Spring学习笔记

一、简介概述

1.1 Spring是什么

- Spring:春天 ---> 给软件行业带来了春天~
- Spring是Java EE编程领域的一个轻量级开源框架，该框架由一个叫Rod Johnson的程序员在 2002 年最早提出并随后创建，是为了解决企业级编程开发中的复杂性，实现敏捷开发的应用型框架 。
- Spring是一个开源容器框架，它集成各类型的工具
- Spring的前身是interface21框架,于2004.03.21发布1.0版本
- Spring是一个轻量级Java开发框架，最早有Rod Johnson创建，目的是为了解决企业级应用开发的业务逻辑层和其他各层的耦合问题。它是一个分层的JavaSE/JavaEE full-stack（一站式）轻量级开源框架，为开发Java应用程序提供全面的基础架构支持。Spring负责基础架构，因此Java开发者可以专注于应用程序的开发。

1.2 优点

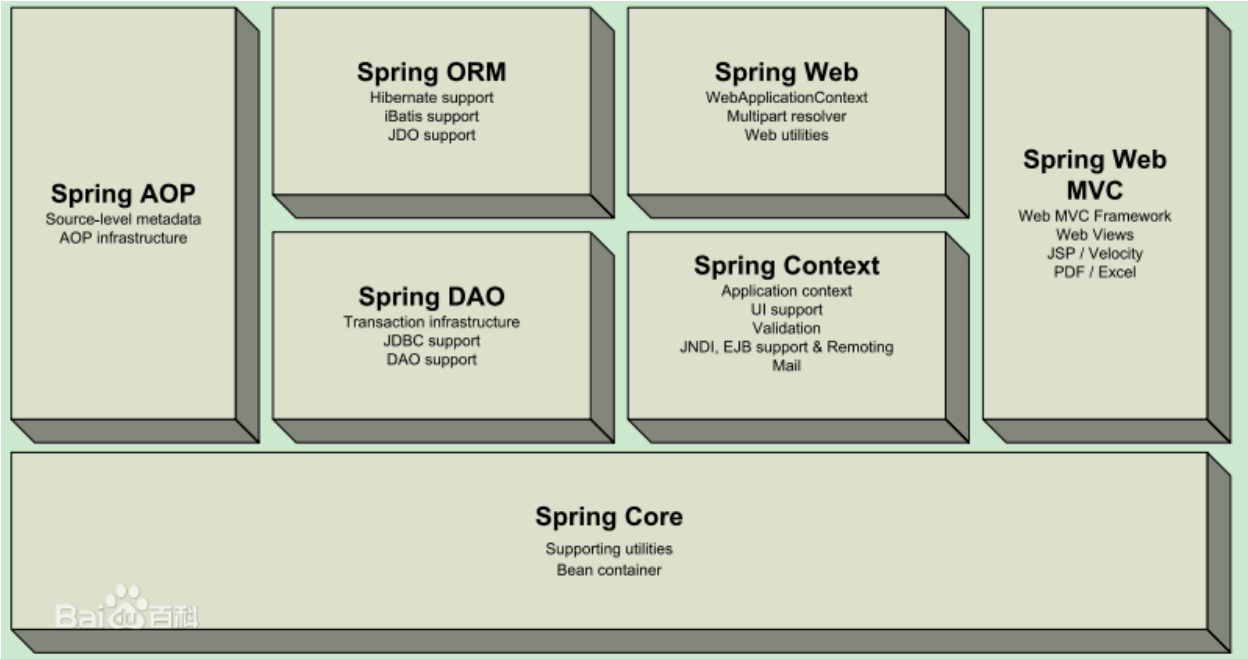
- 方便解耦，简化开发
  Spring就是一个大工厂，可以将所有对象创建和依赖的关系维护，交给Spring管理。
- AOP编程的支持
  Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能。
- 声明式事务的支持
  只需要通过配置就可以完成对事务的管理，而无需手动编程。
- 方便程序的测试
  Spring对Junit4支持，可以通过注解方便的测试Spring程序。
- 方便集成各种优秀框架
  Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架的直接支持（如：Struts、Hibernate、MyBatis等）。
- 降低JavaEE API的使用难度
  Spring对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低。

1.3 地址

- 官网下载地址 https://repo.spring.io/release/org/springframework/spring/
- Github地址 https://github.com/spring-projects/spring-framework
- maven仓库 https://mvnrepository.com/search?q=spring

