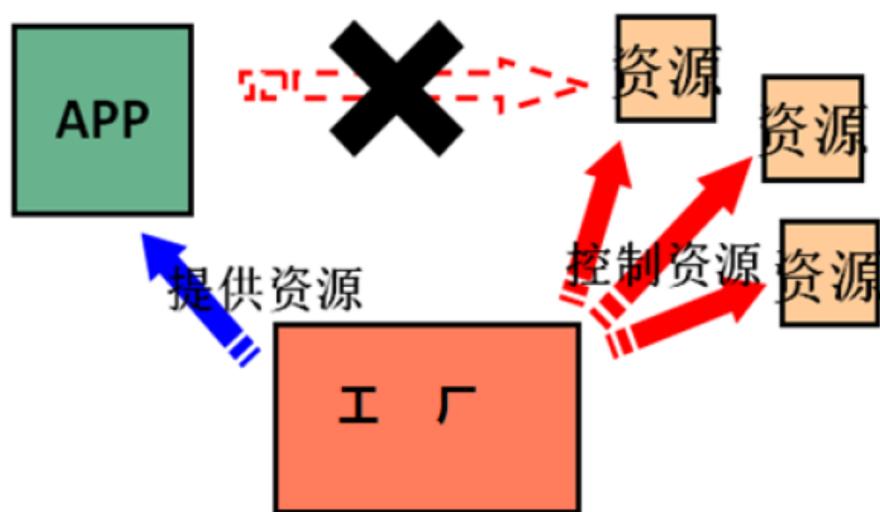
获取对象的方式：

- 原来：我们在获取对象时，都是采用 new 的方式。是主动的。
- 现在：我们获取对象时，同时跟工厂要，有工厂为我们查找或者创建对象。是被动的。

原来



现在



这种被动接收的方式获取对象的思想就是控制反转，它是 spring 框架的核心之一。削减计算机程序的耦合(解除我们代码中的依赖关系)。

代理模式解耦

方法间解耦，AOP。

示例

BeanFactory

增强Service中的方法，这里创建了一个Service的工厂，通过工厂创建实例。

创建方法中，使用Proxy代理方式增强Service中方法。

```
2 * 用于创建Service的代理对象的工厂
3 */
4 public class BeanFactory {
5
6     private AccountService accountService;
7
8     /**
9      * 用于注入
10     * ?为啥加final
11     * @param accountService
12     */
13     public final void setAccountService(AccountService accountService) {
14         this.accountService = accountService;
15     }
16
17     private TransactionManager txManager;
18
19     /**
20      * 需要Spring提供注入，我们提供set方法，这里是xml配置
21      * 注解配置没set也行
22      * @param txManager
23      */
24     public void setTxManager(TransactionManager txManager) {
25         this.txManager = txManager;
26     }
27
28     /**
29      * 本来 获取Service的直接return
30      * 现在 获取Service代理对象
31      * 工厂模式 用该方法提供service, service的方法 被 增强了 ， 包上了厚厚的一层。
32      * @return
33      */
34     public AccountService getAccountService() {
35         return (AccountService)Proxy.newProxyInstance(
36             accountService.getClass().getClassLoader(),
37             accountService.getClass().getInterfaces(),
38             new InvocationHandler() {
39                 /**
40                  * 添加事务支持
41                  * invoke方法具有拦截功能，能拦截 被代理对象 中的所有执行的方法
42                  * @param proxy
43                  * @param method
44                  * @param args
45                  * @return
46                  * @throws Throwable
47                  */
48                  public Object invoke(Object proxy, Method method,
49 Object[] args) throws Throwable {
50                     //消除重复代码 确实， 事务代码一遍完事 全在这里做
51                     //TransactionManager和Service的耦合降低，TX方法该，这里
52                     //改1次， Service改n多次
53                     //多个代码片段公有部分的提取
54                     Object returnval = null;
55                     try {
56                         //1.开启事务
57                         txManager.beginTransaction();
58                         //2.执行操作
59                     } catch (Exception e) {
60                         e.printStackTrace();
61                         returnval = null;
62                     }
63                     return returnval;
64                 }
65             });
66     }
67 }
```

```

57             //List<Account> accounts =
accountDao.findAllAccount();
58                     returnVal = method.invoke(accountService,
args);
59                     //3.提交事务
60                     txManager.commit();
61                     //4.返回结果
62                     return returnVal;
63     } catch (Exception e) {
64                     //5.回滚事务
65                     txManager.rollback();
66                     throw new RuntimeException(e);
67     } finally {
68                     //6.释放连接
69                     txManager.release();
70     }
71 }
72 }
73 );
74 }
75 }

```

bean.xml

```

1 <!-- 配置代理的service对象 -->
2 <!-- 工厂方式创建 -->
3 <bean id="proxyAccountService" factory-bean="beanFactory" factory-
method="getAccountService">
4
5 </bean>
6
7 <!-- 配置 beanFactory 用于事务管理 -->
8 <bean id="beanFactory" class="com.learn.factory.BeanFactory">
9     <!-- 注入service -->
10    <property name="accountService" ref="accountService"></property>
11    <!-- 注入事务管理器 -->
12    <property name="txManager" ref="txManager"></property>
13 </bean>
14
15 <!-- 配置Service -->
16 <bean id="accountService"
17       class="com.learn.service.impl.AccountServiceImpl">
18     <!-- 注入dao:accountService.setXXX() -->
19     <property name="accountDao" ref="accountDao"></property>
20     <!-- 注入事务管理器 事务转移到BeanFactory -->
21     <!--         <property name="txManager" ref="txManager"></property>-->
22 </bean>

```

面向切面编程

AOP : 全称是 Aspect Oriented Programming 即 : 面向切面编程。

Aspects Oriented Programming : 面向切面编程。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是OOP的延续，是函数式编程的一种衍生。

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个部分之间的耦合度降低，提高程序的可重用性，同时提高开发的效率。

作用：

- 在程序运行期间，不修改源码对已有方法进行增强。

优势：

- 减少重复代码
- 提高开发效率
- 维护方便

简单的说它就是把我们程序重复的代码抽取出来，在需要执行的时候，使用动态代理的技术，在不修改源码的基础上，对我们的已有方法进行增强。

- 方法的核心逻辑，往往没那么多。
- 很多的前后处理，是**固定的样板代码**。
- 每次实现某一类方法时均需样板代码包裹，代码重复，耦合度高。
- 从此角度，方法依赖——>方法流|方法的执行顺序。
- 利用代理模式思想，对核心逻辑进行前后包裹。省去前后处理的样板代码。
- 也是关注于方法的执行流程，通过较为基础的一组方法，按序组合成较为高级的方法，实现功能。有织入的说法。

如此，可以关心方法的核心逻辑了，高内聚。

```
1 方法1() {  
2     执行前处理1();  
3     执行前处理2();  
4     执行前处理3();  
5     ...  
6     该方法的执行逻辑();  
7     ...  
8     执行后处理1();  
9     执行后处理2();  
10    执行后处理3();  
11 }  
12 方法2() {...}  
13 方法3() {...}
```

快速入门

手动加载jar包的方式

创建基本的java项目

导入包

- spring-beans-5.2.3.RELEASE.jar
- spring-context-5.2.3.RELEASE.jar
- spring-core-5.2.3.RELEASE.jar
- spring-expression-5.2.3.RELEASE.jar

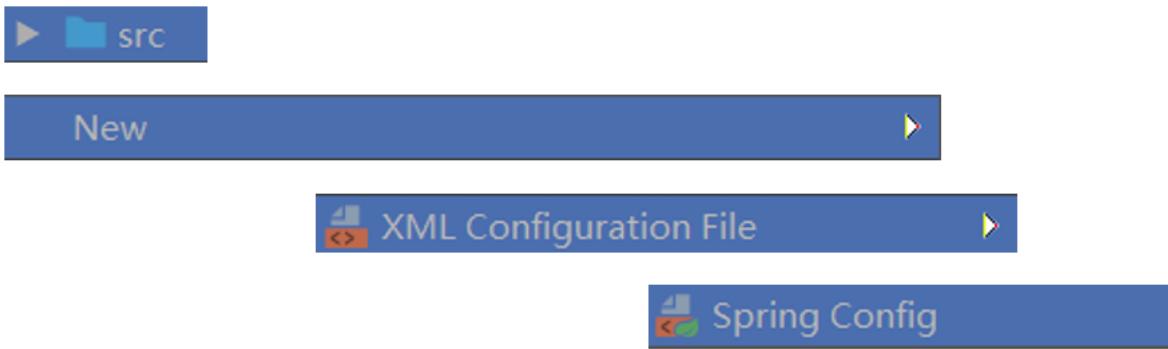
上面几个包依赖下面这个

- commons-logging.jar

<https://mvnrepository.com/>可以方便地找jar包

配置文件

导入jar包后，src下右键 new -> XML Configuration -> Spring Config



```

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   xsi:schemaLocation="http://www.springframework.org/schema/beans
5   http://www.springframework.org/schema/beans/spring-beans.xsd">
6   <!--
7     bean标签表示要创建的对象
8     id:bean的唯一标识,为了跟其他的bean区分
9     class:标识要创建的bean的完全限定名
10    --&gt;
11    &lt;bean id="person" class="com.learn.bean.Person"&gt;
12      <!--
13        给属性赋值使用property标签
14        name:标识属性的名称
15        value:标识具体的属性值
16        --
17        &lt;property name="id" value="1"&gt;&lt;/property&gt;
18        &lt;property name="name" value="张三"&gt;&lt;/property&gt;
19        &lt;property name="age" value="20"&gt;&lt;/property&gt;
20        &lt;property name="gender" value="男"&gt;&lt;/property&gt;
21      &lt;/bean&gt;
22    &lt;/beans&gt;
</pre>

```

java代码

Person

```

1 package com.learn.bean;
2

```

```

3 public class Person {
4     private int id;
5     private String name;
6     private int age;
7     private String gender;
8
9     public Person() {
10         System.out.println("person被创建");
11     }
12
13     //getter setter
14     //toString
15 }
```

Test

ClassPathXmlApplicationContext 和 **FileSystemXmlApplicationContext** 称之为 spring 容器的 **入口**。

- 容器中的对象 在 容器创建完成之前 就已经把对象创建好了(这里的两个容器有歧义)
- ApplicationContext : 表示IOC容器的入口(Spring容器的入口类) , 想要获取对象,必须要创建该类 , 该类有两个读取配置文件的实现类
 - ClassPathXmlApplicationContext:表示从 classpath 中读取数据
 - FileSystemXmlApplicationContext:表示从 当前文件系统 读取数据

```

1 public class MyTest {
2 /**
3 * 容器中的对象 在容器创建完成之前就已经把对象创建好了
4 * @param args
5 */
6 public static void main(String[] args) {
7     /*
8         ApplicationContext:表示IOC容器的入口,想要获取对象,必须要创建该类
9             该类又两个读取配置文件的实现类
10            ClassPathXmlApplicationContext:表示从classpath中读取数据
11            FileSystemXmlApplicationContext:表示从当前文件系统读取数据
12        */
13     ApplicationContext context = new
14     ClassPathXmlApplicationContext("ioc.xml");
15     //从容器中获取具体的bean对象
16     //需要进行强制的类型转换
17     Person person = (Person)context.getBean("person");
18     //不需要强制类型转换
19     Person person1 = context.getBean("person", Person.class);
20     System.out.println(person);
21 }
```

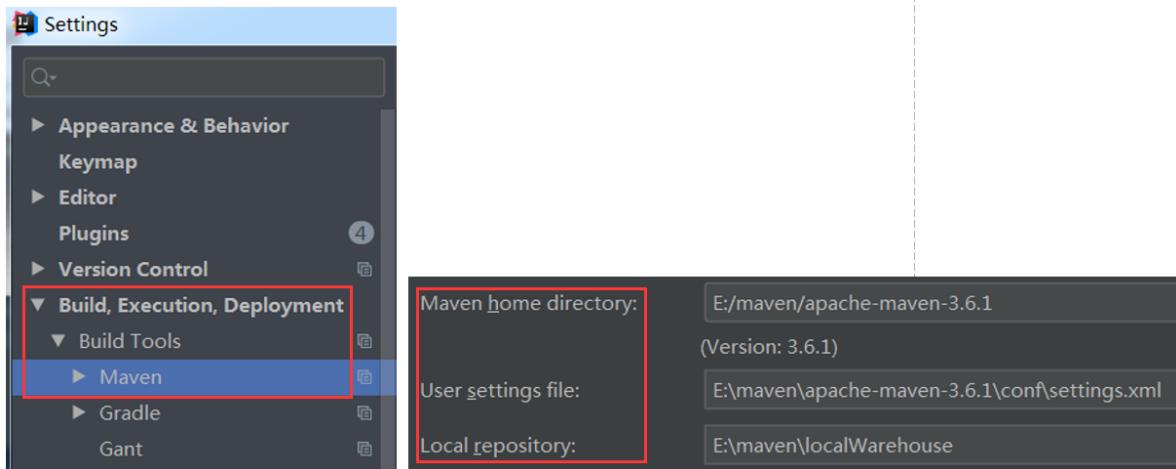
maven方式

使用步骤

- pom context

- xml 配置 : 注册+注入
- java : context.getBean获取对象

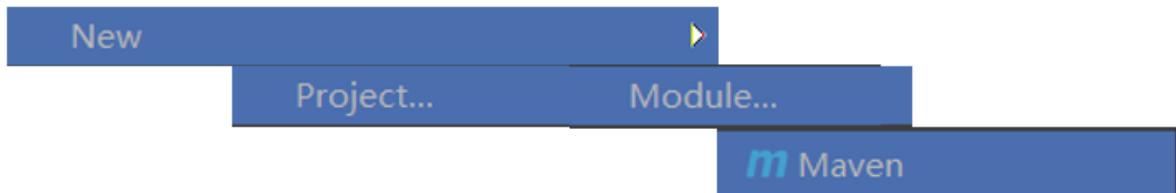
配置maven



Settings->Build,Execution,Deployment->Build Tools->Maven

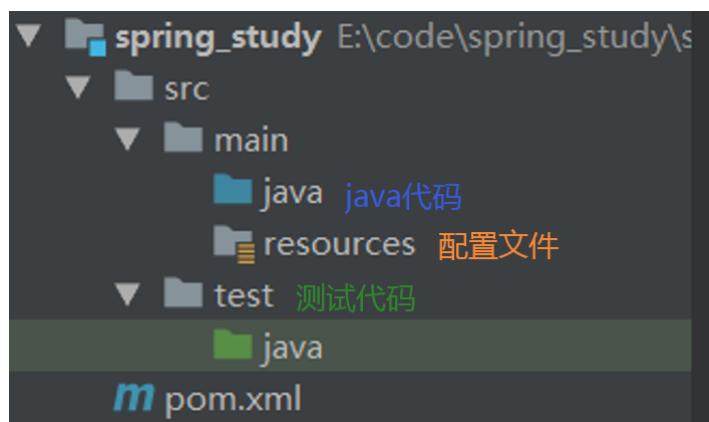
- Maven home directory
- User settings file
- Local repository

创建maven项目



1. new -> project|moudle -> maven即可
2. groupId : 组织名 , 域名倒写
3. ArtifactId : 模块名

项目结构



项目结构：

- main
 - java

- resources : 配置文件
- test

添加对应的pom依赖

- maven仓库搜索依赖
- 复制dependency到项目pom.xml文件中
- maven项目，在resources下添加配置文件。(java项目是src下)

Baidu 百度

[Maven Repository: Search/Browse/Explore](#)

查看此网页的中文翻译，请点击 [翻译此页](#)

What's New in **Maven** Micronaut82 usages io.micronaut » micronaut-runtime » 2.0.0.M3Apache Natively Cloud Native Last Release on Apr 30, 2020...
<https://mvnrepository.com/> - 百度快照

MVNREPOSITORY

1. Spring Context
 org.springframework » spring-context 9,274 usages Apache
 Spring Context
 Last Release on Apr 28, 2020


```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.5.RELEASE</version>
</dependency>
```

4. Spring Web
 org.springframework » spring-web 5,471 usages Apache
 Spring Web
 Last Release on Apr 28, 2020

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.ninthart</groupId>
    <artifactId>spring_study</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.2.5.RELEASE</version>
        </dependency>
    </dependencies>

</project>

```

复制过来即可

pom修改：

- 1 | <packaging>jar</packaging>

- 1 | <dependencies>

2 | <!--

3 | https://mvnrepository.com/artifact/org.springframework/spring-context

-->

4 | <dependency>

5 | <groupId>org.springframework</groupId>

6 | <artifactId>spring-context</artifactId>

7 | <version>5.2.5.RELEASE</version>

8 | </dependency>
 </dependencies>

pom完成：

```

1 | <?xml version="1.0" encoding="UTF-8"?>
2 | <project xmlns="http://maven.apache.org/POM/4.0.0"
3 |           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 |           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5 |               http://maven.apache.org/xsd/maven-4.0.0.xsd">
6 |
7 |     <modelVersion>4.0.0</modelVersion>
8 |
9 |     <groupId>com.ninthart</groupId>
10 |    <artifactId>maven001</artifactId>
11 |    <version>1.0-SNAPSHOT</version>
12 |    <packaging>jar</packaging>
13 |
14 |    <dependencies>
15 |        <!-- https://mvnrepository.com/artifact/org.springframework/spring-
16 |             context -->

```

```
14 <dependency>
15   <groupId>org.springframework</groupId>
16   <artifactId>spring-context</artifactId>
17   <version>5.2.5.RELEASE</version>
18 </dependency>
19 </dependencies>
20
21 </project>
```

只加了一个依赖，但是相关包都导进来了。

配置Spring的xml文件

resources文件夹下新建Spring Config 的xml文件

```
1 accountService=com.learn.service.impl.AccountServiceImpl
2 accountDao=com.learn.dao.impl.AccountDaoImpl
```

```
1 <bean id="accountService" class="com.learn.service.impl.AccountServiceImpl">
2 </bean>
3 <bean id="accountDao" class="com.learn.dao.impl.AccountDaoImpl"></bean>
```

- id : 获取时的唯一标志
- class : 反射要创建对象的全限定类名

从容器获取对象

- 获取根据上面 xml 创建好的容器，然后根据 id 拿出对象即可。
 - 获得spring的 IoC核心容器

```
1 ApplicationContext context = new
2 ClassPathXmlApplicationContext("bean.xml");
```

- 根据id 获得对象

```
1 AccountService as =
2 (AccountService)context.getBean("accountService");
3 AccountDao dao = context.getBean("accountDao",
4 AccountDao.class);
```

总结&问题

以上两种方式创建spring的项目都是可以的，但是在现在的企业开发环境中使用更多的是maven这样的方式，无须自己处理jar之间的依赖关系，也无须提前下载jar包，只需要配置相关的pom即可，因此推荐大家使用maven的方式，具体的maven操作大家可以看maven的详细操作文档。

总结

- ApplicationContext 就是 IOC容器的接口，可以通过此对象 获取容器中创建的对象
- 对象在Spring容器创建完成的时候就已经创建完成，不是需要用的时候才创建
- 对象在IOC容器中存储的时候都是单例的，如果需要多例需要修改属性
- 创建对象给属性赋值的时候是 通过setter方法实现的
- 对象的属性 是由 **setter/getter方法** 决定的，而不是定义的成员属性
- 对象 默认单例

搭建spring项目需要注意的点

- 一定要将配置文件添加到 类路径中，使用idea创建项目的时候要放在resource目录下
- idea的resource 相当于 类路径？
- 导包的时候别忘了commons-logging-1.2.jar包，spring几个相关包依赖该包。手动导包需要注意，maven管理就无需担心了

整合其他单元

整合Junit

测试人员可能不会写拿到context

应用程序的入口：main方法

JUnit单元测试中，没有main方法也能执行：

- JUnit集成了一个main方法
- 该方法就会判断当前测试类中哪些方法有 @Test注解
- JUnit就让有Test注解的方法执行

JUnit不会管我们是否采用Spring框架：

- 在执行测试方法时，JUnit根本不知道我们是不是使用了Spring框架，所以也就不会为我们读取配置文件/配置类创建Spring核心容器

由以上三点可知，**当测试方法执行时，没有Ioc容器，就算写了Autowired注解，也无法事先注入**

注：

- 当我们使用spring 5.x版本的时候，要求JUnit的jar必须是4.12及以上

依赖：

- spring-test
- spring-aop
- JUnit

步骤：

- 导入jar包依赖
- 使用 @RunWith 注解替换原有运行器
- 使用 @ContextConfiguration 执行 spring配置文件的位置
- 使用 @Autowired 给测试类中的变量注入数据

pom

导入spring整合junit的jar坐标

```
1 <dependency>
2   <groupId>org.springframework</groupId>
3   <artifactId>spring-test</artifactId>
4   <version>5.1.6.RELEASE</version>
5 </dependency>
```

@RunWith

使用JUnit提供的一个注解把原有的main方法替换成Spring提供的

```
1 @RunWith(SpringJUnit4ClassRunner.class)
```

@ContextConfiguration

告知Spring的运行器，Spring和loc创建是基于xml还是注解的，并且说明位置

- locations : 指定xml文件的位置，加上classpath关键字，表示在类路径下
- classes : 指定注解类所在的位置

```
1 /**
2  * 使用Junit单元测试,测试我们的配置
3 */
4 @RunWith(SpringJUnit4ClassRunner.class)//创建容器
5 @ContextConfiguration(classes = SpringConfiguration.class)//注解方式
6 @ContextConfiguration(locations = "classpath:bean.xml")//xml方式
7 public class AccountServiceTest {
8
9     @Autowired
10    AccountService service;
11
12    @Test
13    public void testFindAll() {
14        //3.执行方法
15        List<Account> accounts = service.findAllAccount();
16        for (Account account : accounts) {
17            System.out.println(account);
18        }
19    }
20
21    @Test
22    public void testFindOne() {
23        //3.执行方法
24        Account account = service.findAccountById(1);
25        System.out.println(account);
26    }
27}
```

```

28     @Test
29     public void testSave() {
30         //3.执行方法
31         Account account = new Account(null, "呵呵", 1000F);
32         service.saveAccount(account);
33     }
34
35     @Test
36     public void testUpdate() {
37         //3.执行方法
38         Account account = service.findAccountById(3);
39         account.setMoney(10000F);
40         service.updateAccount(account);
41     }
42
43     @Test
44     public void testDelete() {
45         //3.执行方法
46         service.deleteAccount(4);
47     }
48 }
```

问题

为什么不把测试类配到XML中？

在解释这个问题之前，先解除大家的疑虑，配到 XML 中能不能用呢？

答案是肯定的，没问题，可以使用。

那么为什么不采用配置到 xml 中的方式呢？

这个原因是这样的：

第一：当我们在 xml 中配置了一个 bean，spring 加载配置文件创建容器时，就会创建对象。

第二：测试类只是我们在测试功能时使用，而在项目中它并不参与程序逻辑，也不会解决需求上的问题，所以创建完了，并没有使用。那么存在容器中就会造成资源的浪费。

所以，基于以上两点，我们不应该把测试配置到 xml 文件中。

环境搭建

- 单独使用，maven下简单工程

主要2文件

- Spring配置文件。我一般叫 bean.xml
- 测试类 使用

注册对象。
交由容器管理

SpringConfig配置信息

XML配置文件

bean.xml

配置类

@Configuration
SpringConfiguration

或

使用对象。
从容器中获取

```
//1. 获取容器
ApplicationContext context =
    new ClassPathXmlApplicationContext("bean.xml");
//2. 得到业务层对象
AccountService service =
    context.getBean("accountService", AccountService.class);
//3. 执行方法
Account account = service.findAccountById(1);
```

或

```
@Autowired
private AccountDao accountDao;
```

新建maven工程

pom 文件依赖

Spring

- spring-beans-5.2.3.RELEASE.jar
- spring-context-5.2.3.RELEASE.jar
- spring-core-5.2.3.RELEASE.jar
- spring-expression-5.2.3.RELEASE.jar
- 上面几个包依赖下面这个commons-logging.jar

pom文件一个 spring-context依赖即可。包含上面的jar包

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5         http://maven.apache.org/xsd/maven-4.0.0.xsd">
6     <modelVersion>4.0.0</modelVersion>
```

```
6 <groupId>com.ninthart</groupId>
7 <artifactId>maven001</artifactId>
8 <version>1.0-SNAPSHOT</version>
9 <packaging>jar</packaging>
10
11 <dependencies>
12     <!-- https://mvnrepository.com/artifact/org.springframework/spring-
13 context -->
14     <dependency>
15         <groupId>org.springframework</groupId>
16         <artifactId>spring-context</artifactId>
17         <version>5.2.5.RELEASE</version>
18     </dependency>
19 </dependencies>
20
21 </project>
```

数据源

- mysql-connector-java 必需，数据库驱动

```
1 <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
2 <dependency>
3     <groupId>mysql</groupId>
4     <artifactId>mysql-connector-java</artifactId>
5     <version>5.1.47</version>
6 </dependency>
```

druid

```
1 <dependency>
2     <groupId>com.alibaba</groupId>
3     <artifactId>druid</artifactId>
4     <version>1.1.21</version>
5 </dependency>
```

C3P0

- c3p0-xxx.jar

```
1 |
```

DBCP

- commons-dbc.jar
- commons-pool.jar

```
1 |
```

DriverManagerDataSource

- Spring内置数据源
- 无需额外依赖

JdbcTemplate

- spring-jdbc-xxx.jar

事务

- spring-tx-xxx.jar
- (aop : aspectjweaver)
- (spring-jdbc)
- mysql-connector-java

注解

- spring-aop-xxx.jar

AOP

- aspectjweaver : 解析切入点表达式

```
1 <!-- 解析切入点表达式 -->
2 <dependency>
3   <groupId>org.aspectj</groupId>
4   <artifactId>aspectjweaver</artifactId>
5   <version>1.8.2</version>
6 </dependency>
```

Spring配置信息

xml配置文件

位置

- resources/

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5
6   xsi:schemaLocation="http://www.springframework.org/schema/beans
7       http://www.springframework.org/schema/beans/spring-beans.xsd">
8
9 </beans>
```

名称空间&约束

基本spring约束

- 只有beans而已

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5
6     xsi:schemaLocation="http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5
6     xsi:schemaLocation="http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-
beans.xsd">
8     <!--
9         bean标签标识要创建的对象
10        id:bean的唯一标识,为了跟其他的bean区分
11        class:标识要创建的bean的完全限定名
12    -->
13    <bean id="person" class="com.learn.bean.Person">
14        <!--
15            给属性赋值使用property标签
16            name:标识属性的名称
17            value:标识具体的属性值
18        -->
19        <property name="id" value="1"></property>
20        <property name="name" value="张三"></property>
21        <property name="age" value="20"></property>
22        <property name="gender" value="男"></property>
23    </bean>
24 </beans>
```

p名称空间

本质还是调用set方法。

此种方式是通过在 xml中导入 p名称空间，使用 p:propertyName 来注入数据，它的本质仍然是调用类中的 set 方法实现注入功能。

```
1 beans里加
2 <beans xmlns:p="http://www.springframework.org/schema/p">
3   <bean id="person5" class="com.learn.bean.Person"
4     p:id="5"
5     p:name="王五"
6     p:age="25"
7     p:gender="女"
8   ></bean>
9 </beans>
```

context名称空间

加载外部文件 需要 context名称空间

- xmlns:context=
 - "xxx/context"
- xsi:schemaLocation=
 - "xxx/context"
 - "xxx/context/spring-context.xsd"

(context 加载外部依赖文件)

```
1 username=root
2 password=123456
3 url=jdbc:mysql://localhost:3306/demo
4 driverClassName=com.mysql.jdbc.Driver
5
6 jdbc.username 加前缀 解决 同名问题
```

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5
6   xmlns:context="http://www.springframework.org/schema/context"
7
8   xsi:schemaLocation="http://www.springframework.org/schema/beans
9     http://www.springframework.org/schema/beans/spring-beans.xsd
10
11     http://www.springframework.org/schema/context
12     http://www.springframework.org/schema/context/spring-context.xsd">
13   <!--
14     加载外部配置文件
15     在加载外部依赖文件的时候需要context命名空间
16   -->
17   <context:property-placeholder
18     location="classpath:dbconfig.properties"/>
19
20   <bean id="dataSource2" class="com.alibaba.druid.pool.DruidDataSource">
21     <property name="username" value="${username}"></property>
22     <property name="password" value="${password}"></property>
23     <property name="url" value="${url}"></property>
```

```
23     <property name="driverClassName" value="${driverClassName}">
24   </property>
25 </bean>
26 </beans>
```

或

```
1 <bean
2   class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
3   >
4     <property name="location" value="classpath:jdbc.properties"/>
5   </bean>
```

(开启注解扫描)

- context : 开注解

```
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
6   xsi:schemaLocation="http://www.springframework.org/schema/beans
7   http://www.springframework.org/schema/beans/spring-beans.xsd
8
9   http://www.springframework.org/schema/context
10  http://www.springframework.org/schema/context/spring-context.xsd">
11  <context:component-scan base-package="com.learn">会扫描该包及其子包所有类与
12  接口上的注解</context:component-scan>
13 </beans>
14
15 <?xml version="1.0" encoding="UTF-8"?>
16 <beans xmlns="http://www.springframework.org/schema/beans"
17   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
18   xmlns:context="http://www.springframework.org/schema/context"
19   xsi:schemaLocation="http://www.springframework.org/schema/beans
20   http://www.springframework.org/schema/beans/spring-beans.xsd
21   http://www.springframework.org/schema/context
22   http://www.springframework.org/schema/context/spring-context.xsd">
23
24  <!--指定只扫描哪些组件， 默认情况下是全部扫描的， 所以此时要配置的话需要在
25  component-scan标签中添加 use-default-filters="false"-->
26  <context:component-scan base-package="com.learn" use-default-
27  filters="false">
28    <!-- 包含注解 -->
29    <context:exclude-filter type="annotation"
30      expression="org.springframework.stereotype.Controller"/>
31
32    <!-- 排除类 -->
33    <context:include-filter type="assignable"
34      expression="com.learn.service.PersonService"/>
35  </context:component-scan>
36 </beans>
```

aop约束

```
1 <beans xmlns="http://www.springframework.org/schema/beans"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3
4   xmlns:aop="http://www.springframework.org/schema/aop"
5
6   xsi:schemaLocation="http://www.springframework.org/schema/beans
7     http://www.springframework.org/schema/beans/spring-beans.xsd
8
9   http://www.springframework.org/schema/aop
10    http://www.springframework.org/schema/aop/spring-aop.xsd
11  ">
```

(开启aop注解)

- context : 开注解
- aop : 开aop注解

```
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
5   xmlns:aop="http://www.springframework.org/schema/aop"
6   xmlns:context="http://www.springframework.org/schema/context"
7
8   xsi:schemaLocation="
9     http://www.springframework.org/schema/beans
10    http://www.springframework.org/schema/beans/spring-beans.xsd
11    http://www.springframework.org/schema/aop
12    http://www.springframework.org/schema/aop/spring-aop.xsd
13    http://www.springframework.org/schema/context
14    http://www.springframework.org/schema/context/spring-context.xsd">
15
16  <!-- 配置Spring创建容器时要扫描的包 支持注解 -->
17  <context:component-scan base-package="com.learn"></context:component-
18 scan>
19
20  <!-- 配置Spring开启注解AOP的支持 -->
21  <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
22 </beans>
```

事务

- aop
- tx

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

```

5      xmlns:tx="http://www.springframework.org/schema/tx"
6
7      xsi:schemaLocation="http://www.springframework.org/schema/beans
8          http://www.springframework.org/schema/beans/spring-beans.xsd
9              http://www.springframework.org/schema/aop
10             http://www.springframework.org/schema/aop/spring-aop.xsd
11                 http://www.springframework.org/schema/tx
12                     http://www.springframework.org/schema/tx/spring-
13 tx.xsd">
14
15 <!-- 配置spring创建容器时要扫描的包-->
16 <context:component-scan base-package="com.learn"></context:component-scan>
17
18 <!-- 配置事务管理器 -->
19 <bean id="transactionManager"
20     class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
21     <property name="dataSource" ref="dataSource"></property>
22 </bean>
23
24 <!-- 开启spring对注解事务的支持-->
25 <tx:annotation-driven transaction-manager="transactionManager">
26 </tx:annotation-driven>

```

标签

<bean>

- id
- class
- factory-bean
- factory-method
- scope : singleton、prototype、request、session、global session
- init-method
- destroy-method
- abstract
- parent
- depend-on
- autowire : default/no、byName、byType、constructor

<property> : setXxx , 注入

```

1 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
2     <property name="username" value="root"></property>
3     <property name="password" value="123456"></property>
4     <property name="url" value="jdbc:mysql://localhost:3306/demo"></property>
5     <property name="driverClassName" value="com.mysql.jdbc.Driver">
6     </property>
</bean>

```

<constructor-arg> : 方法传参 构建 , 常为 构造方法

```

1 <bean id="accountService" class="com.learn.service.impl.AccountServiceImpl">
2   <constructor-arg name="name" value="test"></constructor-arg>
3   <constructor-arg name="age" value="18"></constructor-arg>
4   <constructor-arg name="birthday" ref="now"></constructor-arg>
5 </bean>
6
7 <!-- 配置一个日期对象:Date now = new Date(); -->
8 <bean id="now" class="java.util.Date"></bean>

```

<property>

set方法注入属性

- name : 用于指定注入时所调用的set方法名称；找的是类中 set 方法后面的部分
- value : 用于提供 基本类型 和 String类型 的数据
- ref : 用于指定其他的bean类型数据。它指的就是在sprng的loc容器中出现过的bean对象

```

1 <bean id="person4" class="com.mashibing.bean.Person">
2   <!--支持任何运算符-->
3   <property name="age" value="#{12*2}"></property>
4   <!--可以引用其他bean的某个属性值-->
5   <property name="name" value="#{address.province}"></property>
6   <!--引用其他bean-->
7   <property name="address" value="#{address}"></property>
8   <!--调用静态方法-->
9   <property name="hobbies" value="#
{T(java.util.UUID).randomUUID().toString().substring(0,4)}"></property>
10  <!--调用非静态方法-->
11  <property name="gender" value="#{address.getCity()}"></property>
12 </bean>

```

<constructor-arg>

(构造)方法注入属性

- name : 用于指定注入时所调用的set方法名称；找的是类中 set 方法后面的部分
- value : 用于提供 基本类型 和 String类型 的数据
- ref : 用于指定其他的bean类型数据。它指的就是在sprng的loc容器中出现过的bean对象

- type : 指定参数在构造函数中的数据类型
- index : 指定参数在构造函数参数列表的索引位置。参数索引：0 , 1 , 2...

<array>

```

1 <bean id="accountService2"
2   class="com.learn.service.impl.AccountServiceImpl">
3     <property name="myStrs">
4       <array>
5         <value>AAA</value>
6         <value>BBB</value>
7         <value>CCC</value>

```

```
7     </array>
8 </property>
9
10    <property name="age" value="21"></property>
11    <property name="birthday" ref="now"></property>
12 </bean>
13 <!-- 配置一个日期对象:Date now = new Date(); -->
14 <bean id="now" class="java.util.Date"></bean>
```

<list>