1.4 七大模块

- Spring框架主要由七部分组成，分别是 Spring Core、 Spring AOP、 Spring ORM、 Spring DAO、Spring Context、 Spring Web和 Spring Web MVC。

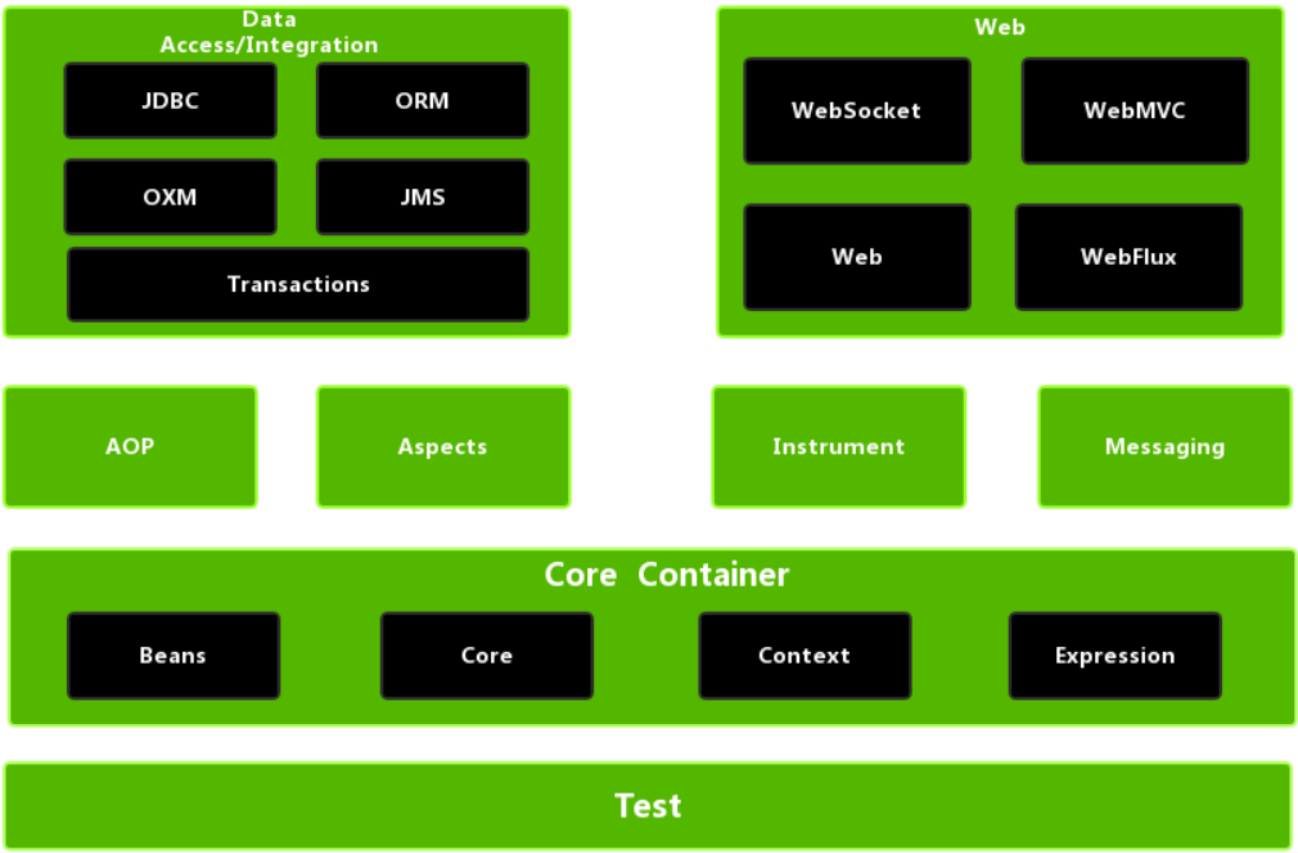


1.5 体系结构

Spring框架至今已集成了20多个模块，这些模块分布在以下模块中：

- 核心容器（Core Container）
- 数据访问/集成（Data Access/Integration）层
- Web层
- AOP（Aspect Oriented Programming）模块
- 植入（Instrumentation）模块
- 消息传输（Messaging）
- 测试（Test）模块