```
1 <property name="myList">
2   <list>
3     <value>AAA</value>
4     <value>BBB</value>
5     <value>CCC</value>
6   </list>
7 </property>
```

<map>

<entry>

```
1 <property name="myMap">
2   <map>
3     <entry key="A" value="aaa"></entry>
4     <entry key="B" value="bbb"></entry>
5     <entry key="C">ccc</entry>
6   </map>
7 </property>
```

<entry>

<set>

<value>

```
1 <property name="mySet">
2   <set>
3     <value>AAA</value>
4     <value>BBB</value>
5     <value>CCC</value>
6   </set>
7 </property>
```

<value>

```
<props>
```

```
<prop>
```

```
1 <property name="myProps">
2   <props>
3     <prop key="A">aaa</prop>
4     <prop key="B">bbb</prop>
5     <prop key="C">ccc</prop>
6   </props>
7 </property>
```

```
<prop>
```

```
<context:property-placeholder>
```

可以引入外部properties文件

```
<context:component-scan>
```

扫描包

```
<aop:config>
```

表明开始AOP的配置

```
<aop:advisor>
```

```
1 <!-- 配置AOP -->
2 <aop:config>
3   <!-- 配置切入点表达式
4     在aop:aspect标签内部 只能在当前切面使用
5     它还可以卸载aop:aspect外面，此时就变成了所有切面可用，必须写在切面前
6   -->
7   <aop:pointcut id="pt2" expression="execution(public *
com.learn.service.impl.*.*(..))"/>
8   <!-- 配置切面 引用通知 -->
9   <aop:aspect id="logAdvice" ref="logger">
10    <!-- 前置通知：在切入点方法执行之前执行 -->
11    <aop:before method="beforePrintLog" pointcut="execution(public *
com.learn.service.impl.*.*(..))"></aop:before>
12    <!-- 后置通知：在切入点方法正常执行之后执行，它和后置通知永远只能执行一个 -->
13    <aop:after-returning method="afterReturnngPrintLog"
pointcut="execution(public * com.learn.service.impl.*.*(..))"></aop:after-
returning>
14    <!-- 异常通知：在切入点方法执行产生异常之后执行 -->
15    <aop:after-throwing method="afterThrowingPrintLog" pointcut-
ref="pt2"></aop:after-throwing>
16    <!-- 最终通知：无论切入点方法是否正常执行它都会在其后面执行 -->
17    <aop:after method="afterPrintLog" pointcut-ref="pt1"></aop:after>
18
19   <!-- 配置切入点表达式
20     在aop:aspect标签内部 只能在当前切面使用
```

```
21          它还可以卸载aop:aspect外面，此时就变成了所有切面可用，但必须写在切面  
22          前  
23          -->  
24          <aop:pointcut id="pt1" expression="execution(public *  
25          com.learn.service.impl.*.*(..))"/>  
26          <!-- 详细注释 情况logger类中 -->  
27          <aop:around method="aroundPrintLog" pointcut-ref="pt1">  
28      </aop:around>  
29      </aop:aspect>  
30  </aop:config>
```

<aop:advisor>

表明配置切面

- id : 给切面提供一个唯一标识
- ref : 指定 **通知类bean** 的id

<aop:pointcut>

切入点

位置：

- 在aop:aspect标签内部，只能在当前切面使用
- 在aop:aspect标签外面，所有切面可用，必须写在切面前

属性：

- id
- expression 切入点表达式

</tx:advice>

- id
- transaction-manager

```
1  <!-- 配置事务的通知 -->  
2  <tx:advice id="txAdvice" transaction-manager="transactionManager">  
3      <!-- 配置事务的属性 -->  
4      <tx:attributes>  
5          <tx:method name="*" propagation="REQUIRED" read-only="false"/>  
6          <tx:method name="find*" propagation="SUPPORTS" read-only="true">  
7      </tx:method>  
8      </tx:attributes>  
9  </tx:advice>
```

```
<tx:attributes>
```

```
1 <!-- 配置事务的通知 -->
2 <tx:advice id="txAdvice" transaction-manager="transactionManager">
3     <!-- 配置事务的属性 -->
4     <tx:attributes>
5         <tx:method name="*" propagation="REQUIRED" read-only="false"/>
6         <tx:method name="find*" propagation="SUPPORTS" read-only="true">
7     </tx:method>
8     </tx:attributes>
9 </tx:advice>
```

```
<tx:method>
```

- name
- propagation
- read-only

```
1 <!-- 配置事务的通知 -->
2 <tx:advice id="txAdvice" transaction-manager="transactionManager">
3     <!-- 配置事务的属性 -->
4     <tx:attributes>
5         <tx:method name="*" propagation="REQUIRED" read-only="false"/>
6         <tx:method name="find*" propagation="SUPPORTS" read-only="true">
7     </tx:method>
8     </tx:attributes>
9 </tx:advice>
```

配置类(注解方式)

基本配置类

```
1 @Configuration
2 @ComponentScan("com.learn")//扫描包
3 public class SpringConfiguration {
4 }
```

AOP支持配置类

```
1 @Configuration
2 @ComponentScan(basePackages="com.learn")
3 @EnableAspectJAutoProxy//aop注解支持
4 public class SpringConfiguration {
5 }
```

事务配置类

- SpringConfiguration
- JdbcConfig
 - jdbcConfig.properties
- TransactionConfig

SpringConfiguration

```
1 /**
2  * Spring的配置类
3  * 相当于bean.xml
4 */
5 @Configuration
6 @ComponentScan("com.learn")
7 @Import({JdbcConfig.class, TransactionConfig.class})
8 @PropertySource("jdbcConfig.properties")
9 @EnableTransactionManagement//事务注解支持
10 public class SpringConfiguration {
11 }
```

TransactionConfig

```
1 /**
2  * 和事务相关的配置类
3 */
4 public class TransactionConfig {
5 /**
6  * 用于创建事务管理器对象
7  * @param dataSource
8  * @return
9 */
10 @Bean(name = "transactionManager")
11 public PlatformTransactionManager createTransactionManager(DataSource
dataSource) {
12     return new DataSourceTransactionManager(dataSource);
13 }
14 }
```

JdbcConfig

```
1 /**
2  * 和连接数据库相关的配置类
3 */
4 public class JdbcConfig {
5
6     @Value("${jdbc.driver}")
7     private String driver;
8
9     @Value("${jdbc.url}")
10    private String url;
11
12    @Value("${jdbc.username}")
13    private String username;
14
15    @Value("${jdbc.password}")
```

```

16     private String password;
17
18     /**
19      * 创建JdbcTemplate对象
20      * @param dataSource
21      * @return
22      */
23     @Bean(name = "jdbcTemplate")
24     public JdbcTemplate createJdbcTemplate(DataSource dataSource) {
25         return new JdbcTemplate(dataSource);
26     }
27
28     @Bean(name = "dataSource")
29     public DataSource createDataSource() {
30         DriverManagerDataSource ds = new DriverManagerDataSource();
31         ds.setDriverClassName(driver);
32         ds.setUrl(url);
33         ds.setUsername(username);
34         ds.setPassword(password);
35         return ds;
36     }
37 }
```

jdbcConfig.properties

```

1 jdbc.driver=com.mysql.cj.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/db
3 jdbc.username=root
4 jdbc.password=xxxx
```

注解

@Component

用于把当前类对象存入spring容器中。

- value : 用于指定bean的id , 当我们不写时 , 它的默认值是当前类名 , 且首字母改小写 ;
`@Componenet(value="id")` value可省略

@Controller

@Service

@Repository

@Autowired

位置：

- 变量上
- 方法上
 - 当@.Autowired添加到方法上的时候，此方法在创建对象的时候会默认调用。
 - 同时方法中的参数会进行自动装配。
 - @Qualifier注解也可以定义在方法的参数列表中，用于指定当前属性的id名称。

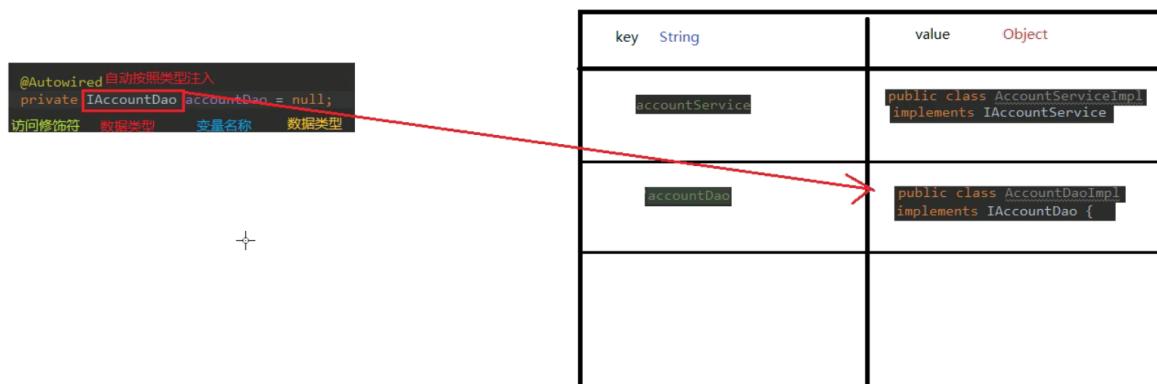
作用：

- 自动按照类型注入。只要容器中有唯一的一个bean对象类型和要注入的变量类型匹配，就可以注入成功
 1. 如果loc容器中没有任何bean的类型和要注入的变量类型匹配，则报错。**类型+唯一bean**
 2. 如果loc容器中有多个类型匹配时，转为id匹配。**对象名 == bean.id**

细节：

- 它只能注入bean类型
- 在使用注解注入时，**set方法就不是必须的了**。没有也能给该属性赋值，自动按照类型注入(都注册到loc容器里，匹配上就...不需要方法了)

Spring的IOC容器：Map结构



当使用@Autowired注解的时候，自动装配的时候默认是根据类型实现的：

1. 如果只找到一个，则直接进行赋值，
2. 如果没有找到，则直接抛出异常，
3. 如果找到多个，那么会按照变量名作为id继续匹配。
 1. 匹配上直接进行装配
 2. 如果匹配不上则直接报异常
4. 如果想通过名字进行查找，可以自己规定名称，使用注解@Qualifier来指定id，让spring不要使用变量名去匹配id：
 1. 找到，直接装配
 2. 找不到，报错

@Qualifier

作用：

- 在按照类型注入的基础之上再按照名称注入。
 - 在给类成员单独注入时，不能单独使用；需和@.Autowired结合使用
 - 在给方法参数注入时，可以单独使用。定义在方法的参数列表中，用于指定当前属性的id名称。

属性：

- value：用于指定注入bean的id

```

1 @Autowired
2 @Qualifier("accountDao1")
3 private AccountDao accountDao = null;

```

```

1 @Bean(name="runner")
2 public QueryRunner createQueryRunner(@Qualifier("ds1") DataSource dataSource)
{
3     return new QueryRunner(dataSource);
4 }
5
6 (@Qualifier("ds1") DataSource dataSource)事实上这里是先autowired的，不匹配才会转
7 qualifier

```

@Resource

作用：

- 直接按照bean的id注入，它可以独立使用
- 只能注入其他bean类型。

属性：

- name：用于指定bean的id

使用@Resource可以完成跟@Autowired相同的功能，但是要注意他们之间的区别：

- @Resource是jdk提供的功能，@Autowired是spring提供的功能
- @Resource可以在其他框架中使用，而@Autowired只能在spring中使用
- @Resource扩展性好，而@Autowired支持的框架比较单一
- @Resource是按照名称进行装配的(先名称，后类型)，而@Autowired是按照类型装配(先类型，后名称)

- @AutoWired:是spring中提供的注解，@Resource:是jdk中定义的注解，依靠的是java的标准
- @AutoWired默认是按照类型进行装配，默认情况下要求依赖的对象必须存在，@Resource默认是按照名字进行匹配的，同时可以指定name属性。
- @AutoWired只适合spring框架，而@Resource扩展性更好

```

1 @Resource(name = "accountDao1")
2 private AccountDao accountDao = null;

```

@Vaule

作用：用于注入 基本类型 和 String 类型 的数据。

属性：

- value : 用于指定数据的值。它可以使用spring中的**SpEL**(Spring中的EL表达式\${表达式})

jdbcConfig.properties

```
1 | jdbc.driver=com.mysql.cj.jdbc.Driver
2 | jdbc.url=jdbc:mysql://localhost:3306/db
3 | jdbc.username=root
4 | jdbc.password=xxxx
```

```
1 | @PropertySource("jdbcConfig.properties")
2 | public class SpringConfiguration {}  

3 |
4 |
5 | public class JdbcConfig {
6 |     //这里用的是Spring的EL表达式，需要在主配置类上
7 |     @PropertySource("jdbcConfig.properties")
8 |         @Value("${jdbc.driver}")
9 |         private String driver;
10 |         @Value("${jdbc.url}")
11 |         private String url;
12 |         @Value("${jdbc.username}")
13 |         private String username;
14 |         @Value("${jdbc.password}")
15 |         private String password;
```

@Scope

作用：指定bean的作用范围

属性：

- value : 指定范围的取值：
 - singleton 单例，默认
 - prototype 多例
 - ...不常用

@PostConstruct

作用：用于指定初始化方法

```
1 @PostConstruct  
2 public void init() {  
3     System.out.println("初始化");  
4 }
```

@PreDestroy

作用：用于指定销毁方法

注意：多例对象的销毁 不是 spring负责的，由jvm做垃圾回收。该注解应和单例对象配合使用

```
1 @PreDestroy  
2 public void destroy() {  
3     System.out.println("销毁");  
4 }
```

@Configuration

指定当前类是一个配置类，完成springconfig.xml的功能

细节：

- 当配置类作为AnnotationConfigApplicationContext对象创建的参数时，该注解可以不写。
尽量不要省略

```
1 ApplicationContext context =  
2 new AnnotationConfigApplicationContext(SpringConfiguration.class);
```

```
1 @Configuration  
2 public class SpringConfiguration {  
3  
4 }
```

@ComponentScan

用于通过注解指定spring在创建容器时要扫描的包

属性：

- value：和basePackages的作用是一样的，都是用于指定创建容器时要扫描的包。

○ 1 | @ComponentScan(basePackages = {"com.learn", "com.config"})

○ 1 | <context:component-scan base-package="com.learn">
</context:component-scan>

```
1 @Configuration  
2 @ComponentScan(basePackages = "com.learn")  
3 public class SpringConfiguration {  
4  
5 }
```

@Bean

作用：

- 该注解 **只能写在方法上**，表明 使用此方法 创建一个对象，并且放入 spring 容器。
- 可以说是专门用于jar包中class准备的。**

属性：

- name：用于指定bean的id。当不写时，默认值是当前方法的名称

细节：

- 方法参数：

- 当我们使用注解配置方法时，如果方法有**参数**，Spring框架会去容器中查找有没有可用的bean对象。
 - 查找的方式和Autowired注解是一样的：先类型，后id。
- 也可以用@Qualifier指定id

```
1 @Configuration  
2 @ComponentScan(basePackages = "com.learn")  
3 public class SpringConfiguration {  
4  
5     @Bean(name="runner")  
6     public QueryRunner createQueryRunner(@Qualifier("ds1") DataSource  
dataSource) {  
7         return new QueryRunner(dataSource);  
8     }  
9  
10    @Bean(name = "ds1")  
11    public DataSource createDataSource() {  
12        try {  
13            ComboPooledDataSource ds = new ComboPooledDataSource();  
14            ds.setDriverClass("com.mysql.cj.jdbc.Driver");  
15            ds.setJdbcUrl("jdbc:mysql://localhost:3306/db");  
16            ds.setUser("root");  
17            ds.setPassword("xxxx");  
18            return ds;  
19        } catch (Exception e) {  
20            throw new RuntimeException(e);  
21        }  
22    }  
23 }
```

@Import

用于导入其他的配置类。多个小的配置类，扫描路径配置困难。(把其他的配置类导到主配置类)这时，对应被导入的配置类就可以省略@Configuration了

属性：

- value：用于指定其他配置类的字节码。当我们使用Import的注解的时候，有Import注解的类就是父配置类，而导入的都是子配置类

```
1 @Configuration
2 @ComponentScan("com.learn")
3 @Import(JdbcConfig.class)//JdbcConfig不用@Configuration了
4 public class SpringConfig {}
5 //这里主配置类SpringConfig和JdbcConfig在同一目录
6 //主配置类SpringConfig 子配置类JdbcConfig
7
8 @Configuration//写不写都行
9 @PropertySource("classpath:jdbc.properties")
10 public class JdbcConfig{
11 }
```

@PropertySource

用于加载.properties 文件中的配置。

属性：用于指定properties文件的位置

- value：指定文件的名称和路径
 - classpath，表示在类路径下

```
1 @ComponentScan("com.learn")
2 @Import(JdbcConfig.class)
3 @PropertySource("classpath:jdbcConfig.properties")
4 public class SpringConfig {}
5 //这里主配置类SpringConfig和JdbcConfig在同一目录
6 //主配置类SpringConfig 子配置类JdbcConfig
```

@Aspect

声明当前类是一个切面类。

```
1 @Component("Logger")
2 @Aspect
3 public class Logger {}
```

@Pointcut

切入点声明。

```
1 @Pointcut("execution(public * com.learn.service.impl.*.*(..))")
2 private void pt1() {}
```

@Before

前置通知

```
1 @Before("pt1()")
2 public void beforePrintLog() {
3     System.out.println("前置通知 Logger类中的before print Log方法开始记录日志
4     了...");
```

@AfterReturning

后置通知

```
1 @AfterReturning("pt1()")
2 public void afterReturnngPrintLog() {
3     System.out.println("后置通知 Logger类中的after returning print Log方法开始记
4     录日志了...");
```

@AfterThrowing

异常通知

```
1 @AfterThrowing("pt1()")
2 public void afterThrowingPrintLog() {
3     System.out.println("异常通知 Logger类中的after throw print Log方法开始记录日
4     志了...");
```

@After

最终通知。finally

```
1 @After("pt1()")
2 public void afterPrintLog() {
3     System.out.println("最终通知 Logger类中的after print Log方法开始记录日志
4     了...");
```

@Around

环绕通知

```
1 @Around("pt1()")
2 public Object aroundPrintLog(ProceedingJoinPoint pjp) {
3     Object rtValue = null;
```

```

4     try {
5         System.out.println("环绕通知 前置 Logger类中的around print Log方法开始
6         记录日志了...");
7         Object[] args = pjp.getArgs();//得到方法执行所需的参数
8         rtValue = pjp.proceed(args);//明确调用业务层方法
9         System.out.println("环绕通知 后置 Logger类中的around print Log方法开始
10        记录日志了...");
11        return rtValue;
12    } catch (Throwable throwable) {
13        System.out.println("环绕通知 异常 Logger类中的around print Log方法开始
14        记录日志了...");
15        throw new RuntimeException(throwable);
16    } finally {
17        System.out.println("环绕通知 最终 Logger类中的around print Log方法开始
18        记录日志了...");
19    }
20 }
```

@EnableTransactionManagement

事务注解支持

```

1 /**
2 * Spring的配置类
3 * 相当于bean.xml
4 */
5 @Configuration
6 @ComponentScan("com.learn")
7 @Import({JdbcConfig.class, TransactionConfig.class})
8 @PropertySource("jdbcConfig.properties")
9 @EnableTransactionManagement//事务注解支持
10 public class SpringConfiguration {
11 }
```

@Bean

```

1 /**
2 * 和事务相关的配置类
3 */
4 public class TransactionConfig {
5 /**
6     * 用于创建事务管理器对象
7     * @param dataSource
8     * @return
9     */
10    @Bean(name = "transactionManager")
11    public PlatformTransactionManager createTransactionManager(DataSource
12 dataSource) {
13        return new DataSourceTransactionManager(dataSource);
14    }
15 }
```

@Transactional

该注解的属性和 xml 中的属性含义一致。该注解可以出现在接口上，类上和方法上。

- 出现接口上，表示该接口的所有实现类都有事务支持。
- 出现在类上，表示类中所有方法有事务支持
- 出现在方法上，表示方法有事务支持。
- 以上三个位置的优先级：方法>类>接口

```
1 @Service("accountService")
2 @Transactional(propagation= Propagation.SUPPORTS, readOnly=true) //只读型事务的
3     配置
4 public class AccountServiceImpl implements AccountService {
5
6     @Autowired
7     private AccountDao accountDao;
8
9     //只读型事务的配置
10    public Account findAccountById(Integer accountId) {
11        return accountDao.findAccountById(accountId);
12    }
13
14    //需要的是读写型事务配置
15    @Transactional(propagation= Propagation.REQUIRED, readOnly=false)
16    public void transfer(String sourceName, String targetName, Float money)
17    {
18        System.out.println("transfer....");
19        //2.1根据名称查询转出账户
20        Account source = accountDao.findAccountByName(sourceName);
21        //2.2根据名称查询转入账户
22        Account target = accountDao.findAccountByName(targetName);
23        //2.3转出账户减钱
24        source.setMoney(source.getMoney() - money);
25        //2.4转入账户加钱
26        target.setMoney(target.getMoney() + money);
27        //2.5更新转出账户
28        accountDao.updateAccount(source);
29
30        int i=1/0;
31
32        //2.6更新转入账户
33        accountDao.updateAccount(target);
34    }
35 }
```

(使用 | 测试类)

main

```
1 public class MyTest {
2     public static void main(String[] args) {
3         ApplicationContext context = new
4             ClassPathXmlApplicationContext("bean.xml");
5         Address address = context.getBean("address", Address.class);
6         System.out.println(address);
7         //applicationContext没有close方法，需要使用具体的子类
8         ((ClassPathXmlApplicationContext)context).close();
9     }
10 }
```

简单测试类

依赖：

- JUnit

```
1 @Test
2 public void testFindone() {
3     //1.获取容器
4     ApplicationContext context = new
5         ClassPathXmlApplicationContext("bean.xml");
6     //2.得到业务层对象
7     AccountService service = context.getBean("accountService",
8         AccountService.class);
9     //3.执行方法
10    Account account = service.findAccountById(1);
11    System.out.println(account);
12 }
```

整合测试类

给测试人员用的，不需要他们去拿context了，他们可能不会。

依赖：

- spring-test
- spring-aop
- JUnit

步骤：

- 导入jar包依赖
- 使用 @RunWith 注解替换原有运行器
- 使用 @ContextConfiguration 执行 spring配置文件的位置
- 使用 @Autowired 给测试类中的变量注入数据

pom

导入spring整合junit的jar坐标

```
1 <dependency>
2   <groupId>org.springframework</groupId>
3   <artifactId>spring-test</artifactId>
4   <version>5.1.6.RELEASE</version>
5 </dependency>
```

@RunWith

使用JUnit提供的一个注解把原有的main方法替换成Spring提供的

```
1 @RunWith(SpringJUnit4ClassRunner.class)
```

@ContextConfiguration

告知Spring的运行器，Spring和Ioc创建是基于xml还是注解的，并且说明位置

- locations : 指定xml文件的位置，加上classpath关键字，表示在类路径下
- classes : 指定注解类所在的位置

```
1 /**
2  * 使用JUnit单元测试，测试我们的配置
3 */
4 @RunWith(SpringJUnit4ClassRunner.class)//创建容器
5 @ContextConfiguration(classes = SpringConfiguration.class)//注解方式
6 @ContextConfiguration(locations = "classpath:bean.xml")//xml方式
7 public class AccountServiceTest {
8
9     @Autowired
10    AccountService service;
11
12    @Test
13    public void testFindAll() {
14        //3.执行方法
15        List<Account> accounts = service.findAllAccount();
16        for (Account account : accounts) {
17            System.out.println(account);
18        }
19    }
20
21    @Test
22    public void testFindOne() {
23        //3.执行方法
24        Account account = service.findAccountById(1);
25        System.out.println(account);
26    }
27
28    @Test
29    public void testSave() {
30        //3.执行方法
31        Account account = new Account(null, "呵呵", 1000F);
32        service.saveAccount(account);
33    }
34}
```

```
35     @Test
36     public void testUpdate() {
37         //3.执行方法
38         Account account = service.findAccountById(3);
39         account.setMoney(10000F);
40         service.updateAccount(account);
41     }
42
43     @Test
44     public void testDelete() {
45         //3.执行方法
46         service.deleteAccount(4);
47     }
48 }
```

Spring IOC

单纯的解决程序间的依赖关系。

loc解决对象间的依赖。

？？？？：单例 多例 与 单线程 多线程

配置的单例多例,要debug多看看。

service dao connection

- 类之间 通常 一对一的 耦合
- 对象 多例 是什么 时候 的多例
- 多线程情况 ,类和类一对一 ,对象单例对多例。什么情况 ?

通过配置的方式实现工厂模式

- 工厂 :反射 创建 bean对象。map容器做统一管理。
 - 创建 :(全路径类名字符串)注册—>工厂—>生成bean对象
 - 提供 :工厂.get 获取bean对象
- 对象 :
 - 注册 —> 工厂 :交由工厂管理
 - 注解
 - xml
 - 注入 :给对象 的 属性 赋值
 - 注解
 - xml

注册Bean到loc容器 :

```

1 <bean id="" class="" scope="" init-method="" destroy-method="">
2   <property name="" value=""|ref=""></property>
3 </bean>

```

控制反转

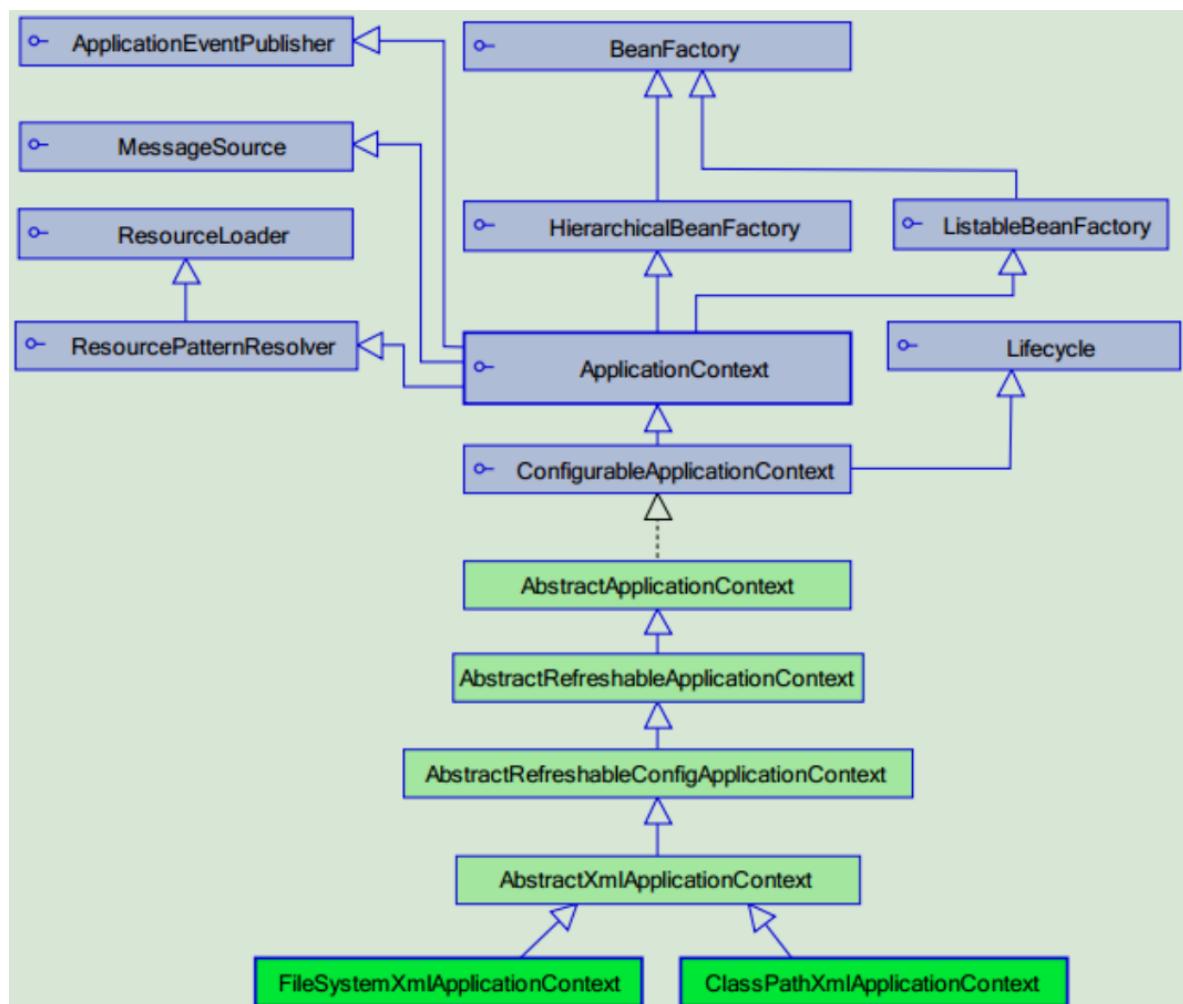
把我们(某java代码片段) 自主创建对象的权力 ,交给 工厂 ,工厂创建对象 ,给我们这里的 未赋值引用 赋值。

```

1 private AccountDao accountDao = new AccountDaoImpl(); //这里的代码是耦合的,写死的
2 //↓将自主创建对象的权力 转交给工厂,工厂根据全限定类名来创建对象。
3 private AccountDao accountDao =
4   (AccountDao)BeanFactory.getBean("accountDao");
  //工厂传参是可变的,配置文件统一管理。只有必须用工厂这里是写死的

```

工厂类



核心容器的两个接口

- BeanFactory 才是 Spring 容器中的顶层接口。

- ApplicationContext 是它的子接口。

BeanFactory 和 ApplicationContext 的区别：创建对象的时间点不一样。

- ApplicationContext：只要一读取配置文件，默认情况下就会创建对象。
 - BeanFactory：什么使用什么时候创建对象。

ApplicationContext

- 它在构建核心容器时，创建对象采取的策略是采用 **立即加载** 的方式。也就是说，只要一读取完配置文件马上就创建配置文件中配置的对象。
 - 单例适用

```
1 ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
2
3 AccountService as = (AccountService)context.getBean("accountService");
4 AccountDao dao = context.getBean("accountDao", AccountDao.class);
```

BeanFactory

(不实用，顶级接口不完善)

- 它在构建核心容器时，创建对象采取的策略是采用 **延迟加载** 的方式。也就是说，什么时候根据id获取对象了，什么时候 根据id获取 对象了，什么时候 才 真正的 创建对象。
 - 多例适用

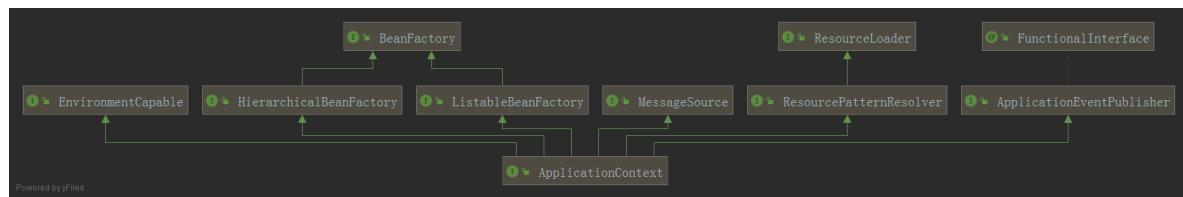
```
1 Resource resource = new ClassPathResource("bean.xml");
2 BeanFactory factory = new XmlBeanFactory(resource);
3
4 AccountService as = (AccountService)factory.getBean("accountService");
```

ApplicationContext

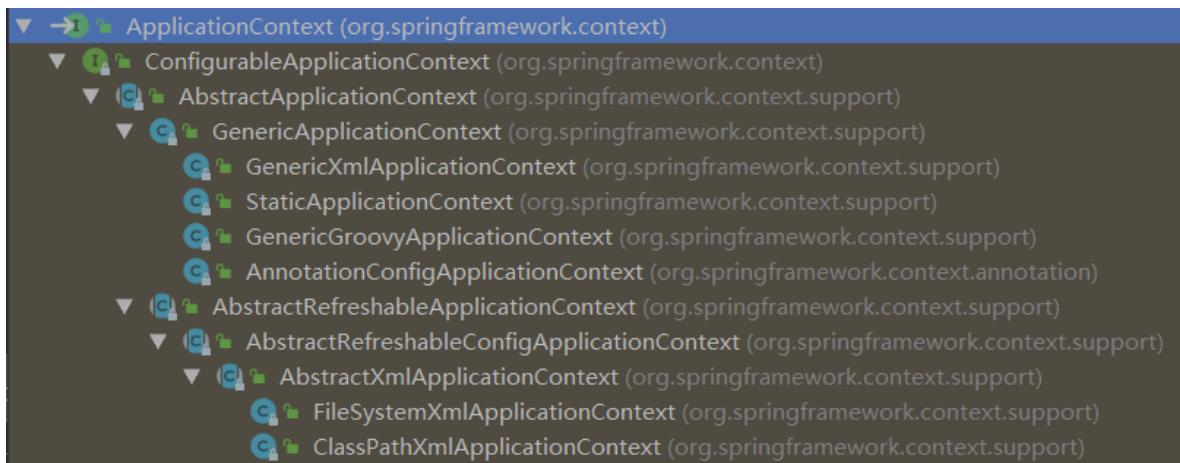
applicationContext没有close方法，需要使用具体的子类

```
1 | ((ClassPathXmlApplicationContext)context).close();
```

继承关系



实现



三个常用实现类

ClassPathXmlApplicationContext

- 它是从 **类的根路径** (跟项目放一起)下加载配置文件。
- 要求配置文件必须在类路径下。不在的话，加载不了。

```

1 | ApplicationContext context =
2 |     new ClassPathXmlApplicationContext("bean.xml");

```

FileSystemXmlApplicationContext

它是从 **磁盘路径** 上加载配置文件，配置文件可以在磁盘的任意位置。

FileSystemXmlApplicationContext：可以加载 **磁盘任意路径** (这里需要是全路径，CDEF盘)下的配置文件(必须有访问权限，是操作系统需要有访问权限，和我们代码没关系)

```

1 |

```

AnnotationConfigApplicationContext

当我们使用注解配置容器对象时，需要使用此类来创建 spring 容器。它用来读取注解。

AnnotationConfigApplicationContext：用于读取**注解**，创建容器

```

1 | ApplicationContext context =
2 |     new AnnotationConfigApplicationContext(SpringConfiguration.class);

```

spring对bean的管理细节

- 创建bean的三种方式
- bean对象的作用范围
- bean对象的生命周期

创建Bean×4

创建对象的分类：

- 自定义类
- 外部jar包

? 区别 ? 静态工厂 ? 实例工厂

-> : 这里其实就是工厂模式

- 工厂方法模式：公司生产产品
- 抽象工厂模式：集团：(子公司1生产产品1；子公司2生产产品2；...)

- 配置对象。让spring来创建。
- 默认情况下它调用的是类中的无参构造。如果没有无参构造则不能创建成功。

属性：

- id : 给对象在容器中提供一个唯一标识。用于获取对象。
- class : 指定类的全限定类名。用于反射创建对象。
 - 默认情况下调用无参构造函数。
- scope : 指定对象的作用范围。
 - singleton :默认值，单例的。
 - prototype :多例的。
 - request :WEB项目中, Spring 创建一个 Bean 的对象, 将对象存入到 request 域中.
 - session :WEB项目中, Spring 创建一个 Bean 的对象, 将对象存入到 session 域中.
 - global session :WEB项目中, 应用在 Portlet 环境. 如果没有 Portlet 环境那么 globalSession 相当于 session.
- init-method : 指定类中的初始化方法名称。
- destroy-method : 指定类中销毁方法名称。

构造函数

使用默认构造函数创建：在spring的配置文件中使用bean标签，配以id和class属性之后，且没有其他属性和标签时。采用的就是默认构造函数创建bean对象，此时如果类中没有**默认构造函数**，则对象无法创建。

- 属性标签，传参

- id : 对象名
- class : 全类名

xml

```
1 | <bean id="accountService" class="com.learn.service.impl.AccountServiceImpl">
| </bean>
```

只要是new对象的过程，坦白来说都可以交由spring容器来控制。

第三方jar包，当然也可以。

在Spring中，很多对象都是单实例的，在日常的开发中，我们经常需要使用某些外部的单实例对象，例如数据库连接池

```
1 <!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
2 <dependency>
3   <groupId>com.alibaba</groupId>
4   <artifactId>druid</artifactId>
5   <version>1.1.21</version>
6 </dependency>
7 <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
8 <dependency>
9   <groupId>mysql</groupId>
10  <artifactId>mysql-connector-java</artifactId>
11  <version>5.1.47</version>
12 </dependency>
```

```
1 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
2   <property name="username" value="root"></property>
3   <property name="password" value="123456"></property>
4   <property name="url" value="jdbc:mysql://localhost:3306/demo"></property>
5   <property name="driverClassName" value="com.mysql.jdbc.Driver">
6 </property>
7 </bean>
```

test

```
1 ApplicationContext context = ClassPathXmlApplicationContext("bean.xml");
2 AccountService as = context.getBean("accountService");
```

```
1 public class MyTest {
2     public static void main(String[] args) throws SQLException {
3         ApplicationContext context = new
4             ClassPathXmlApplicationContext("ioc3.xml");
5         DruidDataSource dataSource = context.getBean("dataSource",
6             DruidDataSource.class);
7         System.out.println(dataSource);
8         System.out.println(dataSource.getConnection());
9     }
10 }
```

工厂实例方法

? : 该类可能是存在于jar包中的，我们无法通过修改源码的方式来提供默认构造函数。

-> : 使用普通工厂中的方法创建对象 | 使用某个类中的方法创建对象，并存入spring容器。

工厂对象.普通方法()

- 实例工厂 bean
- 工厂bean+method = bean对象

实例工厂：工厂本身需要创建对象，工厂类工厂对象=new 工厂类；工厂对象.get对象名()；

- factory-bean : 指定使用哪个工厂实例。用于指定实例工厂 bean 的 id。
- factory-method : 指定使用哪个工厂实例的方法

xml

- factory-bean : 指定使用哪个工厂实例
- factory-method : 指定使用哪个工厂实例的方法

```
1 <bean id="instanceFactory"
2     class="com.learn.factory.InstanceFactory"></bean>
3 <bean id="accountService"
4     class="com.learn.service.impl.AccountServiceImpl"
5     factory-bean="instanceFactory"
6     factory-method="getAccountService"></bean>
7
8 <!--实例工厂使用-->
9 <!--创建实例工厂类-->
10 <bean id="personInstanceFactory"
11     class="com.learn.factory.PersonInstanceFactory"></bean>
12 <!--
13     factory-bean:指定使用哪个工厂实例
14     factory-method:指定使用哪个工厂实例的方法
15 -->
16 <bean id="person6"
17     class="com.learn.bean.Person"
18     factory-bean="personInstanceFactory"
19     factory-method="getPerson">
20     <constructor-arg value="wangwu"></constructor-arg>
21 </bean>
```

工厂类

```
1 /*
2     模拟一个工厂类
3     该类可能是：
4     没有默认构造函数，
5     存在于jar包中，我们无法修改
6 */
7 public class InstanceFactory {
8     public AccountService getAccountService() {
9         return new AccountServiceImpl();
10    }
11 }
```

```
1 public class PersonInstanceFactory {
2     public Person getPerson(String name){
3         Person person = new Person();
4         person.setId(1);
5         person.setName(name);
6         return person;
7     }
8 }
```

test

```
1 ApplicationContext context = ClassPathXmlApplicationContext("bean.xml");
2 AccountService as = context.getBean("accountService");
```

静态工厂方法

使用工厂中的静态方法创建对象 | 使用某个类中的静态方法创建对象，并存入spring容器。

工厂类.静态方法()

静态工厂：工厂本身不需要创建对象，但是可以通过静态方法调用，对象=工厂类.静态工厂方法名()
；

- class : 指定静态工厂类
- factory-method : 指定哪个方法是工厂方法

xml

```
1 <!--
2     静态工厂的使用：
3         class:指定静态工厂类
4         factory-method:指定哪个方法是工厂方法
5 -->
6 <bean id="accountService"
7     class="com.learn.factory.StaticFactory"
8     factory-method="getAccountService">
9 </bean>
10
11 <bean id="person5"
12     class="com.learn.factory.PersonStaticFactory"
13     factory-method="getPerson">
14     <!--constructor-arg: 可以为方法指定参数-->
15     <constructor-arg value="lisi"></constructor-arg>
16 </bean>
```

工厂类

```
1  /*
2   *      加了个 static 修饰方法而已
3   */
4  public class StaticFactory {
5      public static AccountService getAccountService() {
6          return new AccountServiceImpl();
7      }
8  }
```

```
1  public class PersonStaticFactory {
2      //静态方法,两个方法区别就是 差个static
3      public static Person getPerson(String name){
4          Person person = new Person();
5          person.setId(1);
6          person.setName(name);
7          return person;
8      }
9  }
```

test

```
1  ApplicationContext context = ClassPathXmlApplicationContext("bean.xml");
2  AccountService as = context.getBean("accountService");
```

实现FactoryBean创建对象？

? 区别 ? BeanFactory : 定义规范 ? FactoryBean : 获取对象

- BeanFactory , 一个顶级的接口 , 几乎所有的类都要继承于它。定义规范
- FactoryBean , 用于获取对象的。当我们自己想创建一个复杂的bean对象 , 该bean对象不交给Spring容器管理 , 而是由我们自己管理。用FactoryBean创建。
 - getObject
 - getObjectType
 - isSingleton

spring框架里有很多该接口的实现类 , 一般是框架使用的多 , 我们很少去用。

此方式是 spring创建 bean方式的一种补充 , 用户可以按照需求创建对象 , 创建的对象交由spring IOC容器来进行管理。

- **FactoryBean** 是Spring规定的一个接口 , 当前接口的实现类 , Spring都会将其作为一个工厂 , 但是在ioc容器启动的时候不会创建实例 , 只有在使用的时候才会创建对象。
- BeanFactory : 定义规范的接口。

之前spring容器管理 , 单例singleton则上来就创建 , 多例prototype则用时才创建。

- 现在交由自定义工厂FactoryBean，不论单例 多例，均需到用时才创建。
- 无论是否是单例，都是在用到的时候才会创建该对象，不用该对象不会创建。

java

```

1 /**
2  * 实现了FactoryBean接口的类是Spring中可以识别的工厂类，spring会自动调用工厂方法创建
3  * 实例
4  */
5 public class MyFactoryBean implements FactoryBean<Person> {
6
7     /**
8      * 工厂方法，返回需要创建的对象
9      * @return
10     * @throws Exception
11     */
12    @Override
13    public Person getObject() throws Exception {
14        Person person = new Person();
15        person.setName("maliu");
16        return person;
17    }
18
19    /**
20     * 返回创建对象的类型，spring会自动调用该方法返回对象的类型
21     * @return
22     */
23    @Override
24    public Class<?> getObjectType() {
25        return Person.class;
26    }
27
28    /**
29     * 创建的对象是否是单例对象
30     * @return
31     */
32    @Override
33    public boolean isSingleton() {
34        return false;
35    }

```

xml

- 交给spring容器。
- spring会自动调用工厂方法创建实例

```

1 <bean id="myfactorybean"
2      class="com.learn.factory.MyFactoryBean"></bean>

```

Bean对象的作用范围

通过scope属性可以指定当前bean的作用域

- **singleton** : 单例模式 , 从IOC容器中获取的都是同一个对象 , 默认的作用域
 - 每次在创建IOC容器完成之前 , 此对象已经完成创建
- **prototype** : 多例模式 , 从IOC容器中获取的对象每次都是新创建
 - 每次在需要用到此对象的时候才会创建

在spring4.x的版本中还包括另外两种作用域 :

- **request** : 每次发送请求都会有一个新的对象
- **session** : 每次会话都会有一个新的对象

```
1 <bean id="person" class="com.learn.bean.Person" scope="singleton"></bean>
2 <bean id="person" class="com.learn.bean.Person" scope="prototype"></bean>
```

singleton

单例(默认值)

prototype

多例

request

作用于web应用的请求范围

session

作用于web应用的会话范围

global-session

作用域集群环境的会话范围(全局会话范围) , 当不是集群环境时 , 它就是session。跨机器的session , 如下图 :



Bean对象的生命周期

- **init-method** : 在对象创建完成之后会调用初始化方法
- **destory-method** : 在容器关系的时候会调用销毁方法

- 如果是singleton的话，初始化和销毁的方法都存在
- 如果是prototype的话，初始化方法会调用，但是销毁的方法不会调用

单例对象

单例对象：和容器相同

- 出生：当应用加载，创建容器时，对象就被创建了。
- 活着：只要容器在，对象一直活着。
- 死亡：当应用卸载，销毁容器时，对象就被销毁了。

xml

spring容器在创建对象的时候可以指定具体的初始化和销毁方法。

- init-method：在对象创建完成之后会调用初始化方法
- destory-method：在容器关系的时候会调用销毁方法

```

1 <bean id="accounrService"
2   class="com.learn.service.impl.AccountServiceImpl"
3   scope="singleton"
4   init-method="init"
5   destroy-method="destroy"
6 ></bean>

```

java

```

1 public class AccountServiceImpl implements AccountService {
2     public AccountServiceImpl() {
3         System.out.println("对象创建了");
4     }
5
6     public void saveAccount() {
7         System.out.println("service saveAccount方法执行");
8     }
9
10    public void init() {
11        System.out.println("对象初始化了");
12    }
13
14    public void destroy() {
15        System.out.println("对象销毁了");
16    }
17 }

```

test

```

1 ApplicationContext context = ClassPathXmlApplicationContext("bean.xml");
2 AccountService as = context.getBean("accountService");

```

多例对象

多例对象：

- 出生：当使用对象时，创建新的对象实例。
- 活着：只要对象在使用中，就一直活着。
- 死亡：当对象长时间不用，且没有别的对象引用时，由Java垃圾回收机制回收。

xml

```
1 <bean id="accounrService"
2   class="com.learn.service.impl.AccountServiceImpl"
3   scope="prototype"
4   init-method="init"
5   destroy-method="destroy"
6 ></bean>
```

java

```
1 public class AccountServiceImpl implements AccountService {
2     public AccountServiceImpl() {
3         System.out.println("对象创建了");
4     }
5
6     public void saveAccount() {
7         System.out.println("service saveAccount方法执行");
8     }
9
10    public void init() {
11        System.out.println("对象初始化了");
12    }
13
14    public void destroy() {
15        System.out.println("对象销毁了");
16    }
17 }
```

test

```
1 ApplicationContext context = ClassPathXmlApplicationContext("bean.xml");
2 AccountService as = context.getBean("accountService");
```

初始化和销毁

- init-method：在对象创建完成之后会调用初始化方法
- destory-method：在容器关系的时候会调用销毁方法

- 如果是singleton的话，初始化和销毁的方法都存在
- 如果是prototype的话，初始化方法会调用，但是销毁的方法不会调用

java

```
1 public class Address {  
2     private String province;  
3     private String city;  
4     private String town;  
5  
6     public Address() {  
7         System.out.println("address被创建了");  
8     }  
9  
10    public Address(String province, String city, String town) {  
11        this.province = province;  
12        this.city = city;  
13        this.town = town;  
14    }  
15  
16    public String getProvince() {  
17        return province;  
18    }  
19  
20    public void setProvince(String province) {  
21        this.province = province;  
22    }  
23  
24    public String getCity() {  
25        return city;  
26    }  
27  
28    public void setCity(String city) {  
29        this.city = city;  
30    }  
31  
32    public String getTown() {  
33        return town;  
34    }  
35  
36    public void setTown(String town) {  
37        this.town = town;  
38    }  
39  
40    public void init(){  
41        System.out.println("对象被初始化");  
42    }  
43  
44    public void destory(){  
45        System.out.println("对象被销毁");  
46    }  
47  
48    @Override  
49    public String toString() {  
50        return "Address{" +  
51                "province'" + province + '\'' +
```

```
52             ", city='"+ city + '\'' +
53             ", town='"+ town + '\'' +
54             '}';
55     }
56 }
```

xml

spring容器在创建对象的时候可以指定具体的初始化和销毁方法。

- init-method : 在对象创建完成之后会调用初始化方法
- destory-method : 在容器关系的时候会调用销毁方法

```
1 <!--bean生命周期表示bean的创建到销毁
2     如果bean是单例，容器在启动的时候会创建好，关闭的时候会销毁创建的bean
3     如果bean是多例，获取的时候创建对象，销毁的时候不会有任何的调用
4 -->
5 <bean id="address" class="com.learn.bean.Address" init-method="init" destroy-
method="destory"></bean>
```

test

```
1 public class MyTest {
2     public static void main(String[] args) {
3         ApplicationContext context = new
4 ClassPathXmlApplicationContext("ioc2.xml");
5         Address address = context.getBean("address", Address.class);
6         System.out.println(address);
7         //applicationContext没有close方法，需要使用具体的子类
8         ((ClassPathXmlApplicationContext)context).close();
9     }
10 }
```

BeanPostProcessor

用于配置bean对象初始化方法的 前后处理 方法。

spring中包含一个BeanPostProcessor的接口，可以在 bean的初始化方法 的前后 调用该方法，如果配置了初始化方法的 **前置和后置处理器**，无论是否包含初始化方法，都会进行调用。

java

```
1 public class MyBeanPostProcessor implements BeanPostProcessor {
2     /**
3      * 在初始化方法调用之前执行
4      * @param bean 初始化的bean对象
5      * @param beanName xml配置文件中的bean的id属性
6      * @return
7      * @throws BeansException
8      */
9     @Override
```

```

10     public Object postProcessBeforeInitialization(Object bean, String
11         beanName) throws BeansException {
12             System.out.println("postProcessBeforeInitialization:"+beanName+"调用
13             初始化前置方法");
14             return bean;
15         }
16
17     /**
18      * 在初始化方法调用之后执行
19      * @param bean
20      * @param beanName
21      * @return
22      * @throws BeansException
23      */
24     @Override
25     public Object postProcessAfterInitialization(Object bean, String
26         beanName) throws BeansException {
27             System.out.println("postProcessAfterInitialization:"+beanName+"调用
28             初始化后缀方法");
29             return bean;
30         }
31     }

```

xml

有初始化方法init的类，会有创建前后。

没有的，就没有。

使用时 BeanPostProcessor 的实现类在spring容器中注册一下就行。

```

1 <bean id="person" class="com.learn.bean.Person" init-method="init" destroy-
2   method="destory"></bean>
3 <bean id="address" class="com.learn.bean.Address"></bean>
4 <bean id="myBeanPostProcessor" class="com.learn.bean.MyBeanPostProcessor">
5   </bean>

```

Bean之间的继承

- 抽象bean
 - 对象的默认属性
- parent ="bean id"
 - 继承抽象bean属性

```

1 <!-- abstract属性 不能实例化. 使用abstract属性定义抽象bean, 无法进行实例化 -->
2 <bean id="parent" class="com.learn.bean.Person" abstract="true">
3   <property name="id" value="1"></property>
4   <property name="name" value="张三"></property>
5   <property name="gender" value="男"></property>
6   <property name="age" value="20"></property>
7 </bean>
8 <!-- 可以通过parent属性来获取父bean的属性值, 说是继承更合理吧 -->
9 <bean id="son" class="com.learn.bean.Person" parent="parent">
10  <property name="name" value="张小三"></property>
11 </bean>

```

Bean对象创建的依赖关系

`depend-on = "bean id"`

- 对象创建顺序，和定义bean顺序相关。通常是按照xml配置文件定义的顺序。
- 如果需要干扰创建的顺序，可以使用depend-on属性。depend-on指定的bean创建后，才会创建该bean。
- 一般实际工作中不必在意bean创建的顺序，无论谁先创建，需要依赖的对象在创建完成之后都会进行赋值操作。
 - eg：person先创建，address后创建，但是person里包含address。这时person先创建`person.address = null`，等address创建好后，会`person.address = address`。

```

1 <bean id="person" class="com.learn.bean.Person" depend-on="address"></bean>
2 <bean id="address" class="com.learn.bean.Address"></bean>

```

Spring中的依赖注入

- 依赖注入：Dependency Injection。依赖关系的维护，它是spring框架核心ioc的具体实现。
- 依赖关系的管理：以后都交给spring来维护
- 在当前类需要用到其他类的对象：由spring为我们提供，我们只需要在配置文件中说明

依赖注入的数据：

- 基本类型和String
- 其他bean类型(在配置文件中或注解配置过的bean)
- 复杂类型/集合类型

注入的方式：

- 构造函数
- set方法
- 注解提供

注入方式

构造函数注入

给谁赋值：

- type：指定参数在构造函数中的数据类型
 - index：指定参数在构造函数参数列表的索引位置。参数索引：0，1，2...
 - name：指定参数在构造函数中的名称
-
- value：用于提供基本类型和String类型的数据
 - ref：用于指定其他的bean类型数据。指的是在spring的ioc核心容器中出现过的bean对象

优点：

- 在获取bean对象时，注入数据是必须的操作，否则对象无法创建成功

缺点：

- 改变了bean对象的实例化方式，使我们在**创建对象**时，如果用不到这些数据，也必须提供

xml

注入

- constructor-arg：都是构造方法，就看参数列表了。不写就空参，写就找对应的参数列表。

```
1 <bean id="accountService" class="com.learn.service.impl.AccountServiceImpl">
2   <constructor-arg name="name" value="test"></constructor-arg>
3   <constructor-arg name="age" value="18"></constructor-arg>
4   <constructor-arg name="birthday" ref="now"></constructor-arg>
5 </bean>
6
7 <!-- 配置一个日期对象：Date now = new Date(); -->
8 <bean id="now" class="java.util.Date"></bean>
```

类

```
1 public class AccountServiceImpl implements AccountService {
2
3   //如果是经常变化的数据，并不适合注入的方式
4   private String name;
5   private Integer age;
6   private Date birthday;
7
8   public AccountServiceImpl() {
9     System.out.println("对象创建了");
10  }
11
12  public AccountServiceImpl(String name, Integer age, Date birthday) {
13    this.name = name;
```

```

14     this.age = age;
15     this.birthday = birthday;
16     System.out.println("对象创建了");
17 }
18
19 public void saveAccount() {
20     System.out.println("service saveAccount方法执行");
21 }
22 }
```

set方法注入

- 注入只需要set方法

标签property的属性 : (出现在bean标签内部)

- name : 用于指定注入时所调用的set方法名称 ; 找的是类中 set 方法后面的部分
- value : 用于提供 基本类型 和 String类型 的数据
- ref : 用于指定其他的bean类型数据。它指的就是在sprng的loc容器中出现过的bean对象

优点 :

- **创建对象**时没有明确的限制 , 可以直接使用默认构造函数

弊端 :

- 如果有某个成员必须有值 , 则获取对象时 , set方法可能没有执行 , 没set它呗

xml

注入

```

1 <bean id="accountService2" class="com.learn.service.impl.AccountServiceImpl">
2   <property name="name" value="TEST"></property>
3   <property name="age" value="21"></property>
4   <property name="birthday" ref="now"></property>
5 </bean>
6 <!-- 配置一个日期对象:Date now = new Date(); -->
7 <bean id="now" class="java.util.Date"></bean>
```

类

```

1 public class AccountServiceImpl implements AccountService {
2
3   //如果是经常变化的数据 , 并不适用于注入的方式
4   private String name;
5   private Integer age;
6   private Date birthday;
7
8   //set * 3
9 }
```

自动装配autowire

当一个对象中需要引用另外一个对象的时候，在之前的配置中我们都是通过property标签来进行手动配置的。其实在spring中还提供了一个非常强大的功能就是自动装配，使用自动装配的功能，spring会把某些bean注入到另外bean中。可以使用autowire属性来实现自动装配。

可以按照我们指定的规则进行配置，配置的方式有以下几种：

- default/no：不自动装配
- byName：按照名字（serXxx & id）进行装配，根据setXxx方法后面的名称首字母小写去容器中查找组件，如有id为xxx，则进行赋值，如果找不到则装配null
- byType：按照bean的类型进行装配，以属性的类型作为查找依据去容器中找到这个组件，如果有多个类型相同的bean对象，那么会报异常，如果找不到则装配null。**该类型对象须唯一**。？？？使用byType会加载环境变量出来。
- constructor：按照构造器进行装配，先按照有参构造器参数的类型进行装配，没有就直接装配null；如果按照类型找到了多个，那么就使用参数名作为id继续匹配，也就是byType，找到就装配，找不到就装配null，多个也是null。？？？

```
○ 1 public Person(Address address)
  2 {this.address = address;}
```

xml

```
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   xsi:schemaLocation="http://www.springframework.org/schema/beans
5   http://www.springframework.org/schema/beans/spring-beans.xsd">
6 
7   <bean id="address" class="com.mashibing.bean.Address">
8     <property name="province" value="河北"></property>
9     <property name="city" value="邯郸"></property>
10    <property name="town" value="武安"></property>
11  </bean>
12  <bean id="person" class="com.mashibing.bean.Person"
13    autowire="byName"></bean>
14  <bean id="person2" class="com.mashibing.bean.Person"
15    autowire="byType"></bean>
16  <bean id="person3" class="com.mashibing.bean.Person"
17    autowire="constructor"></bean>
18 </beans>
```

p 名称空间注入

本质还是调用set方法。

此种方式是通过在xml中导入p名称空间，使用p:propertyName来注入数据，它的本质仍然是调用类中的set方法实现注入功能。

```
1 beans里加
2 <beans xmlns:p="http://www.springframework.org/schema/p">
3     <bean id="person5" class="com.learn.bean.Person"
4         p:id="5"
5         p:name="王五"
6         p:age="25"
7         p:gender="女"
8     ></bean>
9 </beans>
```

注解

。。。 无需set方法

注入不同数据类型

基本类型和String

其他bean类型

(在配置文件中或注解配置过的bean)

复杂类型/集合类型

用于给List结构注入的标签：

- list
- array
- set

用于给Map结构注入的标签

- map
- props

结构相同，标签可以互换

xml

注入

```
1 <bean id="accountService2"
2     class="com.learn.service.impl.AccountsreviceImpl">
3         <property name="myStrs">
```

```

3     <array>
4         <value>AAA</value>
5         <value>BBB</value>
6         <value>CCC</value>
7     </array>
8 </property>
9 <property name="myList">
10    <list>
11        <value>AAA</value>
12        <value>BBB</value>
13        <value>CCC</value>
14    </list>
15 </property>
16 <property name="mySet">
17    <set>
18        <value>AAA</value>
19        <value>BBB</value>
20        <value>CCC</value>
21    </set>
22 </property>
23 <property name="myMap">
24    <map>
25        <entry key="A" value="aaa"></entry>
26        <entry key="B" value="bbb"></entry>
27        <entry key="C">ccc</entry>
28    </map>
29 </property>
30 <property name="myProps">
31    <props>
32        <prop key="A">aaa</prop>
33        <prop key="B">bbb</prop>
34        <prop key="C">ccc</prop>
35    </props>
36 </property>
37 <property name="age" value="21"></property>
38 <property name="birthday" ref="now"></property>
39 </bean>
40 <!-- 配置一个日期对象:Date now = new Date(); -->
41 <bean id="now" class="java.util.Date"></bean>

```

类

```

1 public class AccountService {
2     private String[] myStrs;
3     private List<String> myList;
4     private Set<String> mySet;
5     private Map<String, String> myMap;
6     private Properties myProps;
7
8     //set方法
9
10 }

```

SpEL #{}

SpEL:Spring Expression Language,spring的表达式语言，支持运行时查询操作对象

使用#{}作为语法规则，所有的大括号中的字符都认为是SpEL。

- 算术运算
- 引用bean
- 调用静态方法
- 调用非静态方法

xml

```
1 <bean id="person4" class="com.mashibing.bean.Person">
2     <!--支持任何运算符-->
3     <property name="age" value="#{12*2}"></property>
4     <!--可以引用其他bean的某个属性值-->
5     <property name="name" value="#{address.province}"></property>
6     <!--引用其他bean-->
7     <property name="address" value="#{address}"></property>
8     <!--调用静态方法-->
9     <property name="hobbies" value="#{T(java.util.UUID).randomUUID().toString().substring(0,4)}"></property>
10    <!--调用非静态方法-->
11    <property name="gender" value="#{address.getCity()}"></property>
12 </bean>
```

XML IOC

xml方式更完整，虽然比注解麻烦。

- 环境配置
- 注册
- 注入
- 获取对象

标签

bean

- id
- class
- factory-bean
- factory-method
- scope

- init-method
- destroy-method
- abstract
- parent
- depend-on
- autowire
 - default/no : 不自动装配
 - byName : 按照名字 (serXxx & id) 进行装配 , 根据 setXxx 方法后面的名称首字母
小写 去容器中查找组件 , 如有 id 为 xxx , 则进行赋值 , 如果找不到则装配 null
 - byType : 按照 bean 的类型进行装配 , 以属性的类型作为查找依据去容器中找到这个组件 , 如果有多个类型相同的 bean 对象 , 那么会报异常 , 如果找不到则装配 null。
该类型对象须唯一。
 - constructor : 按照构造器进行装配 , 先按照有参构造器参数的类型进行装配 , 没有就直接装配 null ; 如果按照类型找到了多个 , 那么就使用参数名作为 id 继续匹配 , 也就是 byType , 找到就装配 , 找不到就装配 null , 多个也是 null. ? ? ?

constructor-arg

构造方法注入属性

- name : 用于指定注入时所调用的 set 方法名称 ; 找的是类中 set 方法后面的部分
- value : 用于提供 基本类型 和 String 类型 的数据
- ref : 用于指定其他的 bean 类型数据。它指的就是在 sprng 的 loc 容器中出现过的 bean 对象

- type : 指定参数在构造函数中的数据类型
- index : 指定参数在构造函数参数列表的索引位置。参数索引 : 0 , 1 , 2...

property

set 方法注入属性

- name : 用于指定注入时所调用的 set 方法名称 ; 找的是类中 set 方法后面的部分
- value : 用于提供 基本类型 和 String 类型 的数据
- ref : 用于指定其他的 bean 类型数据。它指的就是在 sprng 的 loc 容器中出现过的 bean 对象

property

constructor-arg

array

list

map

entry

set

value

props

prop

环境配置

p 命名空间配置

一行即可。

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

引入外部配置文件

properties

```
1 | username=root  
2 | password=123456  
3 | url=jdbc:mysql://localhost:3306/demo  
4 | driverClassName=com.mysql.jdbc.Driver  
5 |  
6 | jdbc.username 加前缀 解决 同名问题
```

xml

- xmlns:context=
 - "xxx/context"
- xsi:schemaLocation=
 - "xxx/context"
 - "xxx/context/spring-context.xsd"

```
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 |    加载外部配置文件  
11 |    在加载外部依赖文件的时候需要context命名空间  
12 | -->  
13 |   <context:property-placeholder  
14 |     location="classpath:dbconfig.properties"/>  
15 |     <bean id="dataSource2" class="com.alibaba.druid.pool.DruidDataSource">  
16 |       <property name="username" value="${username}"></property>  
17 |       <property name="password" value="${password}"></property>  
18 |       <property name="url" value="${url}"></property>  
19 |       <property name="driverClassName" value="${driverClassName}">  
20 |     </property>  
21 |   </bean>
```

或

```

1 <bean
2   class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"
3   >
4     <property name="location" value="classpath:jdbc.properties"/>
5   </bean>

```

test

```

1 public class MyTest {
2   public static void main(String[] args) throws SQLException {
3     ApplicationContext context = new
4     ClassPathXmlApplicationContext("ioc3.xml");
5     //这里切换，传个参就完事
6     DruidDataSource dataSource = context.getBean("dataSource2",
7     DruidDataSource.class);
8     System.out.println(dataSource);
9     System.out.println(dataSource.getConnection());
10   }
11 }

```

注册

通过bean标签。

- id : bean对象名称
- class : 全路径类名
- scope
- init-method
- destroy-method

```
1 <bean id="service" class="com.learn.service.impl.XxServiceImpl"></bean>
```

注入

即，成员属性赋值。

- name : 表示参数列表的名称
- value : 表示实际的具体值
- type : 值的类型
- index : 表示值的下标，从0开始

- Date —— "2020/05/01"

setXxx()常规

xml

```
name="xx" value="xxx"
```

```
1 <bean id="person" class="com.learn.bean.Person">
2   <property name="id" value="1"></property>
3   <property name="name" value="张三"></property>
4   <property name="gender" value="男"></property>
5   <property name="age" value="20"></property>
6 </bean>
```

通过构造器

- 这里属性 name 的值 是由构造器的 **参数列表** 决定的。
- 当通过构造器方法赋值的时候，可以省略 name 属性，但需要和参数列表顺序一致。
 - 非要不一致，可用 index 来标识其下标顺序。0, 1, 2, 3...
- 多个参数个数 相同的构造方法，用下面的。可以认为下面的会覆盖上面的。
 - 非要使用指定构造方法，可以使用 type 的参数。这样可以根据 type 对该属性的类型去寻找对应的构造方法，当然，也未必会找到正确的那个。这里 type 不会 拆装箱。须用 Integer。

xml

```
1 <bean id="person2" class="com.learn.bean.Person">
2   <constructor-arg name="id" value="2"></constructor-arg>
3   <constructor-arg name="name" value="李四"></constructor-arg>
4   <constructor-arg name="age" value="23"></constructor-arg>
5   <constructor-arg name="gender" value="男"></constructor-arg>
6 </bean>
7
8 <!-- 当通过构造器方法赋值的时候，可以省略 name 属性，但需要和参数列表顺序一致 -->
9 <bean id="person3" class="com.learn.bean.Person">
10  <constructor-arg value="3"></constructor-arg>
11  <constructor-arg value="王五"></constructor-arg>
12  <constructor-arg value="25"></constructor-arg>
13  <constructor-arg value="男"></constructor-arg>
14 </bean>
15 <bean id="person3" class="com.learn.bean.Person">
16   <constructor-arg value="3" index="0"></constructor-arg>
17   <constructor-arg value="王五" index="1"></constructor-arg>
18   <constructor-arg value="25" index="2"></constructor-arg>
19   <constructor-arg value="男" index="3"></constructor-arg>
20 </bean>
21
22 <!-- 多个参数个数 相同的构造方法，用下面的。可以认为下面的会覆盖上面的。 -->
23 <bean id="person4" class="com.learn.bean.Person">
24   <constructor-arg value="4"></constructor-arg>
25   <constructor-arg value="王六"></constructor-arg>
26   <constructor-arg value="26"></constructor-arg>
27   <constructor-arg value="男"></constructor-arg>
28 </bean>
```

Person

```
1 | public Person(int id, String name, int age, String gender) {  
2 |     this.id = id;  
3 |     this.name = name;  
4 |     this.age = age;  
5 |     this.gender = gender;  
6 | }
```

自动装配autowire

当一个对象中需要引用另外一个对象的时候，在之前的配置中我们都是通过property标签来进行手动配置的。其实在spring中还提供了一个非常强大的功能就是自动装配，使用自动装配的功能，spring会把某些bean注入到另外bean中。可以使用autowire属性来实现自动装配。

可以按照我们指定的规则进行配置，配置的方式有以下几种：

- default/no：不自动装配
- byName：按照名字（serXxx & id）进行装配，根据setXxx方法后面的名称**首字母小写**去容器中查找组件，如有id为xxx，则进行赋值，如果找不到则装配null
- byType：按照bean的类型进行装配，以属性的类型作为查找依据去容器中找到这个组件，如果有多个类型相同的bean对象，那么会报异常，如果找不到则装配null。**该类型对象须唯一**。？？？使用byType会加载环境变量出来。
- constructor：按照构造器进行装配，先按照有参构造器参数的类型进行装配，没有就直接装配null；如果按照类型找到了多个，那么就使用参数名作为id继续匹配，也就是byType，找到就装配，找不到就装配null，多个也是null。？？？

- 1 | public Person(Address address)
2 | {this.address = address;}

xml

```
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 |   xsi:schemaLocation="http://www.springframework.org/schema/beans  
5 |   http://www.springframework.org/schema/beans/spring-beans.xsd">  
6 |  
7 |     <bean id="address" class="com.mashibing.bean.Address">  
8 |       <property name="province" value="河北"></property>  
9 |       <property name="city" value="邯郸"></property>  
10 |      <property name="town" value="武安"></property>  
11 |    </bean>  
12 |    <bean id="person" class="com.mashibing.bean.Person"  
13 |      autowire="byName"></bean>  
14 |      <bean id="person2" class="com.mashibing.bean.Person"  
15 |        autowire="byType"></bean>  
16 |        <bean id="person3" class="com.mashibing.bean.Person"  
17 |          autowire="constructor"></bean>  
18 |    </beans>
```

命名空间 p:age

主要是解决同名问题：我没看出来解决啥了

```
1 Book {  
2     String name;  
3     double price;  
4     Author author; //这里的name  
5 }  
6 Author {  
7     String name;  
8 }
```

xml

属性写法，简便点而已。

```
1 beans里加  
2 <beans xmlns:p="http://www.springframework.org/schema/p">  
3     <bean id="person5" class="com.learn.bean.Person"  
4         p:id="5"  
5         p:name="王五"  
6         p:age="25"  
7         p:gender="女"  
8     ></bean>  
9 </beans>
```

复杂类型赋值

顾名思义，就是给类中的集合成员传值，它用的也是set方法注入的方式，只不过变量的数据类型都是集合。List,Set,Map,Properties。

- String[] hobbies
- Address address
 - String province
 - String city
 - String town
- List list
- List
 - lists
- Set sets
- Map<String, Object> maps
- Properties properties

数组 <array>

String[] hobbies