Spring体系结构如下图：



1、核心容器 Spring的核心容器是其他模块建立的基础，有Spring-core、Spring-beans、Spring-context、Spring-context-support和Spring-expression（String表达式语言）等模块组成。 Spring-core模块：提供了框架的基本组成部分，包括控制反转（Inversion of Control，IOC）和依赖注入（Dependency Injection，DI）功能。 Spring-beans模块：提供了BeanFactory，是工厂模式的一个经典实现，Spring将管理对象称为Bean。 Spring-context模块：建立在Core和Beans模块的基础之上，提供一个框架式的对象访问方式，是访问定义和配置的任何对象的媒介。ApplicationContext接口是Context模块的焦点。 Spring-context-support模块：支持整合第三方库到Spring应用程序上下文，特别是用于高速缓存（EhCache、JCache）和任务调度（CommonJ、Quartz）的支持。 Spring-expression模块：提供了强大的表达式语言去支持运行时查询和操作对象图。这是对JSP2.1规范中规定的统一表达式语言（Unified EL）的扩展。该语言支持设置和获取属性值、属性分配、方法调用、访问数组、集合和索引器的内容、逻辑和算术运算、变量命名以及从Spring的IOC容器中以名称检索对象。它还支持列表投影、选择以及常用的列表聚合。

2、AOP和Instrumentation Spring-aop模块：提供了一个符合AOP要求的面向切面的编程实现，允许定义方法拦截器和切入点，将代码按照功能进行分离，以便干净地解耦。 Spring-aspects模块：提供了与AspectJ的集成功能，AspectJ是一个功能强大且成熟的AOP框架。 Spring-instrument模块：提供了类植入（Instrumentation）支持和类加载器的实现，可以在特定的应用服务器中使用。

3、消息 Spring4.0以后新增了消息（Spring-messaging）模块，该模块提供了对消息传递体系结构和协议的支持。

4、数据访问/集成 数据访问/集成层由JDBC、ORM、OXM、JMS和事务模块组成。 Spring-jdbc模块：提供了一个JDBC的抽象层，消除了烦琐的JDBC编码和数据库厂商特有的错误代码解析。 Spring-orm模块：为流行的对象关系映射（Object-Relational Mapping）API提供集成层，包括JPA和Hibernate。使用Spring-orm模块可以将这些O/R映射框架与Spring提供的所有其他功能结合使用，例如声明式事务管理功能。 Spring-oxm模块：提供了一个支持对象/XML映射的抽象层实现，例如JAXB、Castor、JiBX和XStream。 Spring-jms模块（Java Messaging Service）：指Java消息传递服务，包含用于生产和使用消息的功能。自Spring4.1以后，提供了与Spring-messaging模块的集成。 Spring-tx模块（事务模块）：支持用于实现特殊接口和所有POJO（普通Java对象）类的编程和声明式事务管理。

> 5、Web Web层由Spring-web、Spring-webmvc、Spring-websocket和Portlet模块组成。 Spring-web模块：提供了基本的Web开发集成功能，例如多文件上传功能、使用Servlet监听器初始化一个IOC容器以及Web应用上下文。 Spring-webmvc模块：也称为Web-Servlet模块，包含用于web应用程序的Spring MVC和REST Web Services实现。Spring MVC框架提供了领域模型代码和Web表单之间的清晰分离，并与Spring Framework的所有其他功能集成。 Spring-websocket模块：Spring4.0以后新增的模块，它提供了WebSocket和SocketJS的实现。 Portlet模块：类似于Servlet模块的功能，提供了Portlet环境下的MVC实现。

> 6、测试 Spring-test模块支持使用JUnit或TestNG对Spring组件进行单元测试和集成测试。

二、IOC

2.1 IOC理论推导

2.1.1 程序间耦合

> **耦合**:程序间的依赖关系 包括 类之间的依赖和方法间的依赖 **解耦**:降低程序间的依赖关系(而不是完全消除,必要的一些依赖是避免不了的) 实际开发过程中,应该做到编译器不依赖,运行期才依赖

案例一:我们看看下面这段JDBC有关代码

```
public class JdbcDemo1 {
    public statiE void main(String[ ] args) throws Exception{
      //1.注册驱动
      DriverManager.registerDriver(new com.mysql.jdbc.Driver());
      //Class.forName("com.mysq1.jdbc.Driver");
      //2.获取连接
      Connection conn = DriverManager.getConnection(url: "jdbc
:mysql://localhost:3306/test","root","root");
      //3.获取操作数据库的预处理对象
      PreparedStatement pstm = conn. prepareStatement( sql: "select * from account" );
      //4.执行SQL,得到结果集
      ResultSet rs = pstm. executeQuery() ;
      //5.遍历结果集
      while(rs.next( )){
        System.out.println(rs.getString("name"));
      }
      //6.释放资源
      rs.close();
      pstm.close();
      conn.close();
    }
}
```

> 当我们不导入mysql-connector-java这个包时,保错是必然的 我们在用DriverManager注册驱动时,程序就已经标红了,这说明是编译期错误. 而我们用Class.forName()时,里面只提供了一个字符串,程序编译并不会错,而是运行时异常.