```

1 <bean id="person" class="com.learn.bean.Person">
2   <property name="id" value="1"></property>
3   <property name="name" value="张三"></property>
4   <property name="gender" value="男"></property>
5   <property name="age" value="20"></property>
6   <!-- 给数组赋值 -->
7   <property name="hobbies" value="book,girl,movie"></property>
8   <property name="hobbies">
9     <array>
10    <value>book</value>
11    <value>girl</value>
12    <value>movie</value>
13  </array>
14 </property>
15 </bean>

```

对象 | bean ref

给引用类型赋值,可以使用 ref属性 引用外部对象 : `ref="对象 id"`

Address address

- Stirng province
- String city
- String town

```

1 <!-- 给引用类型赋值,可以使用ref引用外部对象 -->
2 <bean id="address" class="com.learn.bean.Address">
3   <property name="province" value="河北省"></property>
4   <property name="city" value="邯郸市"></property>
5   <property name="town" value="武安"></property>
6 </bean>
7 <bean id="person" class="com.learn.bean.Person">
8   <property name="province" ref="address"></property>
9 </bean>

```

List <list>

- List list
 - List
- lists*

```

1 <!-- List<String> -->
2 <bean id="person" class="com.learn.bean.Person">
3   <property name="list" value="1,2,3"></property>
4 </bean>
5
6 <!-- List<Address> -->
7 <bean id="address" class="com.learn.bean.Address">
8   <property name="province" value="河北省"></property>
9   <property name="city" value="邯郸市"></property>
10  <property name="town" value="武安"></property>
11 </bean>

```

```

12 <bean id="person" class="com.learn.bean.Person">
13   <property name="lists">
14     <list>
15       <!-- 使用内部bean, java中无法使用, 无法从IOC容器中直接获取对象的值 -->
16       <bean id="address2" class="com.learn.bean.Address">
17         <property name="province" value="河北省"></property>
18         <property name="city" value="邯郸市"></property>
19         <property name="town" value="武安"></property>
20       </bean>
21       <bean class="com.learn.bean.Address">
22         <property name="province" value="A"></property>
23         <property name="city" value="B"></property>
24         <property name="town" value="C"></property>
25       </bean>
26       <!-- 使用外部bean, 可以随意从ICO容器获取对象的值 -->
27       <ref bean="address"></ref>外面的bean也可以用, 通过id
28     </list>
29   </property>
30 </bean>

```

Set <set>

Set sets

```

1 <bean id="person" class="com.learn.bean.Person">
2   <property name="sets">
3     <set>
4       <value>zhangsan</value>
5       <value>lisi</value>
6       <value>wangwu</value>
7     </set>
8   </property>
9 </bean>

```

Map

Map<String, Object>

- <map>
- <entry>

```

1 <bean id="person" class="com.learn.bean.Person">
2   <property name="maps">
3     <map>
4       <entry key="a" value="aaa"></entry>
5       <entry key="address" value-ref="address"></entry>
6       <entry key="address2">
7         <bean class="com.learn.bean.Address">
8           <property name="province" value="广东省"></property>
9         </bean>
10      </entry>
11      <entry>
12        <key>

```

```
13             <value>heihei</value>
14         </key>
15         <value>haha</value>
16     </entry>
17     <entry key="list">
18         <list>
19             <value>11</value>
20             <value>22</value>
21         </list>
22     </entry>
23   </map>
24 </property>
25 </bean>
```

properties

Properties properties

- <props>
- <prop>

```
1 <bean id="person" class="com.learn.bean.Person">
2   <property name="properties">
3     <props>
4       <prop key="111">aaa</prop>
5       <prop key="222">bbb</prop>
6     </props>
7   </property>
8 </bean>
```

获取对象

- id
- 类型：只有当该类型 对象唯一时，可用。当该类型存在 多个对象 时，则报错。

通过id

xml

```
1 <bean id="person" class="com.learn.bean.Person">
2   <property name="id" value="1"></property>
3   <property name="name" value="张三"></property>
4   <property name="gender" value="男"></property>
5   <property name="age" value="20"></property>
6 </bean>
```

java

```
1 Person person = context.getBean("person", Person.class);
2 Person person = (Person)context.getBean("person");
```

通过类型

只有当该类型 对象唯一 时 , 可用。当该类型存在 多个对象 时 , 则报错。

xml

```
1 <bean id="person" class="com.learn.bean.Person">
2   <property name="id" value="1"></property>
3   <property name="name" value="张三"></property>
4   <property name="gender" value="男"></property>
5   <property name="age" value="20"></property>
6 </bean>
7 <bean id="person2" class="com.learn.bean.Person">
8   <property name="id" value="2"></property>
9   <property name="name" value="李四"></property>
10  <property name="gender" value="女"></property>
11  <property name="age" value="22"></property>
12 </bean>
```

java

```
1 Person person = context.getBean(Person.class);
```

注解 IOC

- 环境配置
- 注册
- 注入
- 获取对象

告知spring在创建容器时要扫描的包 , 配置所需要的标签不是在beans的约束中 , 而是一个名称为 context名称空间和约束中。

- context前缀配置
- 扫描包配置 : 会扫描该包及其子包所有类与接口上的注解

```
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   <context:component-scan base-package="com.learn">会扫描该包及其子包所有类与
10  接口上的注解</context:component-scan>
11 </beans>
```

使用注解需要如下步骤 :

1. 添加上述四个注解中的任意一个
2. 添加自动扫描注解的组件，此操作需要依赖context命名空间
3. 添加自动扫描的标签context:component-scan
 - base-package:指定扫描的基础包，spring在启动的时候会将基础包及子包下所有加了注解的类都自动扫描进IOC容器

注意：当使用注解注册组件和使用配置文件注册组件是一样的，但是要注意：

1. 组件的id默认就是组件的类名首字符小写，如果非要改名字的话，直接在注解中添加即可
 @Controller("whatYouWant") | @Controller(value = "whatYouWant")
2. 组件默认情况下都是单例的，如果需要配置多例模式的话，可以在注解下添加@Scope注解。
 @Scope(value="singleton|prototype")

注解

注册

创建bean对象。和在XML配置文件中编写一个 bean 标签实现的功能是一样的

- @Component : 组件。理论上可以在任意的类上进行添加，在扫描的时候都会完成bean的注册。

@Component的衍生注解：作用与属性与@Component相同；Spring提供的三层使用注解，使三层对象更加清晰；语义作用

- @Controller : 表现层。放置在控制层，用来接受用户的请求
- @Service : 业务层。放置在业务逻辑层
- @Repository : 持久层。放置在数据访问层

上述四个注解写在类上面的时候都可以完成注册bean的功能。在spring程序运行过程中，不会对这四个注解做任何区分，看起来是一样的，都会完成bean的注册功能。在实际的开发过程中，最好能分清楚，提高代码的可读性。

id :

- 把当前类的名称的首字母小写之后做识别的。

@Component

用于把当前类对象存入spring容器中。

- value : 用于指定bean的id，当我们不写时，它的默认值是当前类名，且首字母改小写；
 @Component(value="id") value可省略

@Controller

@Service

@Repository

注入

给bean 设置属性。

和在XML配置文件中的 bean 标签 中写一个 property标签的作用是一样的。

只能注入其他bean类型的数据，而基本类型 和 String类型无法使用如下注解实现：

- @Autowired
- @Qualifier
- @Resource

基本类型 和 String类型：

- @Value

集合类型 的注入只能通过XML来实现。

@Autowired

位置：

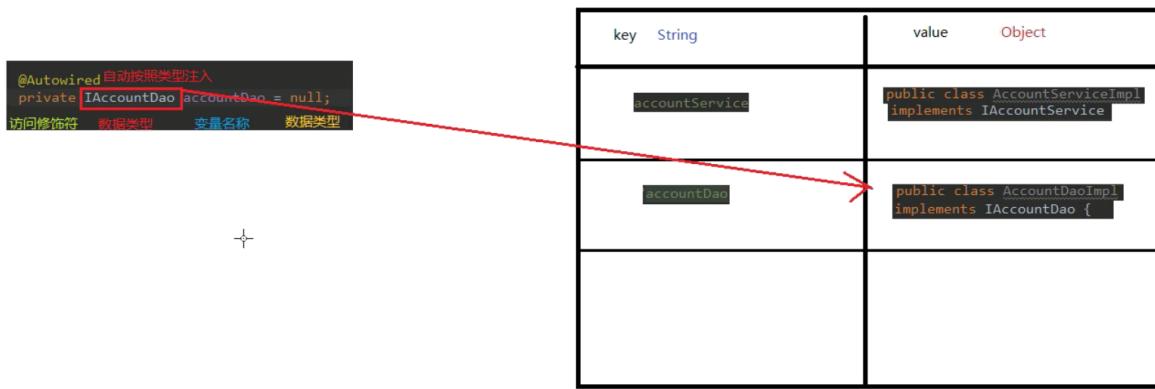
- 变量上
- 方法上
 - 当@.Autowired添加到方法上的时候，此方法在创建对象的时候会 默认调用。
- 同时方法中的参数会进行自动装配。
 - @Qualifier注解也可以定义在方法的参数列表中，用于指定当前属性的id名称。

作用：

- 自动**按照类型** 注入。只要容器中有唯一的一个bean对象类型和要注入的变量类型匹配，就可以注入成功
 1. 如果loc容器中没有任何bean的类型和要注入的变量类型匹配，则报错。 **类型+唯一bean**
 2. 如果loc容器中有多个类型匹配时，转为 id匹配。 **对象名 == bean.id**

细节：

- 它只能注入 bean 类型
- 在使用注解注入时，**set方法就不是必须的了**。没有也能给该属性赋值，自动按照类型注入(都注册到loc容器里，匹配上就 ... 不需要方法了)



当使用@.Autowired注解的时候，自动装配的时候默认是根据类型实现的：

1. 如果只找到一个，则直接进行赋值，
2. 如果没有找到，则直接抛出异常，
3. 如果找到多个，那么会按照变量名作为id继续匹配,
 1. 匹配上直接进行装配
 2. 如果匹配不上则直接报异常
4. 如果想通过名字进行查找，可以自己规定名称，使用注解@Qualifier来指定id，让spring不要使用变量名去匹配id：
 1. 找到，直接装配
 2. 找不到，报错

@Qualifier

作用：

- 在按照类型注入的基础之上再按照名称注入。
 - 在给类成员单独注入时，不能单独使用；需和@Autowired结合使用
 - 在给方法参数注入时，可以单独使用。定义在方法的参数列表中，用于指定当前属性的id名称。

属性：

- value：用于指定注入bean的id

```

1  @Autowired
2  @Qualifier("accountDao1")
3  private AccountDao accountDao = null;
  
```

```
1 @Bean(name="runner")
2 public QueryRunner createQueryRunner(@Qualifier("ds1") DataSource dataSource)
{
3     return new QueryRunner(dataSource);
4 }
5
6 (@Qualifier("ds1") DataSource dataSource)事实上这里是先autowired的，不匹配才会转
7 Qualifier
```

@Resource

作用：

- 直接按照bean的id注入，它可以独立使用
- 只能注入其他 bean 类型。

属性：

- name：用于指定bean的id

使用@Resource可以完成跟@Autowired相同的功能，但是要注意他们之间的区别：

- @Resource是 jdk提供的功能，@Autowired是spring提供的功能
 - @Resource可以在其他框架中使用，而@Autowired只能在spring中使用
 - @Resource扩展性好，而@Autowired支持的框架比较单一
 - @Resource是按照名称进行装配的(先名称，后类型)，而@Autowired是按照类型装配(先类型，后名称)
-
- @AutoWired:是spring中提供的注解，@Resource:是jdk中定义的注解，依靠的是java的标准
 - @AutoWired默认是按照类型进行装配，默认情况下要求依赖的对象必须存在，@Resource默认是按照名字进行匹配的，同时可以指定name属性。
 - @AutoWired只适合spring框架，而@Resource扩展性更好

```
1 @Resource(name = "accountDao1")
2 private AccountDao accountDao = null;
```

@Value

作用：用于注入 基本类型 和 String 类型 的数据。

属性：

- value：用于指定数据的值。它可以使用spring中的**SpEL**(Spring中的EL表达式\${表达式})

jdbcConfig.properties

```
1 jdbc.driver=com.mysql.cj.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/db
3 jdbc.username=root
4 jdbc.password=xxxx
```

```
1 @PropertySource("jdbcConfig.properties")
2 public class SpringConfiguration {}  
3  
4  
5 public class Jdbcconfig {
6     //这里用的是Spring的EL表达式，需要在主配置类上
7     @PropertySource("jdbcConfig.properties")
8         @Value("${jdbc.driver}")
9         private String driver;
10        @Value("${jdbc.url}")
11        private String url;
12        @Value("${jdbc.username}")
13        private String username;
14        @Value("${jdbc.password}")
15        private String password;
16 }
```

作用范围

和在bean标签中使用 scope 属性实现的功能是一样的。

@Scope

作用：指定bean的作用范围

属性：

- value：指定范围的取值：
 - singleton 单例，默认
 - prototype 多例
 - ...不常用

生命周期

和在bean 标签中使用 init-method 和 destroy-method的作用是一样的。

@PostConstruct

作用：用于指定初始化方法

```
1 @PostConstruct
2 public void init() {
3     System.out.println("初始化");
4 }
```

@PreDestroy

作用：用于指定销毁方法

注意：多例对象的销毁 不是 spring负责的，由jvm做垃圾回收。该注解应和单例对象配合使用

```
1 | @PreDestroy  
2 | public void destroy() {  
3 |     System.out.println("销毁");  
4 | }
```

新注解

@Configuration

指定当前类是一个配置类，完成springconfig.xml的功能

细节：

- 当配置类作为AnnotationConfigApplicationContext对象创建的参数时，该注解可以不写。
尽量不要省略

```
1 | ApplicationContext context =  
2 | new AnnotationConfigApplicationContext(SpringConfiguration.class);
```

```
1 | @Configuration  
2 | public class SpringConfiguration {  
3 |  
4 | }
```

@ComponentScan

用于通过注解指定spring在创建容器时要扫描的包

属性：

- value：和basePackages的作用是一样的，都是用于指定创建容器时要扫描的包。

- 1 | @ComponentScan(basePackages = {"com.learn", "com.config"})

- 1 | <context:component-scan base-package="com.learn">
</context:component-scan>

```
1 | @Configuration  
2 | @ComponentScan(basePackages = "com.learn")  
3 | public class SpringConfiguration {  
4 |  
5 | }
```

@Bean

作用：

- 该注解只能写在方法上，表明使用此方法创建一个对象，并且放入spring容器。
- 可以说是专门用于jar包中class准备的。

属性：

- name：用于指定bean的id。当不写时，默认值是当前方法的名称

细节：

- 方法参数：
 - 当我们使用注解配置方法时，如果方法有参数，Spring框架会去容器中查找有没有可用的bean对象。
 - 查找的方式和Autowired注解是一样的：先类型，后id。
 - 也可以用@Qualifier指定id

```
1 @Configuration
2 @ComponentScan(basePackages = "com.learn")
3 public class SpringConfiguration {
4
5     @Bean(name="runner")
6     public QueryRunner createQueryRunner(@Qualifier("ds1") DataSource
7 dataSource) {
8         return new QueryRunner(dataSource);
9     }
10
11     @Bean(name = "ds1")
12     public DataSource createDataSource() {
13         try {
14             ComboPooledDataSource ds = new ComboPooledDataSource();
15             ds.setDriverClass("com.mysql.cj.jdbc.Driver");
16             ds.setJdbcUrl("jdbc:mysql://localhost:3306/db");
17             ds.setUser("root");
18             ds.setPassword("xxxx");
19             return ds;
20         } catch (Exception e) {
21             throw new RuntimeException(e);
22         }
23     }
24 }
```

@Import

用于导入其他的配置类。多个小的配置类，扫描路径配置困难。(把其他的配置类导到主配置类)这时，对应被导入的配置类就可以省略@Configuration了

属性：

- value：用于指定其他配置类的字节码。当我们使用Import的注解的时候，有Import注解的类就是父配置类，而导入的都是子配置类

```
1 @Configuration  
2 @ComponentScan("com.learn")  
3 @Import(JdbcConfig.class)//JdbcConfig不用@Configuration了  
4 public class SpringConfig {}  
5 //这里主配置类SpringConfig和JdbcConfig在同一目录  
6 //主配置类SpringConfig 子配置类JdbcConfig  
7  
8 @Configuration//写不写都行  
9 @PropertySource("classpath:jdbc.properties")  
10 public class JdbcConfig{  
11 }
```

@PropertySource

用于加载.properties 文件中的配置。

属性：用于指定properties文件的位置

- value：指定文件的名称和路径
 - classpath，表示在类路径下

```
1 @ComponentScan("com.learn")  
2 @Import(JdbcConfig.class)  
3 @PropertySource("classpath:jdbcConfig.properties")  
4 public class SpringConfig {}  
5 //这里主配置类SpringConfig和JdbcConfig在同一目录  
6 //主配置类SpringConfig 子配置类JdbcConfig
```

环境配置

多一个依赖：

- spring-aop-xxx.jar

配置类

用于替代bean.xml

```
1 @Configuration  
2 @ComponentScan("com.learn")//扫描包  
3 public class SpringConfiguration {  
4 }
```

XML配置扫描包

必须告诉spring从哪个目录开始扫描。<context:component-scan>

需要事先引入 context命名空间。

context引入

在使用注解的时候，还需要告诉spring应该从哪个包开始扫描。利用context命名空间。

- xmlns:context="<http://www.springframework.org/schema/context>"
- xsi:schemaLocation="<http://www.springframework.org/schema/context>
<http://www.springframework.org/schema/context/spring-context.xsd>"
 - 一开始有beans，我们加个context

```
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 </beans>
```

扫描包

当前这个包下的所有类，都会被完成最基本的扫描功能。告知Spring在创建容器时要扫描的包，配置所需要的标签不是在beans的约束中，而是一个名称为context名称空间和约束中。

```
<context:component-scan>
```

```
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     <context:component-scan base-package="com.learn"></context:component-
10    scan>
</beans>
```

包含|排除 扫描包

在定义好注解的扫描路径之后，可以做更细粒度的控制：选择扫描那个注解，或不扫描哪个注解。

标签：

- `<include-filter>`：表示要包含扫描的注解。一般不用，但是如果引入的第三方包中包含注解，此时就需要使用此标签来进行标识
- `<exclude-filter>`：表示要排除扫描的注解

属性：

- type : 表示指定过滤的规则
 - assignable : 可分配的。可以指定对应的类的名称，但是必须是完全限定名。指定排除某个具体的类，按照类排除
 - annotation : 按照注解进行排除|包含，标注了指定注解的组件。必须是完全限定名
 - regex : 使用正则表达式过滤，不用
 - aspectj : 后面讲aop的时候说明要使用的aspectj表达式，不用
 - custom : 自定义方式。定义一个typeFilter,自己写代码决定哪些类被过滤掉，不用
- expression : 表达式，表示要过滤的注解，表示不注册的具体类名

```
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   <!--指定只扫描哪些组件，默认情况下是全部扫描的，所以此时要配置的话需要在
10  component-scan标签中添加 use-default-filters="false"-->
11  <context:component-scan base-package="com.learn" use-default-
12  filters="false">
13    <!-- 包含注解 -->
14    <context:exclude-filter type="annotation"
15      expression="org.springframework.stereotype.Controller"/>
16
17  <!-- 排除类 -->
18  <context:include-filter type="assignable"
19    expression="com.learn.service.PersonService"/>
20
21 </context:component-scan>
22
23 </beans>
```

注册

@Component 作用和 XML 中的 bean 标签 功能相同。用于把当前类对象存入Spring容器中。
(@Service @Controller @Repository)

属性：

- value : 用于指定bean的 id。当我们不写时，它的默认值是当前类名，且首字母 改小写

注入

@Autowired : 自动按照类型注入。只要容器中有唯一的一个bean对象类型和要注入的变量类型匹配，就可以注入成功

1. 如果loc容器中没有任何bean的类型和要注入的变量类型匹配，则报错。 **类型+唯一bean**
2. 如果loc容器中有多个类型匹配时，转为 id匹配。 **对象名 == bean.id**

作用和XML配置文件中 的bean标签中写一个 property标签 的作用是一样的。

获取对象

同XML

注解与XML

	基于XML配置	基于注解配置
Bean定义		@Component衍生类 @Repository@Service@Controller
Bean名称	通过id或name指定	@Component("person")不写默认类名首字母小写
Bean注入	或者p命名空间	@Autowired按类型注入 @Qualifier按名称注入
生命过程、作用范围	init-method destroy-method 范围 scope属性	@PostConstruct初始化 @PreDestroy销毁 @Scope设置作用范围
适合场景	Bean来自第三方	Bean的实现类由用户自己开发

Spring AOP

通过 配置的方式 实现 动态代理。

- 目标对象：根据目标对象 及 方法 —>切入点表达式。无需修改
- 代理对象：Spring框架动态生成。Spring框架完成，无需考虑
- 通知类：包含通知对应的功能，一组 增强方法
 - xml方式：通知——切入点
 - 注解方法：通知——切入点

子类？接口

- 可选是子类还是接口。在 spring 中，框架会根据目标类是否实现了接口来决定采用哪种动态代理的方式。

开发阶段（我们做的）：

- 编写核心业务代码（开发主线）：大部分程序员来做，要求熟悉业务需求。
- 把公用代码抽取出来，制作成通知。（开发阶段最后再做）：AOP 编程人员来做。

- 在配置文件中，声明切入点与通知间的关系，即切面。：AOP 编程人员来做。

运行阶段（Spring 框架完成的）：

- Spring 框架 监控切入点方法 的执行
- 一旦监控到 切入点方法 被运行
- 使用 代理机制，动态创建 目标对象 的 代理对象
- 根据 通知类别(前后环绕异常)，在 代理对象 的 对应位置，将 通知对应的功能 织入
- 完成完整的代码逻辑运行

动态代理

- 特点：字节码随用创建，随用随加载
- 作用：不修改源码的基础上对方法增强

区别：装饰者模式一上来必须写好一个类，动态代理没必要

基于接口

- 涉及的类：Proxy
- 提供者：JDK官方

如何创建代理对象：

- 使用 **Proxy** 类中的 **newProxyInstance** 方法
 - ClassLoader：类加载器：**xxx.getClass().getClassLoader()**
 - 用于加载代理对象字节码。和被代理对象使用相同的类加载器
 - Class[]：字节码数组：**xxx.getClass().getInterfaces()**
 - 用于让代理对象和被代理对象有相同的方法。
 - InvocationHandler
 - 用于提供增强的代码，写如何代理，一般都是写一个该接口的实现类。通常情况下都是匿名内部类
 - **invoke(Object proxy, Method method, Object[] args)**：被代理对象的任何接口方法都会经过该方法；拦截的功能，全**拦截**
 - proxy：代理对象的引用，一般不同
 - method：当前执行的方法
 - args：当前执行方法所需的参数
 - return Object：和被代理对象方法有相同的返回值

创建代理对象的要求：

- 被代理类最少实现一个 **接口**，如果没有则不能使用
- 需要一个被代理 对象，须加final修饰

接口

- 代理对生产厂家的要求
- 也就是被代理对象(厂家)可以被增强的方法，要提取出来

```

1 /**
2  * 对生产厂家要求的接口
3 */
4 public interface IProducer {
5     /**
6      * 销售
7      * @param money
8      */
9     public void saleProduct(float money);
10
11    /**
12     * 售后
13     * @param money
14     */
15    public void afterService(float money);
16}

```

实现类|被代理类

```

1 /**
2  * 一个生产者
3 */
4 public class Producer implements IProducer {
5
6     /**
7      * 销售
8      * @param money
9      */
10    public void saleProduct(float money) {
11        System.out.println("销售产品,拿到钱:" + money);
12    }
13
14    /**
15     * 售后
16     * @param money
17     */
18    public void afterService(float money) {
19        System.out.println("提供售后服务,拿到钱" + money);
20    }
21
22}

```

代理实现

```

1 /**
2  * 模拟消费者
3 */

```

```

4  public class Client {
5      public static void main(String[] args) {
6          final Producer producer = new Producer();
7          //      producer.saleProduct(10000f);
8          IProducer proxyProducer = (IProducer)Proxy.newProxyInstance(
9              producer.getClass().getClassLoader(),
10             producer.getClass().getInterfaces(),
11             new InvocationHandler() {
12                 public Object invoke(Object proxy,
13                                     Method method,
14                                     Object[] args) throws Throwable {
15                     //提供增强代码
16                     Object returnValue = null;
17                     //1.获取方法执行的参数
18                     Float money = (Float)args[0];
19                     //2.判断当前方法是不是销售
20                     if ("saleProduct".equals(method.getName())) {
21                         returnValue = method.invoke(producer, money *
22                         0.8f);
23                     }
24                     return returnValue;
25                 }
26             });
27             proxyProducer.saleProduct(10000f);
28         }

```

基于子类

- 要求：被代理类不能用 final 修饰的类（最终类）。最终类不能创建子类
- 涉及的类：Enhancer
- 提供者：第三方cglib库

如何创建代理对象：

- 使用 **Enhancer** 类中的 **create** 方法
 - Class 字节码；用于指定被代理对象的字节码；
 - xxx.getClass()
 - Callback 用于提供增强的代码；我们一般都是该接口的子接口实现类：MethodInterceptor extends callback
 - new MethodInterceptor() {}
 - **intercept**(Object proxy, Method method, Object[] args, MethodProxy methodProxy)：方法拦截；执行被代理对象的任何方法都会经过该方法
 - proxy：代理对象的引用，一般不同
 - method：当前执行的方法
 - args：当前执行方法所需的参数
 - methodProxy：执行当前方法的代理对象

依赖

- cglib : 第三方的 CGLib , 如果报 asmxxxx 异常 , 需要导入 asm.jar。

被代理类

```
1 /**
2  * 一个生产者
3 */
4 public class Producer {
5
6     /**
7      * 销售
8      * @param money
9      */
10    public void saleProduct(float money) {
11        System.out.println("销售产品,拿到钱:" + money);
12    }
13
14    /**
15     * 售后
16     * @param money
17     */
18    public void afterService(float money) {
19        System.out.println("提供售后服务,拿到钱" + money);
20    }
21
22 }
```

代理实现

```
1 /**
2  * 模拟消费者
3 */
4 public class Client {
5     public static void main(String[] args) {
6         final Producer producer = new Producer();
7
8         Producer cglibProducer = (Producer)Enhancer.create(
9             producer.getClass(), new MethodInterceptor() {
10
11             public Object intercept(Object proxy,
12                                 Method method,
13                                 Object[] args,
14                                 MethodProxy methodProxy) throws
15             Throwable {
16                 //提供增强代码
17                 Object returnValue = null;
18             }
19         });
20
21         cglibProducer.saleProduct(100);
22         cglibProducer.afterService(100);
23     }
24 }
```

```

17         //1.获取方法执行的参数
18         Float money = (Float) args[0];
19         //2.判断当前方法是不是销售
20         if ("saleProduct".equals(method.getName())) {
21             returnValue = method.invoke(producer, money * 0.8f);
22         }
23         return returnValue;
24     }
25 );
26 cglibProducer.saleProduct(12000f);
27 }
28 }
```

环境依赖

pom

- context , loc。Spring的核心容器是必需的
- 解析切入点表达式

```

1 <dependency>
2   <groupId>org.springframework</groupId>
3   <artifactId>spring-context</artifactId>
4   <version>5.2.5.RELEASE</version>
5 </dependency>
6
7 <!-- 解析切入点表达式 -->
8 <dependency>
9   <groupId>org.aspectj</groupId>
10  <artifactId>aspectjweaver</artifactId>
11  <version>1.8.2</version>
12 </dependency>
```

AOP相关概念

..... X。X

Joinpoint连接点

所谓连接点是指那些被拦截到的点。在 spring 中,这些点指的是方法,因为 spring 只支持方法类型的连接点。

接口中的 方法 , 连接 "业务" 和 "增强方法" 的那个点。我们把增强代码 | 事务控制代码 加到 业务代码 是通过这些接口中的方法来完成的。

能被增强的方法。

Pointcut切入点

所谓切入点是指我们要对哪些 Joinpoint 进行拦截的定义。

被增强的方法。

Advice通知

所谓通知是指拦截到 Joinpoint 之后所要做的事情就是通知。

增强的内容

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：环绕通知中有明确的切入点调用

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    //消除重复代码 确实，事务代码一遍完事 全在这里做
    //TransactionManager和Service的耦合降低, Tx方法该，这里改1次，Service改n多次
    //多个代码片段公有部分的提取
    Object returnVal = null;
    try {
        //1. 开启事务
        txManager.beginTransaction();
        //2. 执行操作
        //List<Account> accounts = accountDao.findAllAccount();
        returnVal = method.invoke(accountService, args);
        //3. 提交事务
        txManager.commit();
        //4. 返回结果
        return returnVal;
    } catch (Exception e) {
        //5. 回滚事务
        txManager.rollback();
        throw new RuntimeException(e);
    } finally {
        //6. 释放连接
        txManager.release();
    }
}
```

Introduction 引介

引介是一种特殊的通知在不修改类代码的前提下, Introduction 可以在运行期为类动态地添加一些方法或 Field。

Target 目标对象

代理的目标对象。也就是 被代理对象。

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。(加入增强的过程)

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。

原有Service对象没法对实现事务的支持，创建一个新的对象|代理对象，在返回代理对象的过程中，加入事务支持。

Proxy 代理

一个类被 AOP 织入增强后，就产生一个 结果代理类。

Aspect 切面

是 切入点 和 通知（引介）的结合。

通知

前置通知

后置通知

异常通知

最终通知

环绕通知

切入点表达式

关键字：

- execution(表达式)

表达式

- 访问修饰符 返回值 包名.包名.包名...类名.方法名(参数列表)

标准表达式写法：

```
1 | public void com.learn.service.impl.AccountServiceImpl.saveAccount()
```

访问修饰符可以省略：

```
1 |     void com.learn.service.impl.AccountServiceImpl.saveAccount()
```

返回值可以使用通配符，表示任意返回值：

```
1 |     * com.learn.service.impl.AccountServiceImpl.saveAccount()
```

包名可以使用通配符，表示任意包。但是有几级包，就需要写几个`*`：

```
1 |     * *.*.*.*.AccountServiceImpl.saveAccount()
```

包名可以使用`..`表示当前包及其子包

```
1 |     * *..AccountServiceImpl.saveAccount()
```

类名和方法名都可以使用`*`来实现通配

```
1 |     * *..*.*()
```

参数列表

- 可以直接写数据类型
 - 基本数据类型直接写名称 int
 - 引用类型写包名.类名 java.lang.String
- 可以使用通配符`*`表示任意类型，但是必须有参数
- 可以使用`..`表示有无参数均可，有参数可以是任意类型

常用写法

全通配写法：

```
1 |     * *..*.*(..)
```

实际开发中切入点表达式的通常写法

```
1 | 切到业务层实现类下的所有方法  
2 |     * com.learn.service.impl.*.*(..)
```

XML AOP

通知类Bean : Logger类

AOP配置 :

- aop:config : 开始AOP配置
 - aop:aspect : 配置切面
 - id : 给切面提供一个唯一标识
 - ref : 指定 **通知类bean** 的 Id
- ↓在aop:aspect内部使用对应的标签来配置通知的类型↓
- aop:before : 前置通知
 - method : 用于指定 通知类bean中 **哪个方法** 是 前置通知
 - pointcut : 用于指定**切入点表达式** , 该表达式的含义指的是对(业务层)中哪些方法增强。

环境配置

bean.xml加入aop , 加入后即可直接使用。

bean.xml加入aop

加入后即可直接使用。

- aop

```
1 <beans xmlns="http://www.springframework.org/schema/beans"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3
4   xmlns:aop="http://www.springframework.org/schema/aop"
5
6   xsi:schemaLocation="http://www.springframework.org/schema/beans
7   http://www.springframework.org/schema/beans/spring-beans.xsd
8
9   http://www.springframework.org/schema/aop
10  http://www.springframework.org/schema/aop/spring-aop.xsd
11 ">
```

标签

```
1 <!-- 配置Logger类 -->
2 <!-- 写一个日志的通知 -->
3 <bean id="logger" class="com.learn.utils.Logger">
4
5 </bean>
6
7 <!-- 配置AOP -->
8 <aop:config>
9   <!-- 配置切入点表达式
10      在aop:aspect标签内部 只能在当前切面使用
```

```

11          它还可以写aop:aspect外面，此时就变成了所有切面可用，必须写在切面前
12      -->
13      <aop:pointcut id="pt2" expression="execution(public *
14 com.learn.service.impl.*.*(..))"/>
15      <!-- 配置切面 引用通知 -->
16      <aop:aspect id="logAdvice" ref="logger">
17          <!-- 前置通知：在切入点方法执行之前执行 -->
18          <aop:before method="beforePrintLog" pointcut="execution(public *
19 com.learn.service.impl.*.*(..))"></aop:before>
20          <!-- 后置通知：在切入点方法正常执行之后执行，它和后置通知永远只能执行一个 -->
21          <aop:after-returning method="afterReturningPrintLog" pointcut-
22 pointcut="execution(public * com.learn.service.impl.*.*(..))"></aop:after-
23 returning>
24          <!-- 异常通知：在切入点方法执行产生异常之后执行 -->
25          <aop:after-throwing method="afterThrowingPrintLog" pointcut-
26 ref="pt2"></aop:after-throwing>
27          <!-- 最终通知：无论切入点方法是否正常执行它都会在其后面执行 -->
28          <aop:after method="afterPrintLog" pointcut-ref="pt1"></aop:after>
29
30          <!-- 配置切入点表达式
31              在aop:aspect标签内部 只能在当前切面使用
32              它还可以写aop:aspect外面，此时就变成了所有切面可用，但必须写在切面前
33          -->
34          <aop:pointcut id="pt1" expression="execution(public *
35 com.learn.service.impl.*.*(..))"/>
36
37          <!-- 详细注释 情况logger类中 -->
38          <aop:around method="aroundPrintLog" pointcut-ref="pt1">
39      </aop:around>
40      </aop:aspect>
41  </aop:config>

```

aop:config AOP

表明开始AOP的配置

aop:advisor

aop:aspect 切面

表明配置切面

- id : 给切面提供一个唯一标识
- ref : 指定 **通知类bean** 的id

aop:pointcut 切入点

位置：

- 在aop:aspect标签内部，只能在当前切面使用

- 在aop:aspect标签外面，所有切面可用，必须写在切面前

属性：

- id
- expression 切入点表达式

通知

在aop:aspect标签的内部使用对应标签来配置通知的类型。

属性：

- method : 用于指定Logger类中哪个方法是前置通知
- pointcut : 用于指定 切入点表达式 ,该表达式的含义指的是对业务层中 哪些方法 增强
- pointcut-ref : 用于指定切入点的表达式的引用

aop:before 前置通知

aop:after-returning 后置通知

aop:after-throwing 异常通知

aop:after 最终通知

aop:around 环绕通知

通知类

常规类，里面提供方法即可。无需多余处理。

- 前置 后置 异常 最终 四个 通知方法，其增强的 切入点方法 会自动执行。
- 环绕 通知方法 ，需要手动执行 切入点方法 。

```
2 * 用于记录日志的工具类,它里面提供了公共的代码
3 */
4 public class Logger {
5
6     /**
7      * 用于打印日志:并且计划在其切入点方法执行之前执行
8      * 切入点方法就是业务层方法
9      */
10     public void printLog() {
11         System.out.println("Logger类中的printLog方法开始记录日志了...");
```

```

53     */
54     public Object aroundPrintLog(ProceedingJoinPoint pjp) {
55         Object rtValue = null;
56         try {
57             System.out.println("环绕通知 前置 Logger类中的around print Log方法
开始记录日志了...");
58             Object[] args = pjp.getArgs(); //得到方法执行所需的参数
59             rtValue = pjp.proceed(args); //明确调用业务层方法
60             System.out.println("环绕通知 后置 Logger类中的around print Log方法
开始记录日志了...");
61             return rtValue;
62         } catch (Throwable throwable) {
63             System.out.println("环绕通知 异常 Logger类中的around print Log方法
开始记录日志了...");
64             throw new RuntimeException(throwable);
65         } finally {
66             System.out.println("环绕通知 最终 Logger类中的around print Log方法
开始记录日志了...");
67         }
68     }
69 }
```