> 从上面可以发现一些解耦的思路 利用Class.forName反射创建对象,而尽量少用new关键字 而用Class.forName,里面的字符串已经被写死了,而之后想要用其他数据库的话需要修改,这时我们可以采用properties配置文件

案例二:在实际开发中,我们一般采用三层架构模式开发,控制层,业务层,持久层,他们之间有严重的依赖关系(耦合性),对方法进行一层层的调用 看以前学javaweb时的代码

```
//模拟持久层实现类
public class AccountDaoImpl implements AccountDao{
    public void save(){
      System.out.println("账户已被保存");
    }
}

//模拟业务层实现类
public class AccountServiceImpl implements AccountService{
    private AccountDao accountDao = new AccountDaoImpl();

    public void save(){
      accountDao.save();
    }
}

//模拟表现层
public class AccountController{
    public static void main(){
      AccountService accountService = new AccountServiceImpl();
      accountService.save();
    }
}
```

> 在业务层和表现层代码里,我们使用了new关键词,使代码有很强的依赖关系,即耦合性,这是我们应该避免的.

2.1.2 工厂模式解耦

名词解释:

- Bean:在计算机英语中有**可重用组件**的意思
- JavaBean:用java语言编写的可重用组件

解决上诉问题:

- 需要一个配置文件(xml/properties)来配置我们的service层和dao层
  唯一标识=全限定类名(key-value结构)
- 通过配置类反射创建对象
  bean.properties文件

```
accountDao = com.aiqi.dao.impl.AccountDaoImpl
accountService = com.aiqi.service.impl.AccountServiceImpl
```

bean工厂,BeanFactory.java

```
public class BeanFactory{
    //定义一个properties对象

    private static Properties props;
```

```
    //静态代码块
    static{
        try{
            //实例化props对象
            props = new Properties();
            //获取流对象
            InputStream in =
BeanFactory.class.getClassLoader.getResourceAsStream("bean.properties");
            props.load(in);
        }catch(Exception e){
            throw new ExceptionInInitializerError("初始化properties失败!");
        }
    }

    //反射创建对象,根据bean的名称获取bean
    public static Object getBean(String beanName){
        Object bean = null;
        try{
            String beanPath = props.getProperty(beanName);
            bean = Class.forName(beanPath).newInstance();//每次调用默认构造函数
        }catch(Exception e){
            e.printStackTrace();
        }
        return bean;
    }
}
```

写完后,我们对2.1.1的案例二代码进行修改

```
//模拟表现层
public class AccountController{
    public static void main(){
      //AccountService accountService = new AccountServiceImpl();
      AccountService accountService =
(AccountService)BeanFactory.getBean("accountService");
      accountService.save();
    }
}

//模拟业务层
public class AccountServiceImpl implements AccountService{
    //private AccountDao accountDao = new AccountDaoImpl();
    private AccountDao accountDao = (AccountDao)BeanFactory.getBean("accountDao");
    public void save(){
      accountDao.save();
    }
}
```