环绕通知

需要注意的是，**环绕通知方法**：

- 参数列表可接收(ProceedingJoinPoint pjp)
- 需要手动执行 切入点方法

```

1 Object[] args = pjp.getArgs(); //得到方法执行所需的参数
2 Object rtValue = pjp.proceed(args);
3 return rtValue;
```

问题：

- 当我们配置了环绕通知之后，切入点方法没有执行，而通知方法执行了

分析：

- 通过对动态代理中的环绕通知代码，发现动态代理的环绕通知有 明确的 切入点调用，而我们的代码中没有

解决：

- Spring框架为我们提供了一个接口：ProceedingJoinPoint。该接口有一个方法proceed()，此方法就相当于明确调用切入点方法。
- 该接口可以作为环绕通知的方法参数，在程序执行时，Spring框架会为我们提供该接口的实现类供我们使用

Spring中的环绕通知：

- 它是Spring框架为我们提供的一种可以在代码中手动控制增强方法合适执行的方式

通知-切入点

在xml配置文件中配置。通过 **切入点表达式** 将 **通知** 和 **切入点** 进行绑定。

```
1 <!-- 配置AOP -->
2 <aop:config>
3     <!-- 配置切入点表达式
4         在aop:aspect标签内部 只能在当前切面使用
5             它还可以卸载aop:aspect外面，此时就变成了所有切面可用，必须写在切面前
6         -->
7         <aop:pointcut id="pt2" expression="execution(public *
com.learn.service.impl.*.*(..))"/>
8             <!-- 配置切面 引用通知 -->
9             <aop:aspect id="logAdvice" ref="logger">
10                 <!-- 前置通知：在切入点方法执行之前执行 -->
11                 <aop:before method="beforePrintLog" pointcut="execution(public *
com.learn.service.impl.*.*(..))"></aop:before>
12                 <!-- 后置通知：在切入点方法正常执行之后执行，它和后置通知永远只能执行一个 -->
13                 <aop:after-returning method="afterReturnngPrintLog"
pointcut="execution(public * com.learn.service.impl.*.*(..))"></aop:after-
returning>
14                 <!-- 异常通知：在切入点方法执行产生异常之后执行 -->
15                 <aop:after-throwing method="afterThrowingPrintLog" pointcut-
ref="pt2"></aop:after-throwing>
16                 <!-- 最终通知：无论切入点方法是否正常执行它都会在其后面执行 -->
17                 <aop:after method="afterPrintLog" pointcut-ref="pt1"></aop:after>
18
19             <!-- 配置切入点表达式
20                 在aop:aspect标签内部 只能在当前切面使用
21                 它还可以卸载aop:aspect外面，此时就变成了所有切面可用，但必须写在切面
前
22             -->
23             <aop:pointcut id="pt1" expression="execution(public *
com.learn.service.impl.*.*(..))"/>
24
25             <!-- 详细注释 情况logger类中 -->
26             <aop:around method="aroundPrintLog" pointcut-ref="pt1">
27             </aop:around>
28         </aop:aspect>
29     </aop:config>
```

注解 AOP

问题：执行调用有问题，最终 在后置和异常的前面

注解还是用环绕吧。不要使用最终注解。

环境配置

- 开注解
- 开AOP注解

xml配置

- context
- aop

```

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
5   xmlns:aop="http://www.springframework.org/schema/aop"
6   xmlns:context="http://www.springframework.org/schema/context"
7
8   xsi:schemaLocation="
9     http://www.springframework.org/schema/beans
10    http://www.springframework.org/schema/beans/spring-beans.xsd
11    http://www.springframework.org/schema/aop
12    http://www.springframework.org/schema/aop/spring-aop.xsd
13    http://www.springframework.org/schema/context
14    http://www.springframework.org/schema/context/spring-context.xsd">
15
16  <!-- 配置Spring创建容器时要扫描的包 支持注解 -->
17  <context:component-scan base-package="com.learn"></context:component-
18  scan>
19
20  <!-- 配置Spring开启注解AOP的支持 -->
21  <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
22 </beans>
```

纯注解配置

```

1 @Configuration
2 @ComponentScan(basePackages="com.learn")
3 @EnableAspectJAutoProxy//aop注解支持
4 public class SpringConfiguration {
5 }
```

注解

@Aspect

声明当前类是一个切面类。

```

1 @Component("logger")
2 @Aspect
3 public class Logger {}
```

@Pointcut

切入点声明。

```
1 @Pointcut("execution(public * com.learn.service.impl.*.*(..))")
2 private void pt1() {}
```

@Before

前置通知

```
1 @Before("pt1()")
2 public void beforePrintLog() {
3     System.out.println("前置通知 Logger类中的before print Log方法开始记录日志
4     了...");
```

@AfterReturning

后置通知

```
1 @AfterReturning("pt1()")
2 public void afterReturnngPrintLog() {
3     System.out.println("后置通知 Logger类中的after returning print Log方法开始记
4     录日志了...");
```

@AfterThrowing

异常通知

```
1 @AfterThrowing("pt1()")
2 public void afterThrowingPrintLog() {
3     System.out.println("异常通知 Logger类中的after throw print Log方法开始记录日
4     志了...");
```

@After

最终通知。finally

```
1 @After("pt1()")
2 public void afterPrintLog() {
3     System.out.println("最终通知 Logger类中的after print Log方法开始记录日志
4     了...");
```

@Around

环绕通知

```

1  @Around("pt1()")
2  public Object aroundPrintLog(ProceedingJoinPoint pjp) {
3      Object rtValue = null;
4      try {
5          System.out.println("环绕通知 前置 Logger类中的around print Log方法开始记录日志了..."); //得到方法执行所需的参数
6          rtValue = pjp.proceed(args); //明确调用业务层方法
7          System.out.println("环绕通知 后置 Logger类中的around print Log方法开始记录日志了..."); //将结果返回给调用者
8          return rtValue;
9      } catch (Throwable throwable) {
10         System.out.println("环绕通知 异常 Logger类中的around print Log方法开始记录日志了..."); //捕获异常并打印
11         throw new RuntimeException(throwable);
12     } finally {
13         System.out.println("环绕通知 最终 Logger类中的around print Log方法开始记录日志了..."); //最后打印
14     }
15 }
16 }
```

通知类

加入注解。

Spring的事务管理器，提供包裹的样板代码。也就是通知类，我们无法对其进行修改。

只能利用 加注解 到类或方法的方式，将指定的类或方法开启事务控制。

```

1  @Component("logger")
2  @Aspect//当前类是一个切面类
3  public class Logger {
4
5      @Pointcut("execution(public * com.learn.service.impl.*.*(..))")
6      private void pt1() {}
7
8      /**
9       * 用于打印日志：并且计划在其切入点方法执行之前执行
10      *
11      * 切入点方法就是业务层方法
12      */
13      public void printLog() {
14          System.out.println("Logger类中的printLog方法开始记录日志了..."); //将结果返回给调用者
15      }
16
17      /**
18       * 前置通知
19       */
20      @Before("pt1()")
21      public void beforePrintLog() {
22          System.out.println("前置通知 Logger类中的before print Log方法开始记录日志了..."); //捕获异常并打印
23      }
24      /**
25       */
```

```

25     * 后置通知
26     */
27     @AfterReturning("pt1()")
28     public void afterReturningPrintLog() {
29         System.out.println("后置通知 Logger类中的after returning print Log方法
开始记录日志了...");
30     }
31
32     /**
33     * 异常通知
34     */
35     @AfterThrowing("pt1()")
36     public void afterThrowingPrintLog() {
37         System.out.println("异常通知 Logger类中的after throw print Log方法开始
记录日志了...");
38     }
39
40     /**
41     * 最终通知
42     */
43     @After("pt1()")
44     public void afterPrintLog() {
45         System.out.println("最终通知 Logger类中的after print Log方法开始记录日志
了...");
46     }
47
48
49     @Around("pt1()")
50     public Object aroundPrintLog(ProceedingJoinPoint pjp) {
51         Object rtValue = null;
52         try {
53             System.out.println("环绕通知 前置 Logger类中的around print Log方法
开始记录日志了...");
54             Object[] args = pjp.getArgs();//得到方法执行所需的参数
55             rtValue = pjp.proceed(args);//明确调用业务层方法
56             System.out.println("环绕通知 后置 Logger类中的around print Log方法
开始记录日志了...");
57             return rtValue;
58         } catch (Throwable throwable) {
59             System.out.println("环绕通知 异常 Logger类中的around print Log方法
开始记录日志了...");
60             throw new RuntimeException(throwable);
61         } finally {
62             System.out.println("环绕通知 最终 Logger类中的around print Log方法
开始记录日志了...");
63         }
64     }
65 }

```

通知-切入点

在通知类中 配置 切入点表达式。

```

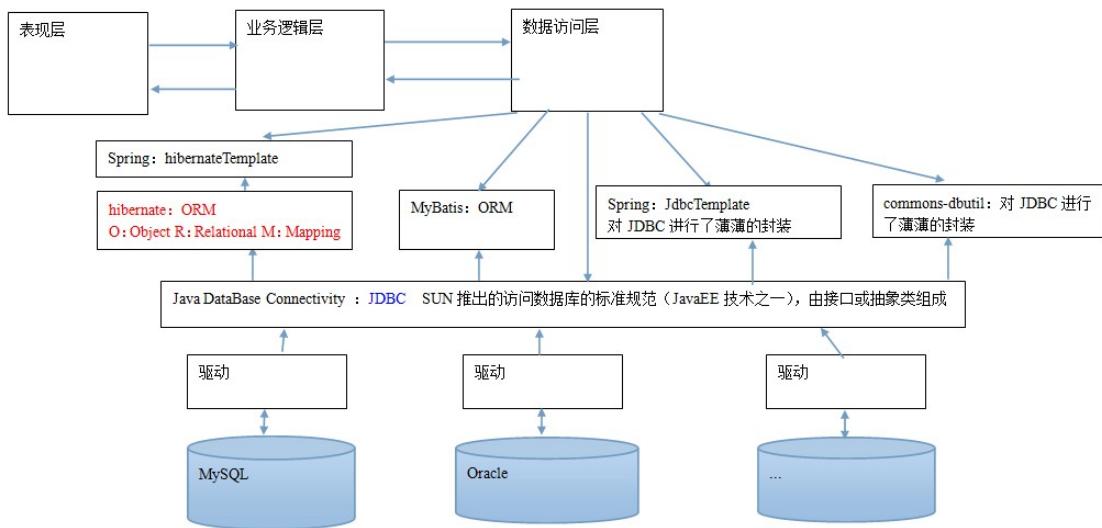
1  @Pointcut("execution(public * com.learn.service.impl.*.*(..))")
2  private void pt1() {}
3
4  @Before("pt1()")
5  public void beforePrintLog() {
6      System.out.println("前置通知 Logger类中的before print Log方法开始记录日志
7  了...");
}

```

JdbcTemplate

它是 spring 框架中提供的一个对象，是对原始 Jdbc API 对象的简单封装。spring 框架为我们提供了很多的 操作模板类。

持久层总图



依赖

- spring-jdbc-xxx.jar
- 数据库驱动

事务：

- spring-tx-xxx.jar

数据源

Spring的内置数据源

DriverManagerDataSource

无需额外依赖

硬编码

```
1 DriverManagerDataSource ds = new DriverManagerDataSource();
2 ds.setDriverClassName("com.mysql.cj.jdbc.Driver");
3 ds.setUrl("jdbc:mysql://localhost:3306/db");
4 ds.setUsername("root");
5 ds.setPassword("xxxx");
```

```
1 //1. 创建JdbcTemplate对象
2 JdbcTemplate jt = new JdbcTemplate(); //这里传ds也一样
3 //1.1给jt设置数据源
4 jt.setDataSource(ds);
5 //2. 执行操作
6 jt.execute("insert into account(name, money) values('ccc', 1000)");
```

xml配置

```
1 <!-- 配置JdbcTemplate -->
2 <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
3   <property name="dataSource" ref="dataSource"></property>
4 </bean>
5
6 <!-- 配置数据源 -->
7 <bean id="dataSource"
8   class="org.springframework.jdbc.datasource.DriverManagerDataSource">
9   <property name="driverClassName" value="com.mysql.jdbc.Driver">
10  </property>
11  <property name="url" value="jdbc:mysql://localhost:3306/db"></property>
12  <property name="username" value="root"></property>
13  <property name="password" value="xxxx"></property>
14 </bean>
```

```
1 //1. 获取容器
2 ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
3 //2. 获取对象
4 JdbcTemplate jt = context.getBean("jdbcTemplate", JdbcTemplate.class);
5 //3. 执行操作
6 jt.execute("insert into account(name, money) values('xxx', 2000)");
```

C3P0

- c3p0-xxx.jar

```

1 <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
2   <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
3   <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/db">
</property>
4   <property name="user" value="root"></property>
5   <property name="password" value="xxxx"></property>
6 </bean>

```

DBCP

- commons-dbcp.jar
- commons-pool.jar

```

1 <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
2   <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
3   <property name="url" value="jdbc:mysql://localhost:3306/db"></property>
4   <property name="username" value="root"></property>
5   <property name="password" value="xxxx"></property>
6 </bean>

```

Druid

- druid

```

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   xsi:schemaLocation="http://www.springframework.org/schema/beans
5   http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7   <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
8     <property name="username" value="root"></property>
9     <property name="password" value="123456"></property>
10    <property name="url" value="jdbc:mysql://localhost:3306/demo">
</property>
11    <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
12  </bean>
</beans>

```

实体类

- 必须实现序列化接口吗？

对象封装

以前是dbutils提供的。

RowMapper

需实现类来定义封装策略。

```
1  /**
2  * JdbcTemplate的CRUD操作
3  */
4  public class JdbcTemplateDemo3 {
5
6      public static void main(String[] args) {
7          //1. 获取容器
8          ApplicationContext context = new
9          ClassPathXmlApplicationContext("bean.xml");
10         //2. 获取对象
11         JdbcTemplate jt = context.getBean("jdbcTemplate",
12             JdbcTemplate.class);
13         //3. 执行操作
14         //3.4查询所有
15         // AccountRowMapper
16         List<Account> accounts = jt.query("select * from account where
17         money > ?",
18                         new AccountRowMapper(), 1000f);
19     }
20 }
21 /**
22 * 定义Account的封装策略
23 */
24 class AccountRowMapper implements RowMapper<Account> {
25     /**
26     * 把结果集中的数据封装到Account中，然后由Spring把每个Account加到集合中
27     * @param resultSet
28     * @param i
29     * @return
30     * @throws SQLException
31     */
32     public Account mapRow(ResultSet resultSet, int i) throws SQLException {
33         Account account = new Account();
34         account.setId(resultSet.getInt("id"));
35         account.setName(resultSet.getString("name"));
36         account.setMoney(resultSet.getFloat("money"));
37         return account;
38     }
39 }
```

BeanPropertyRowMapper

```
1 //BeanPropertyRowMapper 可以返回对象或 集合
2 List<Account> accounts = jt.query("select * from account where money > ?",
3                                     new BeanPropertyRowMapper<Account>
4                                     (Account.class),
5                                     1000f);
6
7 //查询一个
8 List<Account> accounts1 = jt.query("select * from account where id = ?",
9                                     new BeanPropertyRowMapper<Account>
10                                    (Account.class),
11                                    1);
12 System.out.println(accounts1.isEmpty() ? "没有内容" : accounts1.get(0));
```

ResultSetExtractor

方法

update

query

queryForObject

Spring 事务控制

Spring提供了 事务管理器，我们直接用即可。

- JavaEE 体系进行分层开发，事务处理 位于 **业务层**，Spring 提供了分层设计 业务层的事务处理解决方案。
- spring 框架为我们提供了一组 事务控制 的接口。具体在后面的第二小节介绍。
 - 这组接口是在spring-tx-5.0.2.RELEASE.jar 中。

- spring 的事务控制都是 **基于 AOP** 的，它既可以使用编程的方式实现，也可以使用配置的方式实现。我们学习的重点是使用配置的方式实现。
 - 个人认为，事务就是一串包裹 执行逻辑的代码。这样配置只需
 - 包 哪，定下哪个方法即可
 - 包 的细节

API

PlatformTransactionManager

此接口是 spring 的 **事务管理器**，它里面提供了我们常用的操作事务的方法。

- 获取事务状态信息

- 1 | `TransactionStatus getTransaction(TransactionDefinition definition)`

- 提交事务

- 1 | `void commit(TransactionStatus status)`

- 回滚事务

- 1 | `void rollback(TransactionStatus status)`

实现类

DataSourceTransactionManager

使用 SpringJDBC 或或 iBatis 进行持久化数据时 使用。

HibernateTransactionManager

使用 Hibernate 进行持久化数据时使用。

TransactionDefinition

它是事务的 **定义信息** 对象：

- 获得 事务对象名称

- 1 | `String getName()`

- 获得 事务隔离级别

- 1 | `int getIsolationLevel()`

- 获取 事务传播行为

- 1 | `int getPropagationBehavior()`

- 获取 事务超时时间

- 1 | `int getTimeout()`

- 获取 事务是否只读

- 1 | `boolean isReadOnly()`

- 读写型事务：增加、删除、修改 - 开启事务

- 只读型事务：执行查询时，也会开启事务

事务的隔离级别

事务隔离级别 反映 事务提交并发访问时的处理态度

- ISOLATION_DEFAULT
 - 默认级别，归属下列某一种
- ISOLATION_READ_UNCOMMITTED
 - 可以读取未提交数据
- ISOLATION_READ_COMMITTED
 - 只能读取已提交数据，解决脏读问题(Oracle默认级别)
- ISOLATION_REPEATABLE_READ
 - 是否读取其他事务提交修改后的数据，解决不可重复读问题(MySQL默认级别)
- ISOLATION_SERIALIZABLE
 - 是否读取其他事务提交添加后的数据，解决幻读问题

事务的传播行为

？？？

- REQUIRED
 - 如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。一般的选择（默认值）
- SUPPORTS
 - 支持当前事务，如果当前没有事务，就以非事务方式执行（没有事务）
- MANDATORY
 - 使用当前的事务，如果当前没有事务，就抛出异常
- REQUIRES_NEW
 - 新建事务，如果当前在事务中，把当前事务挂起。
- NOT_SUPPORTED
 - 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起
- NEVER
 - 以非事务方式运行，如果当前存在事务，抛出异常

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

超时时间

默认值是-1，没有超时限制。如果有，以秒为单位进行设置。

是否是只读事务

建议查询时设置为只读。

TransactionStatus

此接口提供的是事务 **具体的运行状态**。TransactionStatus接口描述了某个时间点上事务对象的状态信息。

- 刷新事务
 - 1 | `void flush()`
- 获取是否存在 存储点
 - 1 | `boolean hasSavepoint()`
- 获取事务是否完成
 - 1 | `boolean isCompleted()`
- 获取事务是否为新的事务
 - 1 | `boolean isNewTransaction()`
- 获取事务是否回滚
 - 1 | `boolean isRollbackOnly()`
- 设置事务回滚
 - 1 | `void setRollbackOnly()`

XML 事务

- 管理器
- 通知
 - 事务属性

- 切入点表达式
- 对应关系

Spring中基于XML的 声明式事务控制 配置步骤：

1. 配置 事务管理器

2. 配置 事务的通知

- 需要导入事务的约束 tx名称空间和约束，同时也需要 aop的使用

- **tx:advice** 标签配置 事务通知：

- id : 给事务通知起一个唯一标识
- transaction-manager : 给事务通知提供一个事务管理器引用

3. 配置AOP中的通用 切入点表达式

4. 建立事务通知和切入点表达式的对应关系

- aop:advisor标签，在aop:config内部使用

- advice-ref
- pointcut-ref

- aop:aspect标签，在aop:config内部使用

- aop:before
- aop:after-returning
- aop:after-throwing
- aop:after
- aop:around

5. 配置 事务的属性 (是在事务的通知 tx:advice标签的内部)

- isolation : 指定事务的隔离级别。默认值是DEFAULT，表示使用数据库的默认隔离级别
- no-rollback-for : 用于指定一个异常，当产生该异常时，事务不回滚。没有默认值，表示任何异常都回滚。
- propagation : 用于指定事务的传播行为。默认值是REQUIRED，表示一定会有事务。增删改的选择。查询方法可以选择SUPPORTS。
- read-only : 用于指定事务是否只读。只有查询方法才能设置为true。默认值是false，表示读写
- rollback-for : 用于指定一个异常，当产生该异常时，事务回滚，产生其他异常时，事务不回滚。没有默认值。表示任何异常都回滚。
- timeout : 用于指定事务的超时时间，默认值是-1，表示永不超时。如果指定了数值，以秒为单位。

```

1 <beans xmlns="http://www.springframework.org/schema/beans"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3
4   xmlns:aop="http://www.springframework.org/schema/aop"
5   xmlns:tx="http://www.springframework.org/schema/tx"
6
7   xsi:schemaLocation="http://www.springframework.org/schema/beans
8     http://www.springframework.org/schema/beans/spring-beans.xsd
9       http://www.springframework.org/schema/aop
10      http://www.springframework.org/schema/aop/spring-aop.xsd
11      http://www.springframework.org/schema/tx

```

```

12          http://www.springframework.org/schema/tx/spring-
13          tx.xsd">
14
15      <!-- 配置事务管理器 -->
16      <bean id="transactionManager"
17          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
18          <property name="dataSource" ref="dataSource"></property>
19      </bean>
20
21      <!-- 配置事务的通知 -->
22      <tx:advice id="txAdvice" transaction-manager="transactionManager">
23          <!-- 配置事务的属性 -->
24          <tx:attributes>
25              <tx:method name="*" propagation="REQUIRED" read-only="false"/>
26              <tx:method name="find*" propagation="SUPPORTS" read-only="true">
27          </tx:method>
28          </tx:attributes>
29      </tx:advice>
30
31      <!-- 配置aop -->
32      <aop:config>
33          <!-- 配置切入点表达式 -->
34          <aop:pointcut id="pt1" expression="execution(*
com.learn.service.impl.*.*(..))"/>
            <!-- 建立切入点表达式和事务通知的对应关系 -->
            <aop:advisor advice-ref="txAdvice" pointcut-ref="pt1"></aop:advisor>
        </aop:config>

```

pom依赖

- aop : aspectjweaver
- spring-tx
- spring-jdbc
- mysql-connector-java

配置文件 & 约束

配置文件

- 事务管理器
- 事务通知
- aop

```

1      <!-- 配置事务管理器 -->
2      <bean id="transactionManager"
3          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
4          <property name="dataSource" ref="dataSource"></property>
5      </bean>
6
7      <!-- 配置事务的通知 -->
8      <!-- 事务通知transactionManager -->
9      <tx:advice id="txAdvice" transaction-manager="transactionManager">

```

```

9   <!-- 配置事务的属性 -->
10  <!-- 切入点后 的筛选,按方法名筛选 -->
11  <tx:attributes>
12      <tx:method name="*" propagation="REQUIRED" read-only="false"/>
13      <tx:method name="find*" propagation="SUPPORTS" read-only="true">
14  </tx:method>
15  </tx:attributes>
16 </tx:advice>
17
18 <!-- 配置aop -->
19 <aop:config>
20     <!-- 配置切入点表达式 -->
21     <aop:pointcut id="pt1" expression="execution(*
22         com.learn.service.impl.*.*(..))"/>
23     <!-- 建立切入点表达式和事务通知的对应关系 -->
24     <!-- 通知加到哪 -->
25     <aop:advisor advice-ref="txAdvice" pointcut-ref="pt1"></aop:advisor>
26 </aop:config>

```

约束

导入aop和tx两个名称空间。

- aop
- tx

```

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:aop="http://www.springframework.org/schema/aop"
5      xmlns:tx="http://www.springframework.org/schema/tx"
6
7      xsi:schemaLocation="http://www.springframework.org/schema/beans
8          http://www.springframework.org/schema/beans/spring-beans.xsd
9          http://www.springframework.org/schema/aop
10         http://www.springframework.org/schema/aop/spring-aop.xsd
11         http://www.springframework.org/schema/tx
12         http://www.springframework.org/schema/tx/spring-
tx.xsd">

```

标签&属性

- 管理器 : bean , DataSourceTransactionManager
- 通知 : tx:advice
 - 事务属性 : tx:attributes 、 tx:method
- 切入点表达式 : aop:config/aop:pointcut
- 对应关系 : aop:advisor 、 aop:before...

bean

```
1 <!-- 配置事务管理器 -->
2 <bean id="transactionManager"
3   class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
4     <property name="dataSource" ref="dataSource"></property>
5   </bean>
```

事务的通知

- 可以配置 将事务 加到 哪些方法上

tx:advice

- id
- transaction-manager

```
1 <!-- 配置事务的通知 -->
2 <tx:advice id="txAdvice" transaction-manager="transactionManager">
3   <!-- 配置事务的属性 -->
4   <tx:attributes>
5     <tx:method name="*" propagation="REQUIRED" read-only="false"/>
6     <tx:method name="find*" propagation="SUPPORTS" read-only="true">
7   </tx:method>
8   </tx:attributes>
9 </tx:advice>
```

tx:attributes

```
1 <!-- 配置事务的通知 -->
2 <tx:advice id="txAdvice" transaction-manager="transactionManager">
3   <!-- 配置事务的属性 -->
4   <tx:attributes>
5     <tx:method name="*" propagation="REQUIRED" read-only="false"/>
6     <tx:method name="find*" propagation="SUPPORTS" read-only="true">
7   </tx:method>
8   </tx:attributes>
9 </tx:advice>
```

tx:method

- name
- propagation
- read-only

```
1 <!-- 配置事务的通知 -->
2 <tx:advice id="txAdvice" transaction-manager="transactionManager">
3     <!-- 配置事务的属性 -->
4     <tx:attributes>
5         <tx:method name="*" propagation="REQUIRED" read-only="false"/>
6         <tx:method name="find*" propagation="SUPPORTS" read-only="true">
7     </tx:method>
8 </tx:attributes>
9 </tx:advice>
```

aop配置

- 切入点表达式：把事务加到哪些类|方法上
- 配置切入点表达式和事务通知的对应关系

aop:config

aop配置

```
1 <!-- 配置aop -->
2 <aop:config>
3     <!-- 配置切入点表达式 -->
4     <aop:pointcut id="pt1" expression="execution(* com.learn.service.impl.*.*(..))"/>
5     <!-- 建立切入点表达式和事务通知的对应关系 -->
6     <aop:advisor advice-ref="txAdvice" pointcut-ref="pt1"></aop:advisor>
7 </aop:config>
```

aop:pointcut

配置切入点表达式

```
1 <!-- 配置切入点表达式 -->
2 <aop:pointcut id="pt1" expression="execution(* com.learn.service.impl.*.*(..))"/>
```

aop:advisor

建立切入点表达式和事务通知的对应关系

```
1 <!-- 建立切入点表达式和事务通知的对应关系 -->
2 <aop:advisor advice-ref="txAdvice" pointcut-ref="pt1"></aop:advisor>
```

注解 事务

- 管理器
- 通知
 - 事务属性
- 切入点表达式
- 对应关系

pom依赖

- aop : aspectjweaver
- spring-tx
- spring-jdbc
- mysql-connector-java

配置文件&约束

配置文件

xml

- 配置spring创建容器时要扫描的包
- 配置事务管理器
- 开启spring对注解事务的支持

```

1 <!-- 配置spring创建容器时要扫描的包-->
2 <context:component-scan base-package="com.learn"></context:component-scan>
3
4 <!-- 配置事务管理器 -->
5 <bean id="transactionManager"
6   class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
7   <property name="dataSource" ref="dataSource"></property>
8 </bean>
9 <!-- 开启spring对注解事务的支持-->
10 <tx:annotation-driven transaction-manager="transactionManager">
</tx:annotation-driven>
```

注解

- SpringConfiguration
- JdbcConfig
 - jdbcConfig.properties
- TransactionConfig

SpringConfiguration

```
1 /**
2  * Spring的配置类
3  * 相当于bean.xml
4 */
5 @Configuration
6 @ComponentScan("com.learn")
7 @Import({JdbcConfig.class, TransactionConfig.class})
8 @PropertySource("jdbcConfig.properties")
9 @EnableTransactionManagement//事务注解支持
10 public class SpringConfiguration {
11 }
```

TransactionConfig

```
1 /**
2  * 和事务相关的配置类
3 */
4 public class TransactionConfig {
5 /**
6  * 用于创建事务管理器对象
7  * @param dataSource
8  * @return
9 */
10 @Bean(name = "transactionManager")
11 public PlatformTransactionManager createTransactionManager(DataSource
dataSource) {
12     return new DataSourceTransactionManager(dataSource);
13 }
14 }
```

JdbcConfig

```
1 /**
2  * 和连接数据库相关的配置类
3 */
4 public class JdbcConfig {
5
6     @Value("${jdbc.driver}")
7     private String driver;
8
9     @Value("${jdbc.url}")
10    private String url;
11
12    @Value("${jdbc.username}")
13    private String username;
14
15    @Value("${jdbc.password}")
16    private String password;
17
18 /**
19  * 创建JdbcTemplate对象
20  * @param dataSource
21  * @return
22 */
23 @Bean(name = "jdbcTemplate")
24 public JdbcTemplate createJdbcTemplate(DataSource dataSource) {
```

```
25         return new JdbcTemplate(dataSource);
26     }
27
28     @Bean(name = "dataSource")
29     public DataSource createDataSource() {
30         DriverManagerDataSource ds = new DriverManagerDataSource();
31         ds.setDriverClassName(driver);
32         ds.setUrl(url);
33         ds.setUsername(username);
34         ds.setPassword(password);
35         return ds;
36     }
37 }
```

jdbcConfig.properties

```
1 jdbc.driver=com.mysql.cj.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/db
3 jdbc.username=root
4 jdbc.password=xxxx
```

约束

- context
- tx

注解

- 管理器
- 通知
 - 事务属性
- 切入点表达式
- 对应关系

@EnableTransactionManagement

事务注解支持

```
1 /**
2  * Spring的配置类
3  * 相当于bean.xml
4 */
5 @Configuration
6 @ComponentScan("com.learn")
7 @Import({JdbcConfig.class, TransactionConfig.class})
8 @PropertySource("jdbcConfig.properties")
9 @EnableTransactionManagement//事务注解支持
10 public class SpringConfiguration {
11 }
```

管理器

@Bean

```
1  /**
2  * 和事务相关的配置类
3  */
4  public class TransactionConfig {
5      /**
6      * 用于创建事务管理器对象
7      * @param dataSource
8      * @return
9      */
10     @Bean(name = "transactionManager")
11     public PlatformTransactionManager createTransactionManager(DataSource
dataSource) {
12         return new DataSourceTransactionManager(dataSource);
13     }
14 }
```

对应关系

@Transactional

该注解的属性和 xml 中的属性含义一致。该注解可以出现在接口上，类上和方法上。

- 出现接口上，表示该接口的所有实现类都有事务支持。
- 出现在类上，表示类中所有方法有事务支持
- 出现在方法上，表示方法有事务支持。
- 以上三个位置的优先级：方法>类>接口

```
1  @Service("accountService")
2  @Transactional(propagation= Propagation.SUPPORTS, readOnly=true)//只读型事务的
配置
3  public class AccountServiceImpl implements AccountService {
4
5      @Autowired
6      private AccountDao accountDao;
7
8      //只读型事务的配置
9      public Account findAccountById(Integer accountId) {
10         return accountDao.findAccountById(accountId);
11     }
12
13     //需要的是读写型事务配置
14     @Transactional(propagation= Propagation.REQUIRED, readOnly=false)
15     public void transfer(String sourceName, String targetName, Float money)
{
16         System.out.println("transfer....");
17         //2.1根据名称查询转出账户
18         Account source = accountDao.findAccountByName(sourceName);
19         //2.2根据名称查询转入账户
20 }
```

```
20     Account target = accountDao.findAccountByName(targetName);
21     //2.3转出账户减钱
22     source.setMoney(source.getMoney()-money);
23     //2.4转入账户加钱
24     target.setMoney(target.getMoney()+money);
25     //2.5更新转出账户
26     accountDao.updateAccount(source);
27
28     int i=1/0;
29
30     //2.6更新转入账户
31     accountDao.updateAccount(target);
32 }
33 }
```

案例

简单Demo

xml方式配置druid连接池

只要是new对象的过程，坦白来说都可以交由spring容器来控制。

第三方jar包，当然也可以。

在Spring中，很多对象都是单实例的，在日常的开发中，我们经常需要使用某些外部的单实例对象，例如数据库连接池，下面我们来讲解下如何在spring中创建第三方bean实例。

pom

```
1 <!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
2 <dependency>
3   <groupId>com.alibaba</groupId>
4   <artifactId>druid</artifactId>
5   <version>1.1.21</version>
6 </dependency>
7 <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
8 <dependency>
9   <groupId>mysql</groupId>
10  <artifactId>mysql-connector-java</artifactId>
11  <version>5.1.47</version>
12 </dependency>
```

xml

```
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   xsi:schemaLocation="http://www.springframework.org/schema/beans
5   http://www.springframework.org/schema/beans/spring-beans.xsd">
6 
7   <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
8     <property name="username" value="root"></property>
9     <property name="password" value="123456"></property>
10    <property name="url" value="jdbc:mysql://localhost:3306/demo">
11    </property>
12  </bean>
13 </beans>
```

java

```
1 public class MyTest {
2   public static void main(String[] args) throws SQLException {
3     ApplicationContext context = new
4     ClassPathXmlApplicationContext("ioc3.xml");
5     DruidDataSource dataSource = context.getBean("dataSource",
6     DruidDataSource.class);
7     System.out.println(dataSource);
8     System.out.println(dataSource.getConnection());
9   }
10 }
```

AOP入门Demo

通知类Bean : Logger类

AOP配置 :