2.1.3 改善BeanFactory工厂

上面的工厂有个问题,单例对象会出现线程问题,而多例对象比较消耗内存

> 多例:对象被创建多次, 执行效率没有单例对象高 单例:只被创建一次 ,从而类中的成员也就只会初始化一次。

我们再将BeanFactory工厂改造下

```
public class BeanFactory{
    //定义一个properties对象
    private static Properties props;

    //定义一个map,用来存储我们要创建的对象,称之为容器;
    private static Map<String,Object> beans = null;

    //静态代码块
    static{
        try{
            //实例化props对象
            props = new Properties();
            //获取流对象
            InputStream in =
BeanFactory.class.getClassLoader.getResourceAsStream("bean.properties");
            props.load(in);
            //实例化容器
            beans = new HashMap<String,Object>();
            //取出配置文件中所有的Key
            Enumeration keys = props.keys();
            //遍历枚举
            while (keys.hasMoreElements()){
                //取出每个Key
                String key = keys.nextElement().toString();
                / /根据key获取value
                string beanPath = props.getProperty(key);
                //反射创建对象
                object value = Class.forName(beanPath).newInstance();
                //把key和value存入容器中
                beans.put(key,value);
            }
        }catch(Exception e){
            throw new ExceptionInInitializerError("初始化properties失败!");
        }
    }

    //根据bean的名称获取容器(Map)中的bean
    public static Object getBean(String beanName){
        return beans.get(beanName);
    }
}
```

> 此时,获取bean对象就是单例的了

2.1.4 推导

```
//AccountService accountService = new AccountServiceImpl();
AccountService accountService =(AccountService)BeanFactory.getBean("accountService");
```

> 观察以上代码,原来我们通过new关键字创建对象,对对象创建的控制在程序员手中 而使用工厂构建容器获取对应的bean对象,因为我们将bean对象交给了工厂容器统一管理,使对对象的控制发生了转变,这就叫**控制反(翻)转(IOC--Inverse of Control)**

2.2 IOC概念

> **控制反转**: 同义词ioc (IOC) 一般指控制反转 控制反转(Inversion of Control,英文缩写为IoC)**把创建对象的权利交给框架**，是框架的重要特征，并非面向对象编程的专用术语。它包括依赖注入(Dependency Injection,简称DI)和依赖查找(Depenaency Lookup)。 明确ioc的作用: 削减计算机程序的耦合(解除我们代码中的依赖关系)。(**注意**:耦合只能削减而不能完全消除)

2.3 spring-ioc案例

2.3.1 搭建环境

> - 我们创建一个maven父项目工程SpringLearning,不需勾选骨架
> - 添加spring的maven依赖

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
```

> - 创建子项目 02IOC
> - 创建三层结构 Controller Service Dao 及其下的Account接口和实现类,代码略

2.3.2 编写spring配置文件

在resource文件夹下创建applicationContext.xml文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
  <!-- 把对象的创建交给spring来管理 -->
  <bean id="accountDao" class="com.aiqi.dao.impl.AccountDaoImpl"></bean>
  <bean id="accountService" class="com.aiqi.service.impl.AccountServiceImpl">
</beans>
```

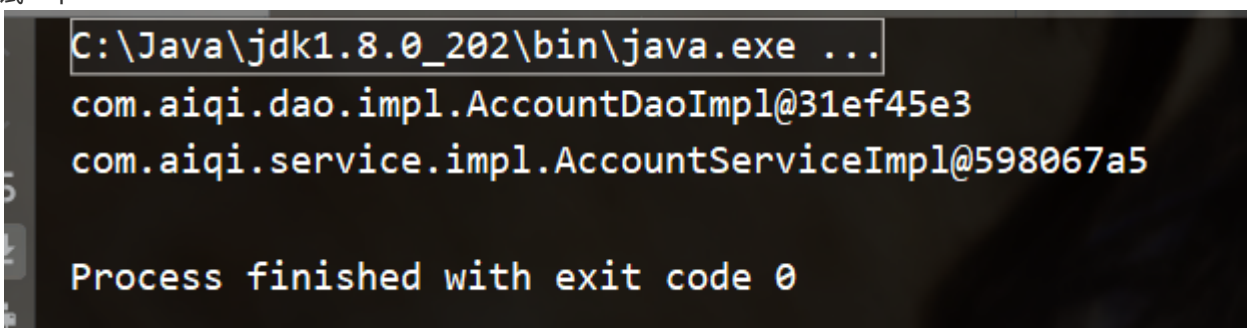
2.3.3 操作bean对象

在上一步中,我们已经配置了springIOC容器,并创建了对应的bean对象,接下来就可以使用了,我们在Controller层下,创建Client.java

```
package com.aiqi.controller;

import com.aiqi.dao.AccountDao;
import com.aiqi.service.AccountService;

import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @description: 模拟表现层调用业务层方法
 * @author: aiqi
 * @create: 2020-05-02 18:56
 **/
public class Client {

    /*
    * @Description: 获取spring核心容器IOC,并根据bean的id获取对象
    * @Param: [args]
    * @return: void
    **/
    public static void main(String[] args) {
        //1.获取核心容器对象
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        //2.根据id获取bean对象
        AccountDao accountDao = (AccountDao) applicationContext.getBean("accountDao");//
强转
        AccountService accountService =
applicationContext.getBean("accountService",AccountService.class);

        System.out.println(accountDao);
        System.out.println(accountService);
    }
}
```