- aop:config : 开始AOP配置
 - aop:aspect : 配置切面
 - id : 给切面提供一个唯一标识
 - ref : 指定 **通知类bean** 的 Id
- ↓在aop:aspect内部使用对应的标签来配置通知的类型↓
- aop:before : 前置通知
 - method : 用于指定 通知类bean中 **哪个方法** 是 前置通知
 - pointcut : 用于指定**切入点表达式** , 该表达式的含义指的是对业务层中哪些方法增强。

pom

```
1 <!-- https://mvnrepository.com/artifact/org.springframework/spring-context
2 -->
3 <dependency>
4     <groupId>org.springframework</groupId>
5     <artifactId>spring-context</artifactId>
6     <version>5.2.5.RELEASE</version>
7 </dependency>
8
9 <!-- 解析切入点表达式 -->
10 <dependency>
11     <groupId>org.aspectj</groupId>
12     <artifactId>aspectjweaver</artifactId>
13     <version>1.8.2</version>
14 </dependency>
```

bean.xml

```
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:aop="http://www.springframework.org/schema/aop"
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/aop
8                         http://www.springframework.org/schema/aop/spring-aop.xsd
9         ">
10        <!-- 配置Spring的Ioc,把service对象配置进来 -->
11        <!-- 需要增强,加个日志 -->
12        <bean id="accountService"
13              class="com.learn.service.impl.AccountServiceImpl">
14        </bean>
15
16
17        <!-- 配置Logger类 -->
18        <!-- 写一个日志的通知 -->
19        <bean id="logger"
20              class="com.learn.utils.Logger">
21        </bean>
22
23
24        <!-- 配置AOP -->
25        <aop:config>
26            <!-- 配置切面 引用通知 -->
27            <aop:aspect id="logAdvice" ref="logger">
28                <!-- 配置通知的类型,并且建立 通知方法 和 切入点方法 的关联 -->
29                <aop:before method="printLog"
30                            pointcut="execution(public void
31 com.learn.service.impl.AccountServiceImpl.saveAccount())"></aop:before>
32            </aop:aspect>
33        </aop:config>
34    </beans>
```

service

AccountService

```
1  /**
2  * 账户的业务层接口
3  */
4  public interface AccountService {
5
6      /**
7      * 模拟保存账户
8      */
9      void saveAccount();
10
11     /**
12     * 模拟更新账户
13     * @param i
14     */
15     void updateAccount(int i);
16
17     /**
18     * 删除账户
19     * @return
20     */
21     int deleteAccount();
22
23 }
```

AccountServiceImpl

```
1  /**
2  * 账户的业务层实现类
3  */
4  public class AccountServiceImpl implements AccountService {
5
6
7      public void saveAccount() {
8          System.out.println("执行了保存");
9      }
10
11     public void updateAccount(int i) {
12         System.out.println("执行了更新");
13     }
14
15     public int deleteAccount() {
16         System.out.println("执行了删除");
17         return 0;
18     }
19 }
```

utils

Logger

```
1  /**
2  * 用于记录日志的工具类,它里面提供了公共的代码
3  */
4  public class Logger {
5
6      /**
7      * 用于打印日志:并且计划在其切入点方法执行之前执行
8      *
9      *          切入点方法就是业务层方法
10     */
11    public void printLog() {
12        System.out.println("Logger类中的printLog方法开始记录日志了...");  

13    }
14 }
```

test

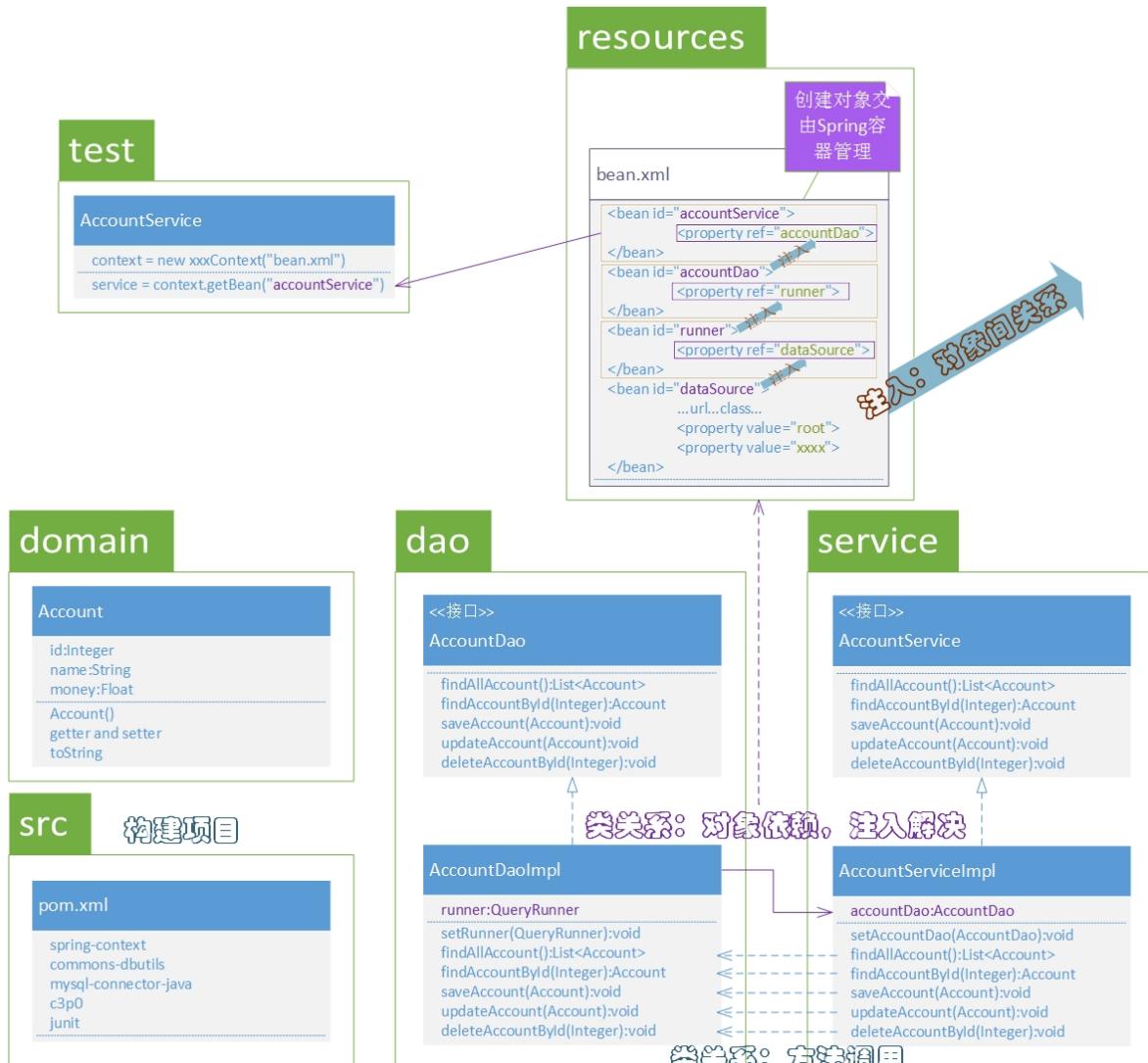
AOPTest

```
1 /**
2 * 测试AOP的配置
3 */
4 public class AOPTest {
5
6     public static void main(String[] args) {
7         //1.获取容器
8         ApplicationContext context = new
9         ClassPathXmlApplicationContext("bean.xml");
10        //2.获取对象
11        AccountService service = (AccountService)
12        context.getBean("accountService");
13        //3.执行方法
14        service.saveAccount();
15        service.deleteAccount();
16    }
17 }
```

Spring实现数据库的CRUD

QueryRunner

XML实现



pom

- spring-context
- commons-dbutils
- mysql-connector-java
- c3p0
- junit

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5          http://maven.apache.org/xsd/maven-4.0.0.xsd">
6      <modelVersion>4.0.0</modelVersion>
7
8      <groupId>com.ninthat</groupId>
9      <artifactId>maven001</artifactId>
10     <version>1.0-SNAPSHOT</version>
11     <packaging>jar</packaging>
12
13     <dependencies>
14         <!-- https://mvnrepository.com/artifact/org.springframework/spring-
15             context -->
16         <dependency>
17             <groupId>org.springframework</groupId>

```

```

16      <artifactId>spring-context</artifactId>
17      <version>5.2.5.RELEASE</version>
18  </dependency>
19
20      <!-- https://mvnrepository.com/artifact/commons-dbutils/commons-
21 dbutils -->
22  <dependency>
23      <groupId>commons-dbutils</groupId>
24      <artifactId>commons-dbutils</artifactId>
25      <version>1.5</version>
26  </dependency>
27
28  <dependency>
29      <groupId>mysql</groupId>
30      <artifactId>mysql-connector-java</artifactId>
31      <version>8.0.15</version>
32  </dependency>
33
34      <!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->
35  <dependency>
36      <groupId>com.mchange</groupId>
37      <artifactId>c3p0</artifactId>
38      <version>0.9.5.2</version>
39  </dependency>
40
41  <dependency>
42      <groupId>junit</groupId>
43      <artifactId>junit</artifactId>
44      <version>4.12</version>
45  </dependency>
46 </dependencies>
47 </project>

```

spring xml

```

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         xsi:schemaLocation="http://www.springframework.org/schema/beans
5                         http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <!-- *****把对象的创建交给spring来管理***** -->
8
9     <bean id="accountService"
10        class="com.learn.service.impl.AccountServiceImpl">
11         <!-- 注入dao -->
12         <!-- accountService.setXxx() -->
13         <property name="accountDao" ref="accountDao"></property>
14     </bean>
15
16     <!-- 配置Dao对象 -->
17     <bean id="accountDao" class="com.learn.dao.impl.AccountDaoImpl">
18         <!-- 注入QueryRunner -->
19         <!-- accountDao.setXxx() -->
20         <property name="runner" ref="runner"></property>

```

```

19    </bean>
20
21    <!-- 配置QueryRunner对象,我们是单表,一条语句的CRUD操作,我们可以选择传入数据源|连接池 -->
22    <!-- 这里如果是单例的话,会出现线程干扰问题 -->
23    <bean id="runner" class="org.apache.commons.dbutils.QueryRunner"
24        scope="prototype">
25        <!-- 注入数据源 -->
26        <!-- QueryRunner没有set方法,只能使用 构造函数注入 -->
27        <!-- new QueryRunner(ds) -->
28        <constructor-arg name="ds" ref="dataSource"></constructor-arg>
29    </bean>
30
31    <!-- 配置数据源 -->
32    <bean id="dataSource"
33        class="com.mchange.v2.c3p0.ComboPooledDataSource">
34        <!-- 注入 连接数据库的必备信息 -->
35        <!-- dataSource.setXXX() 先默认构造,然后set注入 -->
36        <property name="driverClass" value="com.mysql.cj.jdbc.Driver">
37        <!--注入驱动类-->
38        <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/db">
39        <!--注入url-->
39        <property name="user" value="root"></property>
37        <property name="password" value="xxxx"></property>
38    </bean>
39 </beans>

```

java

```

1 /**
2  * 账户的实体类
3 */
4 public class Account {
5
6     // -----变量-----
7
8     private Integer id;
9     private String name;
10    private Float money;
11
12    // -----constructor-----
13
14 /**
15  * 默认构造方法
16 */
17    public Account() {
18    }
19
20 /**
21  *
22  * @param id

```

```
23     * @param name
24     * @param money
25     */
26     public Account(Integer id, String name, Float money) {
27         this.id = id;
28         this.name = name;
29         this.money = money;
30     }
31
32     // -----getter and setter-----
33
34     public Integer getId() {
35         return id;
36     }
37
38     public void setId(Integer id) {
39         this.id = id;
40     }
41
42     public String getName() {
43         return name;
44     }
45
46     public void setName(String name) {
47         this.name = name;
48     }
49
50     public Float getMoney() {
51         return money;
52     }
53
54     public void setMoney(Float money) {
55         this.money = money;
56     }
57
58     // -----toString -----
59
60
61     @Override
62     public String toString() {
63         return "Account{" +
64             "id=" + id +
65             ", name='" + name + '\'' +
66             ", money=" + money +
67             '}';
68     }
69 }
```

```
1 /**
2  * 账户的持久层接口
3 */
```

```
4 public interface AccountDao {  
5  
6     /**  
7      * 查询所有  
8      * @return  
9      */  
10    List<Account> findAllAccount();  
11  
12    /**  
13     * 查询一个  
14     * @return  
15     */  
16    Account findAccountById(Integer accountId);  
17  
18    /**  
19     * 保存  
20     * @param account  
21     */  
22    void saveAccount(Account account);  
23  
24    /**  
25     * 更新  
26     * @param account  
27     */  
28    void updateAccount(Account account);  
29  
30    /**  
31     * 删除  
32     * @param accountId  
33     */  
34    void deleteAccountById(Integer accountId);  
35 }
```

```
1  /**  
2   * 账户的持久层实现类  
3   */  
4  public class AccountDaoImpl implements AccountDao {  
5  
6      private QueryRunner runner;  
7  
8      /**  
9       * 需要Spring提供注入,我们提供set方法,这里是xml配置  
10      * 注解配置没set也行  
11      */  
12      public void setRunner(QueryRunner runner) {  
13          this.runner = runner;  
14      }  
15  
16      public List<Account> findAllAccount() {  
17          try {  
18              return runner.query("select * from account", new  
19          BeanListHandler<Account>(Account.class));  
20          } catch (SQLException e) {  
21              throw new RuntimeException(e);  
22          }  
23      }
```

```

22 }
23
24     public Account findAccountById(Integer accountId) {
25         try {
26             return runner.query("select * from account where id=?", new
27 BeanHandler<Account>(Account.class), accountId);
28         } catch (SQLException e) {
29             throw new RuntimeException(e);
30         }
31     }
32
33     public void saveAccount(Account account) {
34         try {
35             runner.update("insert into account(name,money) values(?,?)",
36 account.getName(), account.getMoney());
37         } catch (SQLException e) {
38             throw new RuntimeException(e);
39         }
40     }
41
42     public void updateAccount(Account account) {
43         try {
44             runner.update("update account set name=?, money=? where id=?",
45 account.getName(), account.getMoney(), account.getId());
46         } catch (SQLException e) {
47             throw new RuntimeException(e);
48         }
49     }
50
51     public void deleteAccoutn(Integer accountId) {
52         try {
53             runner.update("delete from account where id=?", accountId);
54         } catch (SQLException e) {
55             throw new RuntimeException(e);
56         }
57     }

```

```

1 /**
2  * 账户的业务层接口
3 */
4 public interface AccountService {
5
6     /**
7      * 查询所有
8      * @return
9      */
10    List<Account> findAllAccount();
11
12    /**
13     * 查询一个
14     * @return
15     */
16    Account findAccountById(Integer accountId);
17

```

```
18     /**
19      * 保存
20      * @param account
21      */
22     void saveAccount(Account account);
23
24     /**
25      * 更新
26      * @param account
27      */
28     void updateAccount(Account account);
29
30     /**
31      * 删除
32      * @param accountId
33      */
34     void deleteAccountById(Integer accountId);
35 }
```

```
1 /**
2  * 账户的业务层实现类
3 */
4 public class AccountServiceImpl implements AccountService {
5
6     private AccountDao accountDao;
7
8     /**
9      * 需要Spring提供注入,我们提供set方法,这里是xml配置
10     * 注解配置没set也行
11     * @param accountDao
12     */
13     public void setAccountDao(AccountDao accountDao) {
14         this.accountDao = accountDao;
15     }
16
17
18     public List<Account> findAllAccount() {
19         return accountDao.findAllAccount();
20     }
21
22     public Account findAccountById(Integer accountId) {
23         return accountDao.findAccountById(accountId);
24     }
25
26     public void saveAccount(Account account) {
27         accountDao.saveAccount(account);
28     }
29
30     public void updateAccount(Account account) {
31         accountDao.updateAccount(account);
32     }
33
34     public void deleteAccount(Integer accountId) {
35         accountDao.deleteAccounn(accountId);
36     }
}
```

```
37  
38 }
```

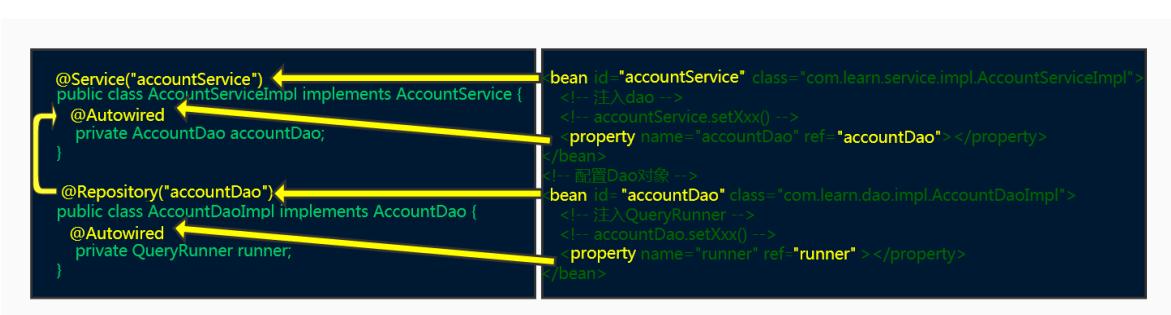
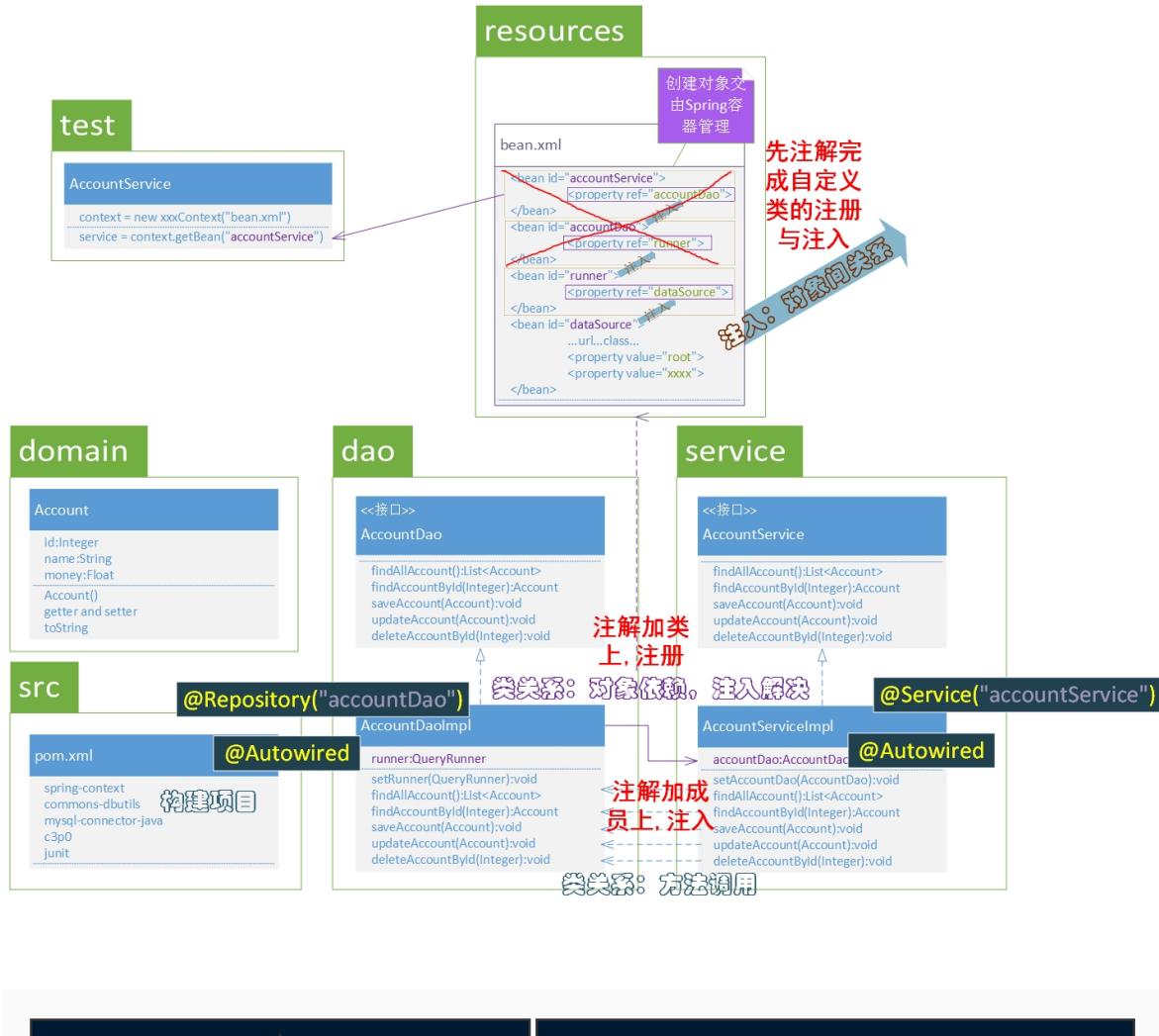
```
1 /**
2  * 使用JUnit单元测试,测试我们的配置
3 */
4 public class AccountServiceTest {
5
6     @Test
7     public void testFindAll() {
8         //1.获取容器
9         ApplicationContext context = new
ClassPathXmlApplicationContext("bean.xml");
10        //2.得到业务层对象
11        AccountService service = context.getBean("accountService",
AccountService.class);
12        //3.执行方法
13        List<Account> accounts = service.findAllAccount();
14        for (Account account : accounts) {
15            System.out.println(account);
16        }
17    }
18
19    @Test
20    public void testFindOne() {
21        //1.获取容器
22        ApplicationContext context = new
ClassPathXmlApplicationContext("bean.xml");
23        //2.得到业务层对象
24        AccountService service = context.getBean("accountService",
AccountService.class);
25        //3.执行方法
26        Account account = service.findAccountById(1);
27        System.out.println(account);
28    }
29
30    @Test
31    public void testSave() {
32        //1.获取容器
33        ApplicationContext context = new
ClassPathXmlApplicationContext("bean.xml");
34        //2.得到业务层对象
35        AccountService service = context.getBean("accountService",
AccountService.class);
36        //3.执行方法
37        Account account = new Account(null, "呵呵", 1000F);
38        service.saveAccount(account);
39    }
40
41    @Test
42    public void testUpdate() {
43        //1.获取容器
44        ApplicationContext context = new
ClassPathXmlApplicationContext("bean.xml");
45        //2.得到业务层对象
```

```
46     AccountService service = context.getBean("accountService",
47     AccountService.class);
48     //3.执行方法
49     Account account = service.findAccountById(3);
50     account.setMoney(10000F);
51     service.updateAccount(account);
52 }
53
54 @Test
55 public void testDelete() {
56     //1.获取容器
57     ApplicationContext context = new
58     ClassPathXmlApplicationContext("bean.xml");
59     //2.得到业务层对象
60     AccountService service = context.getBean("accountService",
61     AccountService.class);
62     //3.执行方法
63     service.deleteAccount(4);
64 }
```

注解实现

- 扫描包：xml|注解 均可实现
- bean注册
 - 自定义类
 - jar包类
- bean注入
 -

在原XML项目上修改



spring xml

- 加入context
 - service dao 的注册及注入交由注解完成，这里注释掉

```
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     <!-- *****注解: 告知spring在创建容器时要扫描的包***** -->
10    <context:component-scan base-package="com.learn"></context:component-
11      scan>
12
13     <!-- *****把对象的创建交给spring来管理***** -->
14     <!-- &lt;!&ndash; 配置Service &ndash;&gt;-->
```

```

14 <!--      <bean id="accountService"
15   class="com.learn.service.impl.AccountServiceImpl">-->
16 <!--          &lt;!&ndash; 注入dao &ndash;&gt;-->
17 <!--          &lt;!&ndash; accountService.setXXX() &ndash;&gt;-->
18 <!--          <property name="accountDao" ref="accountDao"></property>-->
19 <!--      </bean>-->
20
21 <!--          &lt;!&ndash; 配置Dao对象 &ndash;&gt;-->
22 <!--          <bean id="accountDao" class="com.learn.dao.impl.AccountDaoImpl">-->
23 <!--              &lt;!&ndash; 注入QueryRunner &ndash;&gt;-->
24 <!--              &lt;!&ndash; accountDao.setXXX() &ndash;&gt;-->
25 <!--              <property name="runner" ref="runner"></property>-->
26 <!--          </bean>-->
27
28 <!-- 配置QueryRunner对象,我们是单表,一条语句的CRUD操作,我们可以选择传入数据源|连接池 -->
29 <!-- 这里如果是单例的话,会出现线程干扰问题 -->
30 <bean id="runner" class="org.apache.commons.dbutils.QueryRunner"
31 scope="prototype">
32     <!-- 注入数据源 -->
33     <!-- QueryRunner没有set方法,只能使用 构造函数注入 -->
34     <!-- new QueryRunner(ds) -->
35     <constructor-arg name="ds" ref="dataSource"></constructor-arg>
36 </bean>
37
38 <!-- 配置数据源 -->
39 <bean id="dataSource"
40 class="com.mchange.v2.c3p0.ComboPooledDataSource">
41     <!-- 注入 连接数据库的必备信息 -->
42     <!-- dataSource.setXXX() -->
43     <property name="driverClass" value="com.mysql.cj.jdbc.Driver">
44 </property>
45     <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/db">
46 </property>
47     <property name="user" value="root"></property>
48     <property name="password" value="xxxx"></property>
49 </bean>
50 </beans>

```

java

- 注册
- 注入：不需要set方法了，由于是注解注入

```

1 /**
2  * 账户的业务层实现类
3 */
4 @Service("accountService")
5 public class AccountServiceImpl implements AccountService {
6     @Autowired//set方法不再必须
7     private AccountDao accountDao;
8
9 /**
10  * 需要Spring提供注入,我们提供set方法,这里是xml配置
11  * 注解配置没set也行

```

```

12     * @param accountDao
13     */
14 //    public void setAccountDao(AccountDao accountDao) {
15 //        this.accountDao = accountDao;
16 //    }
17
18
19
20    public List<Account> findAllAccount() {
21        return accountDao.findAllAccount();
22    }
23
24    public Account findAccountById(Integer accountId) {
25        return accountDao.findAccountById(accountId);
26    }
27
28    public void saveAccount(Account account) {
29        accountDao.saveAccount(account);
30    }
31
32    public void updateAccount(Account account) {
33        accountDao.updateAccount(account);
34    }
35
36    public void deleteAccount(Integer accountId) {
37        accountDao.deleteAccoun(accountId);
38    }
39
40}

```

```

1 /**
2  * 账户的持久层实现类
3 */
4 @Repository("accountDao")
5 public class AccountDaoImpl implements AccountDao {
6     @Autowired
7     private QueryRunner runner;
8
9     //    public void setRunner(QueryRunner runner) {
10 //        this.runner = runner;
11 //    }
12
13     public List<Account> findAllAccount() {
14         try {
15             return runner.query("select * from account", new
16 BeanListHandler<Account>(Account.class));
17         } catch (SQLException e) {
18             throw new RuntimeException(e);
19         }
20     }
21
22     public Account findAccountById(Integer accountId) {
23         try {

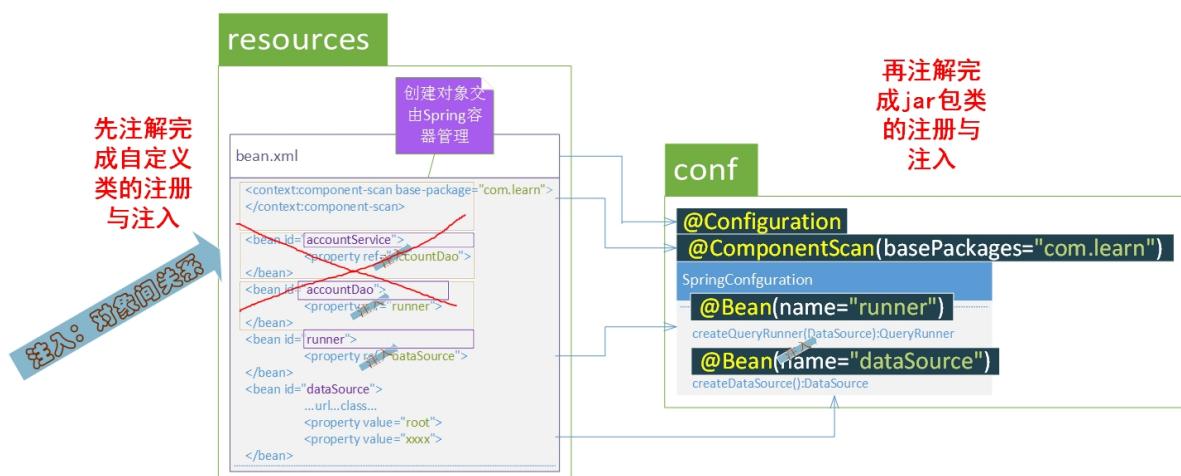
```

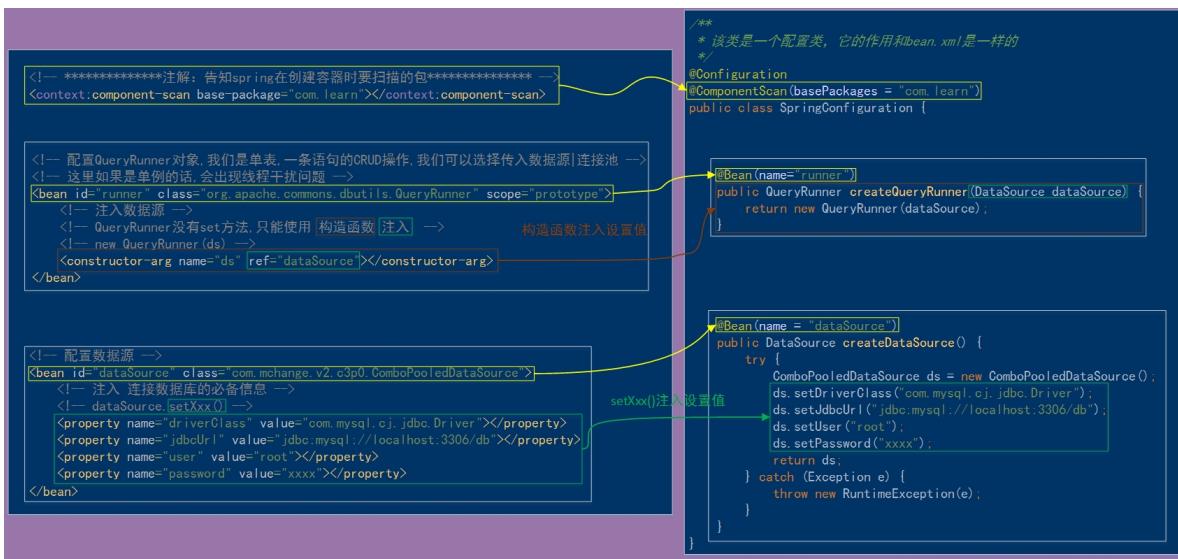
```

23         return runner.query("select * from account where id=?", new
24     BeanHandler<Account>(Account.class), accountId);
25     } catch (SQLException e) {
26         throw new RuntimeException(e);
27     }
28 }
29
30 public void saveAccount(Account account) {
31     try {
32         runner.update("insert into account(name,money) values(?,?)",
33 account.getName(), account.getMoney());
34     } catch (SQLException e) {
35         throw new RuntimeException(e);
36     }
37 }
38
39 public void updateAccount(Account account) {
40     try {
41         runner.update("update account set name=?, money=? where id=?",
42 account.getName(), account.getMoney(), account.getId());
43     } catch (SQLException e) {
44         throw new RuntimeException(e);
45     }
46 }
47
48 public void deleteAccoutn(Integer accountId) {
49     try {
50         runner.update("delete from account where id=?", accountId);
51     } catch (SQLException e) {
52         throw new RuntimeException(e);
53     }
54 }

```

完全注解实现





原xml

- 扫描包
- 两个jar包中的对象

```

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      <!-- *****注解: 告知spring在创建容器时要扫描的包***** --&gt;
10     &lt;context:component-scan base-package="com.learn"&gt;&lt;/context:component-
11     scan&gt;
12
13     &lt!-- 配置QueryRunner对象,我们是单表,一条语句的CRUD操作,我们可以选择传入数据源|连
14     接池 --&gt;
15     &lt!-- 这里如果是单例的话,会出现线程干扰问题 --&gt;
16     &lt;bean id="runner" class="org.apache.commons.dbutils.QueryRunner"
17         scope="prototype"&gt;
18         &lt!-- 注入数据源 --&gt;
19         &lt!-- QueryRunner没有set方法,只能使用 构造函数注入 --&gt;
20         &lt!-- new QueryRunner(ds) --&gt;
21         &lt;constructor-arg name="ds" ref="dataSource"&gt;&lt;/constructor-arg&gt;
22     &lt;/bean&gt;
23
24     &lt!-- 配置数据源 --&gt;
25     &lt;bean id="dataSource"
26         class="com.mchange.v2.c3p0.ComboPooledDataSource"&gt;
27         &lt!-- 注入 连接数据库的必备信息 --&gt;
28         &lt;!-- dataSource.setXXX() --&gt;
29         &lt;property name="driverClass" value="com.mysql.cj.jdbc.Driver"&gt;
30         &lt;/property&gt;
31         &lt;property name="jdbcUrl" value="jdbc:mysql://localhost:3306/db"&gt;
32         &lt;/property&gt;
33         &lt;property name="user" value="root"&gt;&lt;/property&gt;
34         &lt;property name="password" value="xxxx"&gt;&lt;/property&gt;
35     &lt;/bean&gt;
</pre>

```

java

```

1 /**
2  * 该类是一个配置类，它的作用和bean.xml是一样的
3 */
4 @Configuration
5 @ComponentScan(basePackages = "com.learn")
6 public class SpringConfiguration {
7
8     @Bean(name="runner")
9     @Scope("prototype")
10    public QueryRunner createQueryRunner(@Qualifier("ds1") DataSource
11 dataSource) {
12        return new QueryRunner(dataSource);
13    }
14
15    @Bean(name = "ds1")
16    public DataSource createDataSource() {
17        try {
18            ComboPooledDataSource ds = new ComboPooledDataSource();
19            ds.setDriverClass("com.mysql.cj.jdbc.Driver");
20            ds.setJdbcUrl("jdbc:mysql://localhost:3306/db");
21            ds.setUser("root");
22            ds.setPassword("xxxx");
23            return ds;
24        } catch (Exception e) {
25            throw new RuntimeException(e);
26        }
27    }

```

test

```

1 ApplicationContext context =
2         new AnnotationConfigApplicationContext(SpringConfiguration.class);

```

Spring实现转账案例

无事务版

CRUD+无事务转账

pom

```

1 <dependency>
2     <groupId>org.springframework</groupId>
3     <artifactId>spring-context</artifactId>
4     <version>5.2.5.RELEASE</version>

```

```

5 </dependency>
6
7 <!-- https://mvnrepository.com/artifact/commons-dbusils/commons-dbusils -->
8 <dependency>
9   <groupId>commons-dbusils</groupId>
10  <artifactId>commons-dbusils</artifactId>
11  <version>1.5</version>
12 </dependency>
13
14 <dependency>
15   <groupId>mysql</groupId>
16   <artifactId>mysql-connector-java</artifactId>
17   <version>8.0.15</version>
18 </dependency>
19
20 <!-- https://mvnrepository.com/artifact/com.mchange/c3p0 -->
21 <dependency>
22   <groupId>com.mchange</groupId>
23   <artifactId>c3p0</artifactId>
24   <version>0.9.5.2</version>
25 </dependency>
26
27 <dependency>
28   <groupId>junit</groupId>
29   <artifactId>junit</artifactId>
30   <version>4.12</version>
31 </dependency>
32
33 <dependency>
34   <groupId>org.springframework</groupId>
35   <artifactId>spring-test</artifactId>
36   <version>5.1.6.RELEASE</version>
37 </dependency>

```

bean.xml

```

1 <!-- *****把对象的创建交给spring来管理***** -->
2
3 <bean id="accountService"
4   class="com.learn.service.impl.AccountServiceImpl">
5   <!-- 注入dao -->
6   <!-- accountService.setXXX() -->
7   <property name="accountDao" ref="accountDao"></property>
8 </bean>
9
10 <!-- 配置Dao对象 -->
11 <bean id="accountDao" class="com.learn.dao.impl.AccountDaoImpl">
12   <!-- 注入QueryRunner -->
13   <!-- accountDao.setXXX() -->
14   <property name="runner" ref="runner"></property>
15 </bean>
16 <!-- 配置QueryRunner对象,我们是单表,一条语句的CRUD操作,我们可以选择传入数据源|连接池
-->
17 <!-- 这里如果是单例的话,会出现线程干扰问题 -->

```

```

18 <bean id="runner" class="org.apache.commons.dbutils.QueryRunner"
19   scope="prototype">
20     <!-- 注入数据源 -->
21     <!-- QueryRunner没有set方法,只能使用 构造函数注入 -->
22     <!-- new QueryRunner(ds) -->
23     <constructor-arg name="ds" ref="dataSource"></constructor-arg>
24 </bean>
25
26 <!-- 配置数据源 -->
27 <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
28   <!-- 注入 连接数据库的必备信息 -->
29   <!-- dataSource.setXxx() -->
30   <property name="driverClass" value="com.mysql.cj.jdbc.Driver">
31   </property>
32   <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/db">
33   </property>
34   <property name="user" value="root"></property>
35   <property name="password" value="xxxx"></property>
36 </bean>

```

dao

AccountDao

```

1 /**
2  * 账户的持久层接口
3 */
4 public interface AccountDao {
5
6   /**
7    * 查询所有
8    * @return
9    */
10  List<Account> findAllAccount();
11
12 /**
13  * 查询一个
14  * @return
15  */
16  Account findAccountById(Integer accountId);
17
18 /**
19  * 保存
20  * @param account
21  */
22  void saveAccount(Account account);
23
24 /**
25  * 更新
26  * @param account
27  */
28  void updateAccount(Account account);
29
30 /**
31  * 删除
32  * @param accountId

```

```

33     */
34     void deleteAccountById(Integer accountId);
35
36     /**
37      * 根据名称查询账户
38      * @param accountName
39      * @return 如果有唯一的一个结果就返回,如果没有结果就返回null
40      *          如果结果集超过一个就抛异常
41      */
42     Account findAccountByName(String accountName);
43 }

```

AccountDaoImpl

```

1 /**
2  * 账户的持久层实现类
3 */
4 public class AccountDaoImpl implements AccountDao {
5
6     private QueryRunner runner;
7
8     /**
9      * @param runner
10     */
11    public void setRunner(QueryRunner runner) {
12        this.runner = runner;
13    }
14
15    public List<Account> findAllAccount() {
16        //System.out.println("runner="+runner);
17        try {
18            return runner.query("select * from account", new
19            BeanListHandler<Account>(Account.class));
20        } catch (SQLException e) {
21            throw new RuntimeException(e);
22        }
23    }
24
25    public Account findAccountById(Integer accountId) {
26        try {
27            return runner.query("select * from account where id=?", new
28            BeanHandler<Account>(Account.class), accountId);
29        } catch (SQLException e) {
30            throw new RuntimeException(e);
31        }
32    }
33
34    public void saveAccount(Account account) {
35        try {
36            runner.update("insert into account(name,money) values(?,?)",
37            account.getName(), account.getMoney());
38        } catch (SQLException e) {
39            throw new RuntimeException(e);
40        }
41    }
42
43    public void updateAccount(Account account) {

```

```

39     try {
40         runner.update("update account set name=?, money=? where id=?",
41 account.getName(), account.getMoney(), account.getId());
42     } catch (SQLException e) {
43         throw new RuntimeException(e);
44     }
45 }
46 public void deleteAccountById(Integer accountId) {
47     try {
48         runner.update("delete from account where id=?", accountId);
49     } catch (SQLException e) {
50         throw new RuntimeException(e);
51     }
52 }
53
54 public Account findAccountByName(String accountName) {
55     try {
56         List<Account> accounts = runner.query("select * from account
57 where name=?", new BeanListHandler<Account>(Account.class), accountName);
58         if (accounts == null || accounts.size() == 0) {
59             return null;
60         }
61         if (accounts.size() > 1) {
62             throw new RuntimeException("结果集不唯一");
63         }
64         return accounts.get(0);
65     } catch (SQLException e) {
66         throw new RuntimeException(e);
67     }
68 }

```

domain

```

1 public class Account {
2
3     // -----变量-----
4
5     private Integer id;
6     private String name;
7     private Float money;
8
9     // -----constructor-----
10
11    /**
12     * 默认构造方法
13     */
14    public Account() {
15    }
16
17    /**
18     *

```

```

19     * @param id
20     * @param name
21     * @param money
22     */
23     public Account(Integer id, String name, Float money) {
24         this.id = id;
25         this.name = name;
26         this.money = money;
27     }
28
29     // -----getter and setter-----
30
31     public Integer getId() {
32         return id;
33     }
34
35     public void setId(Integer id) {
36         this.id = id;
37     }
38
39     public String getName() {
40         return name;
41     }
42
43     public void setName(String name) {
44         this.name = name;
45     }
46
47     public Float getMoney() {
48         return money;
49     }
50
51     public void setMoney(Float money) {
52         this.money = money;
53     }
54
55
56     // -----tostring -----
57
58     @Override
59     public String toString() {
60         return "Account{" +
61                 "id=" + id +
62                 ", name='" + name + '\'' +
63                 ", money=" + money +
64                 '}';
65     }
66 }
```

service

AccountService

1 | /**

```

2  * 账户的业务层接口
3  */
4  public interface AccountService {
5
6      /**
7       * 查询所有
8       * @return
9       */
10     List<Account> findAllAccount();
11
12    /**
13     * 查询一个
14     * @return
15     */
16     Account findAccountById(Integer accountId);
17
18    /**
19     * 保存
20     * @param account
21     */
22     void saveAccount(Account account);
23
24    /**
25     * 更新
26     * @param account
27     */
28     void updateAccount(Account account);
29
30    /**
31     * 删除
32     * @param accountId
33     */
34     void deleteAccountById(Integer accountId);
35
36    /**
37     * 转账
38     * @param sourceName      转出账户名称
39     * @param targetName      转入账户名称
40     * @param money           转账金额
41     */
42     void transfer(String sourceName, String targetName, Float money);
43 }

```

AccountServiceImpl

```

1 /**
2  * 账户的业务层实现类
3  */
4  public class AccountServiceImpl implements AccountService {
5
6     private AccountDao accountDao;
7
8     /**
9      * 需要Spring提供注入,我们提供set方法,这里是xml配置
10     * 注解配置没set也行

```

```
11     * @param accountDao
12     */
13     public void setAccountDao(AccountDao accountDao) {
14         this.accountDao = accountDao;
15     }
16
17     public List<Account> findAllAccount() {
18         return accountDao.findAllAccount();
19     }
20
21     public Account findAccountById(Integer accountId) {
22         return accountDao.findAccountById(accountId);
23     }
24
25     public void saveAccount(Account account) {
26         accountDao.saveAccount(account);
27     }
28
29     public void updateAccount(Account account) {
30         accountDao.updateAccount(account);
31     }
32
33     public void deleteAccountById(Integer accountId) {
34         accountDao.deleteAccountById(accountId);
35     }
36
37     public void transfer(String sourceName, String targetName, Float money)
{
38         //1.根据名称查询转出账户
39         Account source = accountDao.findAccountByName(sourceName);
40         //2.根据名称查询转入账户
41         Account target = accountDao.findAccountByName(targetName);
42         //3.转出账户 减钱
43         source.setMoney(source.getMoney() - money);
44         //4.转入账户 加钱
45         target.setMoney(target.getMoney() + money);
46         //5.更新转出账户
47         accountDao.updateAccount(source);
48
49         int i = 1/0;
50
51         //6.更新转入账户
52         accountDao.updateAccount(target);
53     }
54 }
```

问题

需要使用ThreadLocal
对象把Connection和
当前线程绑定，从而使
一个线程中只能有一个能
控制事务的对象

```

@Override
public void transfer(String sourceName, String targetName, Float money) {
    //1.根据名称查询转出账户
    Account source = accountDao.findAccountByName(sourceName); 获取一个连接 ✓
    //2.根据名称查询转入账户
    Account target = accountDao.findAccountByName(targetName); 获取一个连接 ✓
    //3.转出账户减钱
    source.setMoney(source.getMoney()-money);
    //4.转入账户加钱
    target.setMoney(target.getMoney()+money);
    //5.更新转出账户
    accountDao.updateAccount(source); 获取一个连接 ✓
    int i=1/0; ✗
    //6.更新转入账户
    accountDao.updateAccount(target); 获取一个连接 ✗
}

```

每次操作都是不同的连接，无法进行事务控制。

需要使用ThreadLocal对象把Connection和当前线程绑定，从而使一个线程中只能有一个能控制事务的对象。

事务应该在业务层。

手写事务版ThreadLocal

bean.xml

runner不再传入DataSource。DataSource转至 ConnectionUtils

```

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   xsi:schemaLocation="http://www.springframework.org/schema/beans
5   http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7   <!-- *****把对象的创建交给spring来管理***** -->
8
9   <bean id="accountService"
10  class="com.learn.service.impl.AccountServiceImpl">
11     <!-- 注入dao:accountService.setXXX() -->
12     <property name="accountDao" ref="accountDao"></property>
13     <!-- 注入事务管理器 -->
14     <property name="txManager" ref="txManager"></property>
15   </bean>
16
17   <!-- 配置Dao对象 -->
18   <bean id="accountDao" class="com.learn.dao.impl.AccountDaoImpl">
19     <!-- 注入QueryRunner:accountDao.setXXX() -->
20     <property name="runner" ref="runner"></property>
21     <!-- 注入Connectionutils:connectionutils.setXXX() -->
22     <property name="connectionutils" ref="connectionUtils"></property>
23   </bean>
24
25   <!-- 配置QueryRunner对象，我们是单表，一条语句的CRUD操作，我们可以选择传入数据源|连接池 -->
26   <!-- 这里如果是单例的话，会出现线程干扰问题 -->
27   <!-- 多例，在单线程中同样会是多例，这就是每次操作一个连接，无法进行事务管理 -->

```

```

26     <bean id="runner" class="org.apache.commons.dbutils.QueryRunner"
27         scope="prototype">
28         <!-- 注入数据源 -->
29         <!-- QueryRunner没有set方法,只能使用 构造函数注入 -->
30         <!-- new QueryRunner(ds) -->
31     <!--     <constructor-arg name="ds" ref="dataSource"></constructor-arg>-->
32     </bean>
33
34     <!-- 配置事务管理器 -->
35     <bean id="txManager" class="com.learn.utils.TransactionManager">
36         <!-- 注入Connectionutils:connectionUtils.setXxx() -->
37         <property name="connectionUtils" ref="connectionUtils"></property>
38     </bean>
39
40     <!-- 配置Connection的工具类 ConnectionUtils -->
41     <bean id="connectionUtils" class="com.learn.utils.ConnectionUtils">
42         <!-- 注入数据源:从QueryRunner转移到这里,QueryRunner要开事务就得无参构造 -->
43     <!--     <property name="dataSource" ref="dataSource"></property>
44     </bean>
45
46     <!-- 配置数据源 -->
47     <bean id="dataSource"
48         class="com.mchange.v2.c3p0.ComboPooledDataSource">
49         <!-- 注入 连接数据库的必备信息 -->
50         <!-- dataSource.setXxx() -->
51         <property name="driverClass" value="com.mysql.cj.jdbc.Driver">
52             </property>
53             <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/db">
54             </property>
55             <property name="user" value="root"></property>
56             <property name="password" value="xxxx"></property>
57     </bean>
58
59 </beans>

```

dao

AccountDaoImpl

- ConnectionUtils : 连接与线程绑定。从此处进行连接管理

```

1 /**
2  * 账户的持久层实现类
3 */
4 public class AccountDaoImpl implements AccountDao {
5
6     private QueryRunner runner;
7
8     private ConnectionUtils connectionUtils;
9
10    //
11    public void setRunner(QueryRunner runner) {
12        this.runner = runner;
13    }

```

```
14 //  
15     public void setConnectionUtils(Connectionutils connectionutils) {  
16         this.connectionUtils = connectionutils;  
17     }  
18  
19     public List<Account> findAllAccount() {  
20         try {  
21             return runner.query(connectionutils.getThreadConnection(),  
"select * from account", new BeanListHandler<Account>(Account.class));  
22         } catch (SQLException e) {  
23             throw new RuntimeException(e);  
24         }  
25     }  
26  
27     public Account findAccountById(Integer accountId) {  
28         try {  
29             return  
runner.query(connectionUtils.getThreadConnection(),"select * from account  
where id=?", new BeanHandler<Account>(Account.class), accountId);  
30         } catch (SQLException e) {  
31             throw new RuntimeException(e);  
32         }  
33     }  
34  
35     public void saveAccount(Account account) {  
36         try {  
37             runner.update(connectionUtils.getThreadConnection(),"insert  
into account(name,money) values(?,?)", account.getName(),  
account.getMoney());  
38         } catch (SQLException e) {  
39             throw new RuntimeException(e);  
40         }  
41     }  
42  
43     public void updateAccount(Account account) {  
44         try {  
45             runner.update(connectionUtils.getThreadConnection(),"update  
account set name=?, money=? where id=?", account.getName(),  
account.getMoney(), account.getId());  
46         } catch (SQLException e) {  
47             throw new RuntimeException(e);  
48         }  
49     }  
50  
51     public void deleteAccountById(Integer accountId) {  
52         try {  
53             runner.update(connectionUtils.getThreadConnection(),"delete  
from account where id=?", accountId);  
54         } catch (SQLException e) {  
55             throw new RuntimeException(e);  
56         }  
57     }  
58  
59     public Account findAccountByName(String accountName) {  
60         try {  
61             List<Account> accounts =  
runner.query(connectionUtils.getThreadConnection(),"select * from account  
where name=?", new BeanListHandler<Account>(Account.class), accountName);  
62         }  
63     }  
64 }
```

```
62         if (accounts == null || accounts.size() == 0) {
63             return null;
64         }
65         if (accounts.size() > 1) {
66             throw new RuntimeException("结果集不唯一");
67         }
68         return accounts.get(0);
69     } catch (SQLException e) {
70         throw new RuntimeException(e);
71     }
72 }
73 }
```

service

AccountServiceImpl

- 添加 TransactionManager，实现事务控制。

问题：

- TransactionManager虽然实现功能，但非常麻烦，每次用都一大坨包裹我们的处理逻辑，用多次。一旦TransactionManager修改，Service要维护就有些困难了。
- 这里就引出了代理模式的概念，再进一步就是AOP了

```
1 /**
2  * 账户的业务层实现类
3  * 加入事务管理操作
4 */
5 public class AccountServiceImpl implements AccountService {
6
7     private AccountDao accountDao;
8     private TransactionManager txManager;
9
10    /**
11     * 需要Spring提供注入，我们提供set方法，这里是xml配置
12     * 注解配置没set也行
13     * @param accountDao
14     */
15    public void setAccountDao(AccountDao accountDao) {
16        this.accountDao = accountDao;
17    }
18
19    /**
20     * 需要Spring提供注入，我们提供set方法，这里是xml配置
21     * 注解配置没set也行
22     * @param txManager
23     */
24    public void setTxManager(TransactionManager txManager) {
25        this.txManager = txManager;
26    }
27
28    public List<Account> findAllAccount() {
29        try {
30            //1.开启事务
31            txManager.beginTransaction();
```

```
32         //2.执行操作
33         List<Account> accounts = accountDao.findAllAccount();
34         //3.提交事务
35         txManager.commit();
36         //4.返回结果
37         return accounts;
38     } catch (Exception e) {
39         //5.回滚事务
40         txManager.rollback();
41         throw new RuntimeException(e);
42     } finally {
43         //6.释放连接
44         txManager.release();
45     }
46 }
47
48 public Account findAccountById(Integer accountId) {
49     try {
50         //1.开启事务
51         txManager.beginTransaction();
52         //2.执行操作
53         Account account = accountDao.findAccountById(accountId);
54         //3.提交事务
55         txManager.commit();
56         //4.返回结果
57         return account;
58     } catch (Exception e) {
59         //5.回滚事务
60         txManager.rollback();
61         throw new RuntimeException(e);
62     } finally {
63         //6.释放连接
64         txManager.release();
65     }
66 }
67
68 public void saveAccount(Account account) {
69     try {
70         //1.开启事务
71         txManager.beginTransaction();
72         //2.执行操作
73         accountDao.saveAccount(account);
74         //3.提交事务
75         txManager.commit();
76         //4.返回结果
77
78     } catch (Exception e) {
79         //5.回滚事务
80         txManager.rollback();
81         throw new RuntimeException(e);
82     } finally {
83         //6.释放连接
84         txManager.release();
85     }
86 }
87
88 public void updateAccount(Account account) {
89     try {
```

```
90     //1.开启事务
91     txManager.beginTransaction();
92     //2.执行操作
93     accountDao.updateAccount(account);
94     //3.提交事务
95     txManager.commit();
96     //4.返回结果
97
98 } catch (Exception e) {
99     //5.回滚事务
100    txManager.rollback();
101    throw new RuntimeException(e);
102 } finally {
103     //6.释放连接
104     txManager.release();
105 }
106
107 }
108
109 public void deleteAccountById(Integer accountId) {
110     try {
111         //1.开启事务
112         txManager.beginTransaction();
113         //2.执行操作
114         accountDao.deleteAccountById(accountId);
115         //3.提交事务
116         txManager.commit();
117         //4.返回结果
118     } catch (Exception e) {
119         //5.回滚事务
120         txManager.rollback();
121         throw new RuntimeException(e);
122     } finally {
123         //6.释放连接
124         txManager.release();
125     }
126
127 }
128
129 public void transfer(String sourceName, String targetName, Float
money) {
130     System.out.println("qqq");
131     try {
132         //1.开启事务
133         txManager.beginTransaction();
134         System.out.println("aa");
135         //2.执行操作
136         List<Account> accounts = accountDao.findAllAccount();
137
138         //2.1.根据名称查询转出账户
139         Account source = accountDao.findAccountByName(sourceName);
140         //2.2.根据名称查询转入账户
141         Account target = accountDao.findAccountByName(targetName);
142         //2.3.转出账户 减钱
143         source.setMoney(source.getMoney() - money);
144         //2.4.转入账户 加钱
145         target.setMoney(target.getMoney() + money);
146         //2.5.更新转出账户
```

```
147         accountDao.updateAccount(source);
148
149         int i = 1/0;
150
151         //2.6.更新转入账户
152         accountDao.updateAccount(target);
153
154         //3.提交事务
155         txManager.commit();
156         //4.返回结果
157     } catch (Exception e) {
158         System.out.println("bb");
159         //5.回滚事务
160         txManager.rollback();
161         throw new RuntimeException(e);
162     } finally {
163         System.out.println("cc");
164         //6.释放连接
165         txManager.release();
166     }
167 }
168 }
```

utils

ConnectionUtils

用来保证，同一线程 同一连接。

- 管理连接，获取 释放 (通过DataSource)
- 连接与线程 绑定 解绑

```
1 /**
2  * 连接的工具类,它用于从数据源中获取一个连接,并且实现和线程的绑定
3  * ThreadLocal - TransactionManager - service
4  * 怎么确定当前线程的?
5 */
6 public class ConnectionUtils {
7
8     /**
9      * 这里Thread是固定的,无需注入解耦
10     */
11
12     /**
13      * 不能new 或 自己创建;等着Spring为我们注入,xml方式提供set方法
14     */
15     private DataSource dataSource;
16
17     /**
18      * 用于注入
19      * @param dataSource
20     */
21     public void setDataSource(DataSource dataSource) {
22         this.dataSource = dataSource;
```

```

23     }
24
25     /**
26      * 获取当前线程上的连接
27      *      有,则返回
28      *      无,则获取 绑定当前线程
29      * @return
30      */
31     public Connection getThreadConnection() {
32         try {
33             //1.先从ThreadLocal上获取
34             Connection conn = tl.get();
35             //2.判断当前线程上是否有连接
36             if (conn == null) {
37                 //3.从数据源中获取一个连接,并且 和线程绑定(存入ThreadLocal中)
38                 conn = dataSource.getConnection();
39                 //4.把conn存入ThreadLocal中
40                 tl.set(conn);
41             }
42             //5.返回当前线程上的连接
43             return conn;
44         } catch (Exception e) {
45             throw new RuntimeException(e);
46         }
47     }
48
49     /**
50      * 连接 和 线程 解绑
51      */
52     public void removeConnection() {
53         tl.remove();
54     }
55 }
```

TransactionManager

获取该线程连接—>事务控制

- 开启事务
- 提交事务
- 回滚事务
- 释放连接

```

1 /**
2  * 和事务管理相关的工具类,它包含了:开启事务,提交事务,回滚事务 和 释放连接
3 */
4 public class TransactionManager {
5
6     /**
7      * 用于获取当前线程上的connection
8      * 等着Spring给我们注进来
9      */
10    private ConnectionUtils connectionUtils;
11
12    public void setConnectionUtils(ConnectionUtils connectionUtils) {
```

```

13     this.connectionUtils = connectionUtils;
14 }
15
16 /**
17 * 开启事务
18 */
19 public void beginTransaction() {
20     try {
21         connectionUtils.getThreadConnection().setAutoCommit(false);
22     } catch (SQLException e) {
23         e.printStackTrace();
24     }
25 }
26
27 /**
28 * 提交事务
29 */
30 public void commit() {
31     try {
32         connectionUtils.getThreadConnection().commit();
33     } catch (SQLException e) {
34         e.printStackTrace();
35     }
36 }
37
38 /**
39 * 回滚事务
40 */
41 public void rollback() {
42     try {
43         connectionUtils.getThreadConnection().rollback();
44     } catch (SQLException e) {
45         e.printStackTrace();
46     }
47 }
48
49 /**
50 * 释放连接
51 */
52 public void release() {
53     try {
54         connectionUtils.getThreadConnection().close(); //连接 还回 池中
55         connectionUtils.removeConnection(); //解除 绑定
56     } catch (SQLException e) {
57         e.printStackTrace();
58     }
59 }
60 }

```

动态代理实现事务控制

增强Service中的方法，这里创建了一个Service的工厂，通过工厂创建实例。

创建方法中，使用Proxy代理方式增强Service中方法。

BeanFactory

```
1 /**
2  * 用于创建Service的代理对象的工厂
3  */
4 public class BeanFactory {
5
6     private AccountService accountService;
7
8     /**
9      * 用于注入
10     * ?为啥加final
11     * @param accountService
12     */
13    public final void setAccountService(AccountService accountService) {
14        this.accountService = accountService;
15    }
16
17    private TransactionManager txManager;
18
19    /**
20     * 需要Spring提供注入，我们提供set方法，这里是xml配置
21     * 注解配置没set也行
22     * @param txManager
23     */
24    public void setTxManager(TransactionManager txManager) {
25        this.txManager = txManager;
26    }
27
28    /**
29     * 本来 获取Service的直接return
30     * 现在 获取Service代理对象
31     * 工厂模式 用该方法提供service, service的方法 被 增强了 ， 包上了厚厚的一层。
32     * @return
33     */
34    public AccountService getAccountService() {
35        return (AccountService)Proxy.newProxyInstance(
36            accountService.getClass().getClassLoader(),
37            accountService.getClass().getInterfaces(),
38            new InvocationHandler() {
39                /**
40                 * 添加事务支持
41                 * invoke方法具有拦截功能，能拦截 被代理对象 中的所有执行的方法
42                 * @param proxy
43                 * @param method
44                 * @param args
45                 * @return
46                 * @throws Throwable
47                 */
48                 public Object invoke(Object proxy, Method method,
49                                     Object[] args) throws Throwable {
50                     //消除重复代码 确实， 事务代码一遍完事 全在这里做
51                     //TransactionManager和Service的耦合降低，Tx方法该， 这里
52                     //改1次， Service改n多次
53                     //多个代码片段公有部分的提取
54                     Object returnVal = null;
55                     try {
```

```

54                     //1.开启事务
55                     txManager.beginTransaction();
56                     //2.执行操作
57                     //List<Account> accounts =
58                     accountDao.findAllAccount();
59                     returnVal = method.invoke(accountService,
60                     args);
61                     //3.提交事务
62                     txManager.commit();
63                     //4.返回结果
64                     return returnVal;
65             } catch (Exception e) {
66                     //5.回滚事务
67                     txManager.rollback();
68                     throw new RuntimeException(e);
69             } finally {
70                     //6.释放连接
71                     txManager.release();
72             }
73         );
74     }
75 }
```

Bean.xml

```

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      xsi:schemaLocation="http://www.springframework.org/schema/beans
5          http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7      <!-- *****把对象的创建交给spring来管理***** -->
8
9      <!-- 配置代理的service对象 -->
10     <!-- 工厂方式创建 -->
11     <bean id="proxyAccountService" factory-bean="beanFactory" factory-
12         method="getAccountService">
13
14         </bean>
15
16         <!-- 配置 beanFactory 用于事务管理 -->
17         <bean id="beanFactory" class="com.learn.factory.BeanFactory">
18             <!-- 注入service -->
19             <property name="accountService" ref="accountService"></property>
20             <!-- 注入事务管理器 -->
21             <property name="txManager" ref="txManager"></property>
22         </bean>
23
24         <!-- 配置Service -->
25         <bean id="accountService"
26             class="com.learn.service.impl.AccountServiceImpl">
27             <!-- 注入dao:accountService.setXXX() -->
28             <property name="accountDao" ref="accountDao"></property>
29         </bean>
30     
```

```

26      <!-- 注入事务管理器 事务转移到BeanFactory -->
27  <!--      <property name="txManager" ref="txManager"></property>-->
28  </bean>
29
30  <!-- 配置Dao对象 -->
31  <bean id="accountDao" class="com.learn.dao.impl.AccountDaoImpl">
32      <!-- 注入QueryRunner:accountDao.setXXX() -->
33      <property name="runner" ref="runner"></property>
34      <!-- 注入Connectionutils:connectionutils.setXXX() -->
35      <property name="connectionutils" ref="connectionutils"></property>
36  </bean>
37
38  <!-- 配置QueryRunner对象,我们是单表,一条语句的CRUD操作,我们可以选择传入数据源|连接池 -->
39  <!-- 这里如果是单例的话,会出现线程干扰问题 -->
40  <!-- 多例,在单线程中同样会是多例,这就是每次操作一个连接,无法进行事务管理 -->
41  <bean id="runner" class="org.apache.commons.dbutils.QueryRunner"
scope="prototype">
42      <!-- 注入数据源 -->
43      <!-- QueryRunner没有set方法,只能使用 构造函数注入 -->
44      <!-- new QueryRunner(ds) -->
45  <!--      <constructor-arg name="ds" ref="dataSource"></constructor-arg>-
->
46  </bean>
47
48  <!-- 配置事务管理器 -->
49  <bean id="txManager" class="com.learn.utils.TransactionManager">
50      <!-- 注入Connectionutils:connectionutils.setXXX() -->
51      <property name="connectionutils" ref="connectionutils"></property>
52  </bean>
53
54  <!-- 配置Connection的工具类 Connectionutils -->
55  <bean id="connectionutils" class="com.learn.utils.ConnectionUtils">
56      <!-- 注入数据源:从QueryRunner转移到这里,QueryRunner要开事务就得无参构造 -->
57      <property name="dataSource" ref="dataSource"></property>
58  </bean>
59
60  <!-- 配置数据源 -->
61  <bean id="dataSource"
62      class="com.mchange.v2.c3p0.ComboPooledDataSource">
63      <!-- 注入 连接数据库的必备信息 -->
64      <!-- dataSource.setXXX() -->
65      <property name="driverClass" value="com.mysql.cj.jdbc.Driver">
66      </property>
67      <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/db">
68      </property>
69      <property name="user" value="root"></property>
70      <property name="password" value="xxxx"></property>
71  </bean>
72
73  </beans>

```

XML AOP实现事务控制

bean.xml

事务管理器 方法 关联到 切入点即可。

- 事务管理器
- accountService

```
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:aop="http://www.springframework.org/schema/aop"
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/aop
8   http://www.springframework.org/schema/aop/spring-aop.xsd
9 ">
10
11    <!-- *****把对象的创建交给spring来管理***** -->
12
13    <!-- 配置Service -->
14    <bean id="accountService"
15      class="com.learn.service.impl.AccountServiceImpl">
16      <property name="accountDao" ref="accountDao"></property>
17    </bean>
18
19    <!-- 配置Dao对象 -->
20    <bean id="accountDao" class="com.learn.dao.impl.AccountDaoImpl">
21      <property name="runner" ref="runner"></property>
22      <property name="connectionUtils" ref="connectionUtils"></property>
23    </bean>
24
25    <!-- 配置QueryRunner对象,我们是单表,一条语句的CRUD操作,我们可以选择传入数据源|连接池 -->
26    <bean id="runner" class="org.apache.commons.dbutils.QueryRunner"
27      scope="prototype">
28      </bean>
29
30    <!-- 配置Connection的工具类 ConnectionUtils -->
31    <bean id="connectionUtils" class="com.learn.utils.ConnectionUtils">
32      <property name="dataSource" ref="dataSource"></property>
33    </bean>
34
35    <!-- 配置数据源 -->
36    <bean id="dataSource"
37      class="com.mchange.v2.c3p0.ComboPooledDataSource">
38      <property name="driverClass" value="com.mysql.cj.jdbc.Driver">
39      </property>
40      <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/db">
41      </property>
42      <property name="user" value="root"></property>
43      <property name="password" value="xxxx"></property>
44    </bean>
45
46    <!-- 配置事务管理器 -->
47    <bean id="txManager" class="com.learn.utils.TransactionManager">
48      <property name="connectionUtils" ref="connectionUtils"></property>
49    </bean>
```

```

46
47      <!-- aop -->
48      <aop:config>
49          <!-- 配置切入点表达式 -->
50          <aop:pointcut id="pt1" expression="execution(*
com.learn.service.impl.*.*(..))"/>
51
52          <aop:aspect id="txAdvice" ref="txManager">
53              <!-- 配置前置通知: 开启事务 -->
54              <aop:before method="beginTransaction" pointcut-ref="pt1">
55          </aop:before>
56              <!-- 配置后置通知: 提交事务 -->
57              <aop:after-returning method="commit" pointcut-ref="pt1">
58          </aop:after-returning>
59              <!-- 配置异常通知: 回滚事务 -->
60              <aop:after-throwing method="rollback" pointcut-ref="pt1">
61          </aop:after-throwing>
62              <!-- 配置最终通知: 释放连接 -->
63              <aop:after method="release" pointcut-ref="pt1"></aop:after>
64      </aop:aspect>
65  </aop:config>
66
67 </beans>

```

注解 AOP实现事务控制

注解配置就不说了。基本注册注入忽略。

```

1 /**
2  * 和事务管理相关的工具类,它包含了:开启事务,提交事务,回滚事务 和 释放连接
3 */
4 @Component("txManager")
5 @Aspect
6 public class TransactionManager {
7
8     /**
9      * 用于获取当前线程上的connection
10     * 等着spring给我们注进来
11     */
12     @Autowired
13     private Connectionutils connectionutils;
14
15     @Pointcut("execution(* com.learn.service.impl.*.*(..))")
16     private void pt1() {}
17
18 //     public void setConnectionutils(Connectionutils connectionutils) {
19 //         this.connectionutils = connectionutils;
20 //     }
21
22     /**
23      * 开启事务
24      */
25     @Before("pt1()")

```

```
26     public void beginTransaction() {
27         try {
28             connectionUtils.getThreadConnection().setAutoCommit(false);
29         } catch (SQLException e) {
30             e.printStackTrace();
31         }
32     }
33
34     /**
35      * 提交事务
36      */
37     @AfterReturning("pt1()")
38     public void commit() {
39         try {
40             connectionUtils.getThreadConnection().commit();
41         } catch (SQLException e) {
42             e.printStackTrace();
43         }
44     }
45
46     /**
47      * 回滚事务
48      */
49     @AfterThrowing("pt1()")
50     public void rollback() {
51         try {
52             connectionUtils.getThreadConnection().rollback();
53         } catch (SQLException e) {
54             e.printStackTrace();
55         }
56     }
57
58     /**
59      * 释放连接
60      */
61     @After("pt1()")
62     public void release() {
63         try {
64             connectionUtils.getThreadConnection().close(); //连接 还回 池中
65             connectionUtils.removeConnection(); //解除 绑定
66         } catch (SQLException e) {
67             e.printStackTrace();
68         }
69     }
70
71     @Around("pt1()")
72     public Object arroundAdvice(ProceedingJoinPoint pjp) {
73         Object rtValue = null;
74         try {
75             //1.获取参数
76             Object[] args = pjp.getArgs();
77             //2.开启事务
78             this.beginTransaction();
79             //3.执行方法
80             rtValue = pjp.proceed(args);
81             //4.提交事务
82             this.commit();
83         }
```

```
84         //返回结果
85         return rtValue;
86     } catch (Throwable throwable) {
87         //5.回滚事务
88         this.rollback();
89         throw new RuntimeException(throwable);
90     } finally {
91         //6.释放资源
92         this.release();
93     }
94 }
95
96 }
```

Spring AOP 日志

XML实现

pom

```
1 <dependency>
2     <groupId>org.springframework</groupId>
3     <artifactId>spring-context</artifactId>
4     <version>5.2.5.RELEASE</version>
5 </dependency>
6
7 <!-- 解析切入点表达式 -->
8 <dependency>
9     <groupId>org.aspectj</groupId>
10    <artifactId>aspectjweaver</artifactId>
11    <version>1.8.2</version>
12 </dependency>
```

bean.xml

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