测试一下



2.3.4 知识点

1.ApplicationContext的三个常用实现类

> ClassPathXmlApplicationContext 它可以加载类路径下的配置文件(要求配置文件必须在类路径下)
>
> FileSystemXmlApplicationContext 它可以加载磁盘任意路径下的配置文件(需要有操作权限)
>
> AnnotationConfigApplicationContext 它是用于读取注解创建容器的 第一个和第二个比,第一个常用

2.ApplicationContext和BeanFactory两个创建核心容器接口的区别

> 前者是立即创建:只要一读取完配置文件,就马上创建配置文件中配置的对象 单例对象适用 常用 后者是延迟加载:什么时候根据id获取bean了,才创建对象 多例对象适用

3.配置bean的三种方式

> applicationContext.xml

```
<!-- 配bean的第一种方式
使用默认构造函数(无参构造)创建:当该bean对象没有默认构造函数时,使用该方式配置bean,则会报错
-->
<bean id="accountDao" class="com.aiqi.dao.impl.AccountDaoImpl"></bean>
<bean id="accountService" class="com.aiqi.service.impl.AccountServiceImpl"></bean>

<!-- 配bean的第二种方式
        用于从工厂(某个类)的 普通方法 中获取创建对象
-->
<bean id="beanFactory" class="com.aiqi.factory.BeanFactory"></bean>
<bean id="accountService" factory-bean="beanFactory" factory-method="getAccountService">
</bean>

<!-- 配bean的第三种方式
    用于从工厂(某个类)的 静态方法 中获取创建对象
-->
<bean id="accountDao" class="com.aiqi.factory.BeanFactory" factory-
method="getAccountDao"></bean>
```

> BeanFactory.java

```
package com.aiqi.factory;

import com.aiqi.dao.AccountDao;
import com.aiqi.dao.impl.AccountDaoImpl;
import com.aiqi.service.AccountService;
import com.aiqi.service.impl.AccountServiceImpl;

/**
 * @description: 模拟bean工厂
 * @author: aiqi
 * @create: 2020-05-03 19:35
 **/
public class BeanFactory {
    public AccountService getAccountService(){
        return new AccountServiceImpl();
    }

    public static AccountDao getAccountDao(){
        return new AccountDaoImpl();
    }
}
```

4.bean的作用范围

> scope属性用于指定bean作用范围 取值有5个:
>
> - singleton 单例的(默认值)
> - prototype 多例的

> - request web应用的请求范围
> - session web应用的会话范围
> - global-session web应用的集群环境(全局)会话范围(当不是集群环境时,就是session)
>
> 我们常用的是singleton和prototype

例子:

```
<bean id="accountDao" class="com.aiqi.dao.impl.AccountDaoImpl" scope="prototype"></bean>
```

5.bean的生命周期

> singleton: 出生: 跟着spring容器的创建而产生 (解析完配置文件后产生) 活着: 只要容器还在,bean就一直活着 死亡: 容器销毁,bean对象消亡 (调用close方法销毁容器后,bean就死亡了)

> prototype: 出生: 当我们使用该bean对象时产生 (getBean方法) 活着: 对象在使用过程中就一直活着 死亡:当长时间不用时,java的垃圾回收机制将自动回收

2.4 依赖注入

2.4.1 概念

> DI(Dependency Injection):把有依赖关系的类放到容器中，解析出这些类的实例，就是依赖注入。目的是实现类的解耦。
>
> IoC的一个重点是在系统运行中，动态的向某个对象提供它所需要的其他对象(包括对象,资源,常量数据)。这一点是通过DI（Dependency Injection，依赖注入）来实现的
>
> 在当前类中需要用到其他类的对象,spring为我们提供,我们只需在xml文件中配置,依赖关系的维护就称之为依赖注入
>
> IoC和DI有什么关系呢？其实它们是同一个概念的不同角度描述，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以2004年大师级人物Martin Fowler又给出了一个新的名字：“依赖注入”，相对IoC 而言，“依赖注入”明确描述了“被注入对象依赖IoC容器配置依赖对象”。