```

5      xsi:schemaLocation="http://www.springframework.org/schema/beans
6          http://www.springframework.org/schema/beans/spring-beans.xsd
7
8          http://www.springframework.org/schema/aop
9          http://www.springframework.org/schema/aop/spring-aop.xsd
10
11    ">
12    <!-- 配置Spring的Ioc,把service对象配置进来 -->
13    <!-- 需要增强,加个日志 -->
14    <bean id="accountService"
15        class="com.learn.service.impl.AccountServiceImpl">
16        </bean>
17
18    <!-- 配置Logger类 -->
19    <!-- 写一个日志的通知 -->
20    <bean id="logger" class="com.learn.utils.Logger">
21        </bean>
22
23    <!-- 配置AOP -->
24    <aop:config>
25        <!-- 配置切入点表达式
26            在aop:aspect标签内部 只能在当前切面使用
27            它还可以卸载aop:aspect外面, 此时就变成了所有切面可用,必须写在切面前
28            -->
29        <aop:pointcut id="pt2" expression="execution(public *
30 com.learn.service.impl.*.*(..))"/>
31        <!-- 配置切面 引用通知 -->
32        <aop:aspect id="logAdvice" ref="logger">
33            <!-- 前置通知: 在切入点方法执行之前执行 -->
34            <aop:before method="beforePrintLog" pointcut="execution(public
35 * com.learn.service.impl.*.*(..))"></aop:before>
36            <!-- 后置通知: 在切入点方法正常执行之后执行, 它和后置通知永远只能执行一个
37            -->
38            <aop:after-returning method="afterReturningPrintLog"
39                pointcut="execution(public * com.learn.service.impl.*.*(..))"></aop:after-
40                returning>
41            <!-- 异常通知: 在切入点方法执行产生异常之后执行 -->
42            <aop:after-throwing method="afterThrowingPrintLog" pointcut-
43                ref="pt2"></aop:after-throwing>
44            <!-- 最终通知: 无论切入点方法是否正常执行它都会在其后面执行 -->
45            <aop:after method="afterPrintLog" pointcut-ref="pt1">
46        </aop:after>
47
48        <!-- 配置切入点表达式
49            在aop:aspect标签内部 只能在当前切面使用
50            它还可以卸载aop:aspect外面, 此时就变成了所有切面可用,但必须写在切面
前
51            -->
52        <aop:pointcut id="pt1" expression="execution(public *
53 com.learn.service.impl.*.*(..))"/>
54
55        <!-- 详细注释 情况logger类中 -->
56        <aop:around method="aroundPrintLog" pointcut-ref="pt2">
57        </aop:around>
58        </aop:aspect>
59    </aop:config>
60</beans>

```

service

AccountService

```
1  /**
2  * 账户的业务层接口
3  */
4  public interface AccountService {
5
6      /**
7      * 模拟保存账户
8      */
9      void saveAccount();
10
11     /**
12     * 模拟更新账户
13     * @param i
14     */
15     void updateAccount(int i);
16
17     /**
18     * 删除账户
19     * @return
20     */
21     int deleteAccount();
22
23 }
```

AccountServiceImpl

```
1 /**
2 * 账户的业务层实现类
3 */
4 public class AccountServiceImpl implements AccountService {
5
6
7     public void saveAccount() {
8         System.out.println("执行了保存");
9     }
10
11    public void updateAccount(int i) {
12        System.out.println("执行了更新");
13        int a = 1/0;
14    }
15
16    public int deleteAccount() {
17        System.out.println("执行了删除");
18        return 0;
19    }
20 }
```

utils

Logger

```
1  /**
2  * 用于记录日志的工具类,它里面提供了公共的代码
3  */
4  public class Logger {
5
6      /**
7      * 用于打印日志:并且计划在其切入点方法执行之前执行
8      * 切入点方法就是业务层方法
9      */
10     public void printLog() {
11         System.out.println("Logger类中的printLog方法开始记录日志了...");  

12     }
13
14     /**
15     * 前置通知
16     */
17     public void beforePrintLog() {
18         System.out.println("前置通知 Logger类中的before print Log方法开始记录日志了...");  

19     }
20
21     /**
22     * 后置通知
23     */
24     public void afterReturningPrintLog() {
25         System.out.println("后置通知 Logger类中的after returning print Log方法开始记录日志了...");  

26     }
27
28     /**
29     * 异常通知
30     */
31     public void afterThrowingPrintLog() {
32         System.out.println("异常通知 Logger类中的after throw print Log方法开始记录日志了...");  

33     }
34
35     /**
36     * 最终通知
37     */
38     public void afterPrintLog() {
39         System.out.println("最终通知 Logger类中的after print Log方法开始记录日志了...");  

40     }
41
42     /**
43     * 环绕通知
44     * 问题:
45     *       当我们配置了环绕通知之后, 切入单方法没有执行, 而通知方法执行了
46     * 分析:
```

```

47     *      通过对比动态代理中的环绕通知代码，发现动态代理的环绕通知有 明确的 切入点调
48     *      用，而我们的代码中没有
49     *      解决：
50     *      Spring框架为我们提供了一个接口： ProceedingJoinPoint。该接口有一个方法
51     *      proceed()，此方法就相当于明确调用切入点方法。
52     *      该接口可以作为环绕通知的方法参数，在程序执行时， Spring框架会为我们提供该
53     *      接口的实现类供我们使用
54     *      Spring中的环绕通知：
55     *      它是Spring框架为我们提供的一种可以在代码中手动控制增强方法合适执行的方式
56     */
57     public Object aroundPrintLog(ProceedingJoinPoint pjp) {
58         Object rtValue = null;
59         try {
60             System.out.println("环绕通知 前置 Logger类中的around print Log方法
开始记录日志了...");
61             Object[] args = pjp.getArgs(); //得到方法执行所需的参数
62             rtValue = pjp.proceed(args); //明确调用业务层方法
63             System.out.println("环绕通知 后置 Logger类中的around print Log方法
开始记录日志了...");
64             return rtValue;
65         } catch (Throwable throwable) {
66             System.out.println("环绕通知 异常 Logger类中的around print Log方法
开始记录日志了...");
67             throw new RuntimeException(throwable);
68         } finally {
69             System.out.println("环绕通知 最终 Logger类中的around print Log方法
开始记录日志了...");
70         }
71     }
72 }

```

注解实现

pom

```

1 <!-- https://mvnrepository.com/artifact/org.springframework/spring-context
-->
2 <dependency>
3     <groupId>org.springframework</groupId>
4     <artifactId>spring-context</artifactId>
5     <version>5.2.5.RELEASE</version>
6 </dependency>
7
8 <!-- 解析切入点表达式 -->
9 <dependency>
10    <groupId>org.aspectj</groupId>
11    <artifactId>aspectjweaver</artifactId>
12    <version>1.8.2</version>
13 </dependency>

```

bean.xml

```

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
5     xmlns:aop="http://www.springframework.org/schema/aop"
6     xmlns:context="http://www.springframework.org/schema/context"
7
8     xsi:schemaLocation="
9       http://www.springframework.org/schema/beans
10      http://www.springframework.org/schema/beans/spring-beans.xsd
11      http://www.springframework.org/schema/aop
12      http://www.springframework.org/schema/aop/spring-aop.xsd
13      http://www.springframework.org/schema/context
14      http://www.springframework.org/schema/context/spring-context.xsd">
15
16    <!-- 配置Spring创建容器时要扫描的包 支持注解 -->
17    <context:component-scan base-package="com.learn"></context:component-
18    scan>
19
20    <!-- 配置Spring开启注解AOP的支持 -->
21    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
22  </beans>

```

service

AccountService

```

1 /**
2  * 账户的业务层接口
3 */
4 public interface AccountService {
5
6   /**
7    * 模拟保存账户
8    */
9   void saveAccount();
10
11  /**
12   * 模拟更新账户
13   * @param i
14   */
15  void updateAccount(int i);
16
17  /**
18   * 删除账户
19   * @return
20   */
21  int deleteAccount();
22
23 }

```

AccountServiceImpl

```

1 /**

```

```

2  * 账户的业务层实现类
3  */
4 @Service("accountService")
5 public class AccountServiceImpl implements AccountService {
6
7
8     public void saveAccount() {
9         System.out.println("执行了保存");
10    }
11
12    public void updateAccount(int i) {
13        System.out.println("执行了更新");
14        int a = 1/0;
15    }
16
17    public int deleteAccount() {
18        System.out.println("执行了删除");
19        return 0;
20    }
21}

```

utils

Logger

```

1 /**
2  * 用于记录日志的工具类,它里面提供了公共的代码
3  *      Spring注解 执行调用有问题 最终 在后置和异常的前面
4  *      用环绕就没这个问题,自己写想咋就咋
5  */
6 @Component("logger")
7 @Aspect//当前类是一个切面类
8 public class Logger {
9
10    @Pointcut("execution(public * com.learn.service.impl.*(..))")
11    private void pt1() {}
12
13 /**
14  * 用于打印日志:并且计划在其切入点方法执行之前执行
15  *                      切入点方法就是业务层方法
16  */
17    public void printLog() {
18        System.out.println("Logger类中的printLog方法开始记录日志了... ");
19    }
20
21 /**
22  * 前置通知
23  */
24 @Before("pt1()")
25    public void beforePrintLog() {
26        System.out.println("前置通知 Logger类中的before print Log方法开始记录日志了... ");
27    }
28
29 /**
30  * 后置通知

```

```

31     */
32     @AfterReturning("pt1()")
33     public void afterReturningPrintLog() {
34         System.out.println("后置通知 Logger类中的after returning print Log方法
开始记录日志了...");
35     }
36
37     /**
38      * 异常通知
39      */
40     @AfterThrowing("pt1()")
41     public void afterThrowingPrintLog() {
42         System.out.println("异常通知 Logger类中的after throw print Log方法开始
记录日志了...");
43     }
44
45     /**
46      * 最终通知
47      */
48     @After("pt1()")
49     public void afterPrintLog() {
50         System.out.println("最终通知 Logger类中的after print Log方法开始记录日志
了...");
51     }
52
53
54     @Around("pt1()")
55     public Object aroundPrintLog(ProceedingJoinPoint pjp) {
56         Object rtValue = null;
57         try {
58             System.out.println("环绕通知 前置 Logger类中的around print Log方法
开始记录日志了...");
59             Object[] args = pjp.getArgs(); //得到方法执行所需的参数
60             rtValue = pjp.proceed(args); //明确调用业务层方法
61             System.out.println("环绕通知 后置 Logger类中的around print Log方法
开始记录日志了...");
62             return rtValue;
63         } catch (Throwable throwable) {
64             System.out.println("环绕通知 异常 Logger类中的around print Log方法
开始记录日志了...");
65             throw new RuntimeException(throwable);
66         } finally {
67             System.out.println("环绕通知 最终 Logger类中的around print Log方法
开始记录日志了...");
68         }
69     }
70 }

```

SpringJdbcTemplate实现CRUD

内置数据源。

硬编码数据源

```
1 /**
2  * JdbcTemplate的最基本用法
3 */
4 public class JdbcTemplateDemo1 {
5
6     public static void main(String[] args) {
7         //准备数据源: Spring的内置数据源
8         DriverManagerDataSource ds = new DriverManagerDataSource();
9         ds.setDriverClassName("com.mysql.cj.jdbc.Driver");
10        ds.setUrl("jdbc:mysql://localhost:3306/db");
11        ds.setUsername("root");
12        ds.setPassword("xxxx");
13
14        //1.创建JdbcTemplate对象
15        JdbcTemplate jt = new JdbcTemplate(); //这里传ds也一样
16        //1.1给jt设置数据源
17        jt.setDataSource(ds);
18        //2.执行操作
19        jt.execute("insert into account(name, money) values('ccc', 1000)");
20    }
21
22 }
```

xml配置

xml

```
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       xsi:schemaLocation="http://www.springframework.org/schema/beans
5                           http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7     <!-- 配置JdbcTemplate -->
8     <bean id="jdbcTemplate"
9           class="org.springframework.jdbc.core.JdbcTemplate">
10        <property name="dataSource" ref="datasource"></property>
11    </bean>
12
13    <!-- 配置数据源 -->
14    <bean id="datasource"
15          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
16        <property name="driverClassName" value="com.mysql.jdbc.Driver">
17        </property>
18        <property name="url" value="jdbc:mysql://localhost:3306/db">
19        </property>
20        <property name="username" value="root"></property>
21        <property name="password" value="xxxx"></property>
22    </bean>
23 </beans>
```

java

```
1 /**
2 * JdbcTemplate的最基本用法
3 */
4 public class JdbcTemplateDemo2 {
5
6     public static void main(String[] args) {
7         //1. 获取容器
8         ApplicationContext context = new
9             ClassPathXmlApplicationContext("bean.xml");
10        //2. 获取对象
11        JdbcTemplate jt = context.getBean("jdbcTemplate",
12            JdbcTemplate.class);
13        //3. 执行操作
14        jt.execute("insert into account(name, money) values('xxx', 2000)");
15    }
16 }
```

CRUD封装对象

```
1 /**
2 * JdbcTemplate的CRUD操作
3 */
4 public class JdbcTemplateDemo3 {
5
6     public static void main(String[] args) {
7         //1. 获取容器
8         ApplicationContext context = new
9             ClassPathXmlApplicationContext("bean.xml");
10        //2. 获取对象
11        JdbcTemplate jt = context.getBean("jdbcTemplate",
12            JdbcTemplate.class);
13        //3. 执行操作
14        //        jt.execute("insert into account(name, money) values('xxx',
15        //        2000)");
16
17        //3.1保存
18        jt.update("insert into account(name, money) values(?,?)", "eee",
19        3333);
20        //3.2更新
21        jt.update("update account set name=?, money=? where id=?", "test",
22        4567, 7);
23        //3.3删除
24        jt.update("delete from account where id=?", 8);
25        //3.4查询所有
26        // AccountRowMapper
27        //        List<Account> accounts = jt.query("select * from account where
28        //        money > ?", new AccountRowMapper(), 1000f);
29        //BeanPropertyRowMapper 可以返回对象或 集合
30        List<Account> accounts = jt.query("select * from account where
31        money > ?", new BeanPropertyRowMapper<Account>(Account.class), 1000f);
32        for (Account account : accounts) {
33            System.out.println(account);
34        }
35    }
36 }
```

```

27     }
28     //查询一个
29     List<Account> accounts1 = jt.query("select * from account where id
= ?", new BeanPropertyRowMapper<Account>(Account.class), 1);
30     System.out.println(accounts1.isEmpty() ? "没有内容" :
accounts1.get(0));
31     //查询返回一行一列 使用聚合函数 但不加 group by子句
32     Integer count = jt.queryForObject("select count(*) from account
where money > ?", Integer.class,/*Long.class*/ 1000f);
33     System.out.println(count);
34
35 }
36
37
38 /**
39 * 定义Account的封装策略
40 */
41 class AccountRowMapper implements RowMapper<Account> {
42     /**
43     * 把结果集中的数据封装到Account中，然后由Spring把每个Account加到集合中
44     * @param resultSet
45     * @param i
46     * @return
47     * @throws SQLException
48     */
49     public Account mapRow(ResultSet resultSet, int i) throws SQLException {
50         Account account = new Account();
51         account.setId(resultSet.getInt("id"));
52         account.setName(resultSet.getString("name"));
53         account.setMoney(resultSet.getFloat("money"));
54         return account;
55     }
56 }

```

dao

xml

```

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      xsi:schemaLocation="http://www.springframework.org/schema/beans
5      http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7      <!-- 配置账户的持久层 -->
8      <bean id="accountDao" class="com.learn.dao.impl.AccountDaoImpl">
9          <!-- 注入JdbcTemplate -->
10         <property name="jdbcTemplate" ref="jdbcTemplate"></property>
11     </bean>
12
13     <!-- 配置JdbcTemplate -->
14

```

```

15     <bean id="jdbcTemplate"
16         class="org.springframework.jdbc.core.JdbcTemplate">
17         <property name="dataSource" ref="datasource"></property>
18     </bean>
19
20     <!-- 配置数据源 -->
21     <bean id="datasource"
22         class="org.springframework.jdbc.datasource.DriverManagerDataSource">
23         <property name="driverClassName" value="com.mysql.jdbc.Driver">
24             </property>
25         <property name="url" value="jdbc:mysql://localhost:3306/db">
26             </property>
27         <property name="username" value="root"></property>
28         <property name="password" value="xxxx"></property>
29     </bean>
30 </beans>

```

java

```

1 /**
2 * 账户的持久层实现类
3 */
4 public class AccountDaoImpl implements AccountDao {
5
6     private JdbcTemplate jdbcTemplate;
7
8     public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
9         this.jdbcTemplate = jdbcTemplate;
10    }
11
12    public Account findAccountById(Integer accountId) {
13        List<Account> accounts = jdbcTemplate.query("select * from account
14 where id = ?",
15             new BeanPropertyRowMapper<Account>(Account.class),
16             accountId);
17        return accounts.isEmpty() ? null : accounts.get(0);
18    }
19
20    public Account findAccountByName(String accountName) {
21        List<Account> accounts = jdbcTemplate.query("select * from account
22 where id = ?",
23             new BeanPropertyRowMapper<Account>(Account.class),
24             accountName);
25        if (accounts.isEmpty()) {
26            return null;
27        }
28        if (accounts.size() > 1) {
29            throw new RuntimeException("结果集不唯一");
30        }
31        return accounts.get(0);
32    }
33
34    public void updateAccount(Account account) {
35        jdbcTemplate.update("update account set name=?,
36 money=? where
37 id=?",
38

```

```
30             account.getName(), account.getMoney(),
31         account.getId());
32     }
```

daoSupport

消除每个dao中的重复代码。提取出一个父类。

JdbcDaoSupport

Spring中提供JdbcDaoSupport

```
1  /**
2  * 用于抽取dao中的重复代码
3  *      不同的DAO都需要有 jdbcTemplate引用 和 set方法
4  *
5  *      DaoImpl继承该类
6  */
7  public class JdbcDaoSupport {
8
9      //----- 生成对象，传jdbcTemplate 或 ds都会给 jdbcTemplate 赋
10     -----值
11
12     /**
13      * set 注入 1: 直接 jdbcTemplate
14      */
15     private jdbcTemplate jdbcTemplate;
16
17     public jdbcTemplate getJdbcTemplate() {
18         return jdbcTemplate;
19     }
20
21     public void setJdbcTemplate(jdbcTemplate jdbcTemplate) {
22         this.jdbcTemplate = jdbcTemplate;
23     }
24
25     /**
26      * set 注入 2: dataSource
27      */
28     private dataSource dataSource;
29
30     //set
31     public void setDataSource(dataSource dataSource) {
32         this.dataSource = dataSource;
33         if (jdbcTemplate == null) {
34             jdbcTemplate = createJdbcTemplate(dataSource);
35         }
36     }
37
38     //new
39     private jdbcTemplate createJdbcTemplate(dataSource dataSource) {
```

```
39     return new JdbcTemplate(dataSource);
40 }
41 }
```

AccountDaoImpl

```
1 /**
2  * 账户的持久层实现类
3 */
4 public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {
5
6
7     public Account findAccountById(Integer accountId) {
8         List<Account> accounts = super.getJdbcTemplate().query("select *
from account where id = ?", new BeanPropertyRowMapper<Account>
(Account.class), accountId);
9         return accounts.isEmpty() ? null : accounts.get(0);
10    }
11
12    public Account findAccountByName(String accountName) {
13        List<Account> accounts = super.getJdbcTemplate().query("select *
from account where id = ?", new BeanPropertyRowMapper<Account>
(Account.class), accountName);
14        if (accounts.isEmpty()) {
15            return null;
16        }
17        if (accounts.size() > 1) {
18            throw new RuntimeException("结果集不唯一");
19        }
20        return accounts.get(0);
21    }
22
23    public void updateAccount(Account account) {
24        super.getJdbcTemplate().update("update account set name=?, money=?
where id=?",
25                                         account.getName(), account.getMoney(),
26                                         account.getId());
27    }
}
```

bean.xml

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

```

6      <!-- 配置账户的持久层 -->
7      <bean id="accountDao" class="com.learn.dao.impl.AccountDaoImpl">
8          <!-- 注入dataSource,触发setDataSource方法 -->
9          <property name="dataSource" ref="dataSource"></property>
10     </bean>
11
12     <!-- 配置数据源 -->
13     <bean id="dataSource"
14         class="org.springframework.jdbc.datasource.DriverManagerDataSource">
15         <property name="driverClassName" value="com.mysql.jdbc.Driver">
16             <property name="url" value="jdbc:mysql://localhost:3306/db">
17             <property name="username" value="root"></property>
18             <property name="password" value="xxxx"></property>
19         </bean>
20     </beans>

```

Spring 事务控制

无事务 环境

pom

```

1 <dependencies>
2     <dependency>
3         <groupId>org.springframework</groupId>
4         <artifactId>spring-context</artifactId>
5         <version>5.1.6.RELEASE</version>
6     </dependency>
7
8     <dependency>
9         <groupId>org.springframework</groupId>
10        <artifactId>spring-jdbc</artifactId>
11        <version>5.1.6.RELEASE</version>
12    </dependency>
13
14    <dependency>
15        <groupId>org.springframework</groupId>
16        <artifactId>spring-tx</artifactId>
17        <version>5.1.6.RELEASE</version>
18    </dependency>
19
20    <dependency>
21        <groupId>mysql</groupId>
22        <artifactId>mysql-connector-java</artifactId>
23        <version>8.0.15</version>
24    </dependency>
25
26    <!-- 事务控制也是基于AOP的 -->
27    <dependency>

```

```

28      <groupId>org.aspectj</groupId>
29      <artifactId>aspectjweaver</artifactId>
30      <version>1.8.2</version>
31  </dependency>
32
33
34  <dependency>
35      <groupId>junit</groupId>
36      <artifactId>junit</artifactId>
37      <version>4.12</version>
38  </dependency>
39
40  <dependency>
41      <groupId>org.springframework</groupId>
42      <artifactId>spring-test</artifactId>
43      <version>5.1.6.RELEASE</version>
44  </dependency>
45 </dependencies>

```

bean.xml

```

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      xsi:schemaLocation="http://www.springframework.org/schema/beans
5          http://www.springframework.org/schema/beans/spring-beans.xsd">
6
7      <!-- 配置业务层 -->
8      <bean id="accountService"
9          class="com.learn.service.impl.AccountServiceImpl">
10         <property name="accountDao" ref="accountDao"></property>
11     </bean>
12
13     <!-- 配置账户的持久层 -->
14     <!-- 继承了JdbcDaoSupport -->
15     <bean id="accountDao" class="com.learn.dao.impl.AccountDaoImpl">
16         <!-- 注入dataSource,触发setDataSource方法 -->
17         <property name="dataSource" ref="dataSource"></property>
18     </bean>
19
20     <!-- 配置数据源 -->
21     <bean id="dataSource"
22         class="org.springframework.jdbc.datasource.DriverManagerDataSource">
23         <property name="driverClassName" value="com.mysql.jdbc.Driver">
24             <property name="url" value="jdbc:mysql://localhost:3306/db">
25         </property>
26         <property name="username" value="root"></property>
27         <property name="password" value="xxxx"></property>
28     </bean>
29 </beans>

```

domain

```
1  /**
2  * 账户的实体类
3  */
4  public class Account implements Serializable {
5
6      private Integer id;
7      private String name;
8      private Float money;
9
10     public Integer getId() {
11         return id;
12     }
13
14     public void setId(Integer id) {
15         this.id = id;
16     }
17
18     public String getName() {
19         return name;
20     }
21
22     public void setName(String name) {
23         this.name = name;
24     }
25
26     public Float getMoney() {
27         return money;
28     }
29
30     public void setMoney(Float money) {
31         this.money = money;
32     }
33
34     @Override
35     public String toString() {
36         return "Account{" +
37                 "id=" + id +
38                 ", name='" + name + '\'' +
39                 ", money=" + money +
40                 '}';
41     }
42 }
```

dao

AccountDao

```
1  /**
2  * 账户的持久层接口
3  */
4  public interface AccountDao {
5      /**
6       * 根据Id查询账户
7       * @param accountId
8       * @return
```

```
9 */  
10 Account findAccountById(Integer accountId);  
11  
12 /**  
13 * 根据名称查询账户  
14 * @param accountName  
15 * @return  
16 */  
17 Account findAccountByName(String accountName);  
18  
19 /**  
20 * 更新账户  
21 * @param account  
22 */  
23 void updateAccount(Account account);  
24  
25 }
```

AccountDaoImpl

```
1  /**
2   * 账户的持久层实现类
3   */
4  public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {
5
6
7      public Account findAccountById(Integer accountId) {
8          List<Account> accounts = super.getJdbcTemplate().query("select * from account where id = ?",
9          new BeanPropertyRowMapper<Account>(Account.class), accountId);
10         return accounts.isEmpty() ? null : accounts.get(0);
11     }
12
13     public Account findAccountByName(String accountName) {
14         List<Account> accounts = super.getJdbcTemplate().query("select * from account where name = ?",
15         new BeanPropertyRowMapper<Account>(Account.class),
16         accountName);
17         if (accounts.isEmpty()) {
18             return null;
19         }
20         if (accounts.size() > 1) {
21             throw new RuntimeException("结果集不唯一");
22         }
23         return accounts.get(0);
24     }
25
26     public void updateAccount(Account account) {
27         super.getJdbcTemplate().update("update account set name=?, money=? where id=?",
28         account.getName(), account.getMoney(),
29         account.getId());
30     }
31 }
```

service

AccountService

```
1  /**
2  * 账户的业务层接口
3  */
4  public interface AccountService {
5
6      /**
7      * 根据id查询账户信息
8      * @param accountId
9      * @return
10     */
11    Account findAccountById(Integer accountId);
12
13    /**
14     * 转账
15     * @param sourceName    转出账户名称
16     * @param targetName   转入账户名称
17     * @param money        转账金额
18     */
19    void transfer(String sourceName, String targetName, Float money);
20
21 }
```

AccountServiceImpl

```
1 /**
2 * 账户的业务层实现类
3 *
4 * 事务控制应该都是在业务层
5 */
6  public class AccountServiceImpl implements AccountService {
7
8      private AccountDao accountDao;
9
10     public void setAccountDao(AccountDao accountDao) {
11         this.accountDao = accountDao;
12     }
13
14     public Account findAccountById(Integer accountId) {
15         return accountDao.findAccountById(accountId);
16     }
17
18     public void transfer(String sourceName, String targetName, Float money)
{
19         System.out.println("transfer...");
20         Account source = accountDao.findAccountByName(sourceName);
21         Account target = accountDao.findAccountByName(targetName);
22
23         System.out.println("??");
24
25         source.setMoney(source.getMoney() - money);
26         target.setMoney(target.getMoney() + money);
27     }
28 }
```

```

27         accountDao.updateAccount(source);
28
29     //     int i = 1/0;
30
31     accountDao.updateAccount(target);
32 }
33
34 }
```

XML API 事务

加入声明式事务控制，bean.xml 加入如下内容即可完成配置。管理器为 Spring 提供，我们做好对应关联即可。

- aop
- tx

```

1 <beans xmlns="http://www.springframework.org/schema/beans"
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3
4   xmlns:aop="http://www.springframework.org/schema/aop"
5   xmlns:tx="http://www.springframework.org/schema/tx"
6
7   xsi:schemaLocation="http://www.springframework.org/schema/beans
8
9   http://www.springframework.org/schema/beans/spring-beans.xsd
10          http://www.springframework.org/schema/aop
11
12          http://www.springframework.org/schema/aop/spring-aop.xsd
13          http://www.springframework.org/schema/tx
14          http://www.springframework.org/schema/tx/spring-
15          tx.xsd">
16
17     <!-- 配置事务管理器 -->
18     <bean id="transactionManager"
19       class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
20       <property name="dataSource" ref="dataSource"></property>
21     </bean>
22
23     <!-- 配置事务的通知 -->
24     <tx:advice id="txAdvice" transaction-manager="transactionManager">
25       <!-- 配置事务的属性 -->
26       <tx:attributes>
27         <!-- findAccountById transfer -->
28         <tx:method name="*" propagation="REQUIRED" read-only="false"/>
29         <!-- findAccountById -->
30         <tx:method name="find*" propagation="SUPPORTS" read-
31         only="true"></tx:method>
32       </tx:attributes>
33     </tx:advice>
34
35     <!-- 配置aop -->
```

```

31    <aop:config>
32        <!-- 配置切入点表达式 -->
33        <aop:pointcut id="pt1" expression="execution(*
34            com.learn.service.impl.*.*(..))"/>
35        <!-- 建立切入点表达式和事务通知的对应关系 -->
36        <aop:advisor advice-ref="txAdvice" pointcut-ref="pt1">
37    </aop:advisor>
38    </aop:config>

```

注解 API 事务

bean

- 注解需加入context。
- 扫描包
- +JdbcTemplate
- 事务通知 aop配置 移除，交由注解
- 开启spring对注解的支持

```

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:aop="http://www.springframework.org/schema/aop"
5      xmlns:tx="http://www.springframework.org/schema/tx"
6      xmlns:context="http://www.springframework.org/schema/context"
7      xsi:schemaLocation="
8          http://www.springframework.org/schema/beans
9          http://www.springframework.org/schema/beans/spring-beans.xsd
10         http://www.springframework.org/schema/tx
11         http://www.springframework.org/schema/tx/spring-tx.xsd
12         http://www.springframework.org/schema/aop
13         http://www.springframework.org/schema/aop/spring-aop.xsd
14         http://www.springframework.org/schema/context
15         http://www.springframework.org/schema/context/spring-context.xsd">
16
17     <!-- 配置spring创建容器时要扫描的包-->
18     <context:component-scan base-package="com.learn"></context:component-
19     scan>
20
21     <!-- 配置JdbcTemplate -->
22     <bean id="jdbcTemplate"
23         class="org.springframework.jdbc.core.JdbcTemplate">
24         <property name="dataSource" ref="dataSource"></property>
25     </bean>
26
27     <!-- 配置数据源 -->
28     <bean id="dataSource"
29         class="org.springframework.jdbc.datasource.DriverManagerDataSource">
30         <property name="driverClassName" value="com.mysql.jdbc.Driver">
31     </property>

```

```

28         <property name="url" value="jdbc:mysql://localhost:3306/db">
29     </property>
30         <property name="username" value="root"></property>
31         <property name="password" value="xxxx"></property>
32     </bean>
33
34     <!-- spring中基于注解 的声明式事务控制配置步骤
35         1、配置事务管理器
36         2、开启spring对注解事务的支持
37         3、在需要事务支持的地方使用@Transactional注解
38     -->
39     <!-- 配置事务管理器 -->
40     <bean id="transactionManager"
41         class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
42         <property name="dataSource" ref="dataSource"></property>
43     </bean>
44
45     <!-- 开启spring对注解事务的支持-->
46     <tx:annotation-driven transaction-manager="transactionManager">
47     </tx:annotation-driven>
48
49 </beans>

```

AccountServiceImpl

注册注入

加@Transactional 表示需要事务支持

```

1 /**
2 * 账户的业务层实现类
3 *
4 * 事务控制应该都是在业务层
5 */
6 @Service("accountService")
7 @Transactional(propagation= Propagation.SUPPORTS, readOnly=true)//只读型事务的
8 //配置
9 public class AccountServiceImpl implements AccountService {
10
11     @Autowired
12     private AccountDao accountDao;
13
14     //只读型事务的配置
15     public Account findAccountById(Integer accountId) {
16         return accountDao.findAccountById(accountId);
17     }
18
19     //需要的是读写型事务配置
20     @Transactional(propagation= Propagation.REQUIRED, readOnly=false)
21     public void transfer(String sourceName, String targetName, Float money)
22     {
23         System.out.println("transfer....");
24         //2.1根据名称查询转出账户
25         Account source = accountDao.findAccountByName(sourceName);
26     }
27
28 }

```

```

25         //2.2根据名称查询转入账户
26         Account target = accountDao.findAccountByName(targetName);
27         //2.3转出账户减钱
28         source.setMoney(source.getMoney()-money);
29         //2.4转入账户加钱
30         target.setMoney(target.getMoney()+money);
31         //2.5更新转出账户
32         accountDao.updateAccount(source);
33
34         int i=1/0;
35
36         //2.6更新转入账户
37         accountDao.updateAccount(target);
38     }
39 }
```

AccountDaoImpl

注册注入

不能继承JdbcDaoSupport

```

1 /**
2  * 账户的持久层实现类
3 */
4 @Repository("accountDao")
5 public class AccountDaoImpl implements AccountDao {
6
7     @Autowired
8     private JdbcTemplate jdbcTemplate;
9
10    public Account findAccountById(Integer accountId) {
11        List<Account> accounts = jdbcTemplate.query("select * from account
12 where id = ?",new BeanPropertyRowMapper<Account>(Account.class),accountId);
13        return accounts.isEmpty()?null:accounts.get(0);
14    }
15
16    public Account findAccountByName(String accountName) {
17        List<Account> accounts = jdbcTemplate.query("select * from account
18 where name = ?",new BeanPropertyRowMapper<Account>
19 (Account.class),accountName);
20        if(accounts.isEmpty()){
21            return null;
22        }
23        if(accounts.size()>1){
24            throw new RuntimeException("结果集不唯一");
25        }
26        return accounts.get(0);
27    }
28
29    public void updateAccount(Account account) {
30        jdbcTemplate.update("update account set name=?,money=? where
31 id=?",account.getName(),account.getMoney(),account.getId());
32    }
33 }
```

纯注解 API 事务

去掉bean.xml。

jdbcConfig.properties

```
1 jdbc.driver=com.mysql.cj.jdbc.Driver  
2 jdbc.url=jdbc:mysql://localhost:3306/db  
3 jdbc.username=root  
4 jdbc.password=xxxx
```

SpringConfiguration

```
1 /**
2  * Spring的配置类
3  * 相当于bean.xml
4 */
5 @Configuration
6 @ComponentScan("com.learn")
7 @Import({JdbcConfig.class, TransactionConfig.class})
8 @PropertySource("jdbcConfig.properties")
9 @EnableTransactionManagement//事务注解支持
10 public class SpringConfiguration {
11 }
```

JdbcConfig

```
1 /**
2  * 和连接数据库相关的配置类
3 */
4 public class JdbcConfig {
5
6     @Value("${jdbc.driver}")
7     private String driver;
8
9     @Value("${jdbc.url}")
10    private String url;
11
12    @Value("${jdbc.username}")
13    private String username;
14
15    @Value("${jdbc.password}")
16    private String password;
17
18    /**
19     * 创建JdbcTemplate对象
20 }
```

```
20     * @param dataSource
21     * @return
22     */
23     @Bean(name = "jdbcTemplate")
24     public JdbcTemplate createJdbcTemplate(DataSource dataSource) {
25         return new JdbcTemplate(dataSource);
26     }
27
28     @Bean(name = "dataSource")
29     public DataSource createDataSource() {
30         DriverManagerDataSource ds = new DriverManagerDataSource();
31         ds.setDriverClassName(driver);
32         ds.setUrl(url);
33         ds.setUsername(username);
34         ds.setPassword(password);
35         return ds;
36     }
37
38 }
```

TransactionConfig

```
1 /**
2  * 和事务相关的配置类
3  */
4 public class TransactionConfig {
5     /**
6      * 用于闯进事务管理器对象
7      * @param dataSource
8      * @return
9      */
10     @Bean(name = "transactionManager")
11     public PlatformTransactionManager createTransactionManager(DataSource
12     dataSource) {
13         return new DataSourceTransactionManager(dataSource);
14     }
15 }
```

EOF