2.4.2 知识点

1.可注入的数据类型

> - 基本类型和String
> - 其他bean类型(在xml中或注解配置过的bean对象)
> - 复杂类型/集合类型

2.注入方式

> 构造函数注入: (不常用) 优点: 在获取bean对象时,如果该对象没有默认构造(无参构造),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.
>
> set方法注入: (相比构造注入更常用) 优点: 创建bean对象时没有明确的限制,可以直接使用默认构造函数创建 缺点: 如果有某个成员变量必须有值,则获取对象时,有可能set方法没有执行,无法保证一定传值进行注入操作
>
> 注解注入:接下来讲注解开发时再了解

例子:

```
<!-- 构造函数注入
        使用constructor-arg标签
        属性: 表示key的有 type:参数类型,index:参数下标(从0开始),name:参数名(用的最多)
              表示value的有 value:参数值,ref:参数引用对象(其他bean类型的需要单独配bean,然后引
用)
    -->
    <bean id="accountService" class="com.aiqi.service.imp.AccountServiceImpl">
        <constructor-arg name="username" value="aiqi"></constructor-arg>
        <constructor-arg name="age" value="18"></constructor-arg>
        <constructor-arg name="date" ref="now"></constructor-arg>
    </bean>

    <bean id="now" class="java.util.Date"></bean>

    <!-- set方法注入
        使用property标签
        属性: 表示key的有 name:方法名(去除setXXX()方法的set)
              表示value的有 value:参数值,ref:参数引用对象(复杂类型的需要单独配bean,然后引用)
    -->
    <bean id="accountService1" class="com.aiqi.service.imp.AccountServiceImpl1">
        <property name="username" value="aiqi1"></property>
        <property name="age" value="19"></property>
        <property name="date" ref="now"></property>
    </bean>

    <!-- set方法注入集合类型
        集合类型分为两类:set和map
        所以 Array List Set 的 array,list,set标签可以随便互用
            Map Properties 的 map,props标签也可以互用
    -->
    <bean id="accountService2" class="com.aiqi.service.imp.AccountServiceImpl2">
        <property name="strArr">
            <array>
                <value>aiqi</value>
                <value>wyx</value>
            </array>
        </property>
        <property name="list">
            <list>
                <value>aiqi</value>
                <value>wyx</value>
            </list>
        </property>
        <property name="set">
            <set>
                <value>aiqi</value>
                <value>wyx</value>
            </set>
        </property>
        <property name="map">

            <map>
```

```
                <entry key="name1" value="aiqi"></entry>
<!--                <entry key="name2" value="wyx"></entry>-->
                <entry key="name2">
                    <value>wyx</value>
                </entry>
            </map>
        </property>
        <property name="props">
            <props>
                <prop key="name1">aiqi</prop>
                <prop key="name2">wyx</prop>
            </props>
        </property>
    </bean>
```

2.5 常用注解

在使用注解开发时,需要规定spring在创建容器时需要扫描的包

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

    <!-- 告知spring在创建容器时要扫描的包 -->
    <!-- 扫描com.aiqi包下所有使用注解的类型 -->
    <context:component-scan base-package="com.aiqi"/>

</beans>
```

> 我们在xml中是这样配置bean的

```
<bean id="accountService" class="com.aiqi.service.impl.AccountServiceImpl"
        init-method="" destroy-method="" scope="">
    <property name="" value="" ref=""></property>
</bean>
```

> 其对应的注解如下 1.用于创建对象 对应bean标签的 id,class等
>
> - 1.1**@Component** 用于将该对象存入spring ioc容器中
>   它有value属性,可以省略,表示id
>   不写的话,默认值是类名,首字母小写
> - 1.2**@Controller** 一般用于表现层
>   **@Service** 一般用于业务层
>   **@Repository** 一般用于持久层
>   以上三个注解他们的属性和用法和Component一样,它们是Spring框架为我们提供的三层开发所使用的

> 注解,
> 使我们的三层对象更加清晰

> 2.用于注入数据 对应property标签
>
> - 2.1 用于注入其他bean类型(自定义对象类型)
> **@Autowired** 自动按照类型注入 容器中有唯一的一个bean对象类型与要注入的变量类型匹配时,就可以直接注入成功
> 如果IOC容器中没有任何bean类型和要注入的变量类型匹配,就会报错
> 如果有多个匹配的话,首先按照类型匹配找到多个bean对象,然后在通过bean的id(key)找到那个bean对象,如果没有找到,则报错
> 书写位置:可以是属性上,可以是方法上,(即类成员变量上)
> 注意:当使用注解注入时,set方法就不是必须的了(不用写了)
> **@Qualifier** 解决@Autowired匹配多个bean对象的问题
> 按照类型注入的基础上,再按照名称注入.它在给类成员注入时不能单独使用(要和@Autowired配合使用),但是在给方法参数注入时可以(见2.6.2 06项目JdbcConfig)
> 它有value属性,用于指定注入bean对象的id
> **@Resource**
> 前两者都有缺陷的地方
> 它可以通过匹配bean对象的id进行注入,而且可以单独使用
> 它有name属性,用于指定注入bean对象的id
> - 2.2 用于注入基本数据类型和String类型
> **@Value()**
> 它有value属性,用于指定数据的值,而且它可以使用spring的spEL(也就是spring的EL表达式),写法:${express}
> - 2.3 集合类型只能通过xml配置注入

> 3.用于改变作用范围 对应bean标签的scope属性 **@Scope** 它有value属性,指定范围的取值:常用取值singleton和prototype(不写默认singleton) (分别对应单例和多例)
>
> 4.和生命周期相关 对应bean标签的init-method,destroy-method等 **@PostConstruct** 用于指定对象创建的方法 **@PreDestroy** 用于指定对象销毁的方法

2.6 IOC和DI练习单表CRUD

> 环境搭建 在本次练习中,我们使用c3p0数据池,dbutils数据库连接对象,spring-test整合junit进行测试 导入相关依赖(父级项目中的pom.xml)

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
<dependency>
    <groupId>commons-dbutils</groupId>
    <artifactId>commons-dbutils</artifactId>
    <version>1.4</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
```

```
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>
</dependency>
<dependency>
    <groupId>com.mchange</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.5.3</version>
</dependency>
```

> 创建子项目 05XML_IOC_CRUD 及 06Anno_IOC_CRUD

2.6.1 基于xml

部分代码如下

- AccountDaoImpl.java

```
package com.aiqi.dao.impl;

import com.aiqi.dao.AccountDao;
import com.aiqi.pojo.Account;
import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.handlers.BeanHandler;
import org.apache.commons.dbutils.handlers.BeanListHandler;

import java.util.List;

/**
 * @description: 账户持久层接口实现类
 * @author: aiqi
 * @create: 2020-05-02 18:49
 **/
public class AccountDaoImpl implements AccountDao {
    //dbutils
    private QueryRunner runner;

    //set方法依赖注入
    public void setRunner(QueryRunner runner) {
        this.runner = runner;
    }

    public void add(Account account) {
        try {
            runner.update("insert into t_test (name,money) values
(?,?)",account.getName(),account.getMoney());
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }
```

```
    public void delete(Integer id) {
        try {
            runner.update("delete from t_test where id =?",id);
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }

    public void update(Account account) {
        try {
            runner.update("update t_test set name=?,money=? where
id=?",account.getName(),account.getMoney(),account.getId());
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }

    public List<Account> selectAll() {
        try {
            return runner.query("select * from t_test",new BeanListHandler<Account>
(Account.class));
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }

    public Account selectById(Integer id) {
        try {
            return runner.query("select * from t_test where id = ?",new
BeanHandler<Account>(Account.class),id);
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }
}
```

- AccountServiceImpl.java

```
package com.aiqi.service.impl;

import com.aiqi.dao.AccountDao;
import com.aiqi.pojo.Account;
import com.aiqi.service.AccountService;

import java.util.List;

/**
 * @description: 账户业务层接口实现类
 * @author: aiqi
 * @create: 2020-05-02 18:51
 **/
public class AccountServiceImpl implements AccountService {
```

```
    private AccountDao accountDao;

    //set方法依赖注入
    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }


    public void add(Account account) {
        accountDao.add(account);
    }

    public void delete(Integer id) {
        accountDao.delete(id);
    }

    public void update(Account account) {
        accountDao.update(account);
    }

    public List<Account> selectAll() {
        return accountDao.selectAll();
    }

    public Account selectById(Integer id) {
        return accountDao.selectById(id);
    }
}
```

- applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- 配dao -->
    <bean id="accountDao" class="com.aiqi.dao.impl.AccountDaoImpl">
        <property name="runner" ref="runner"></property> <!-- 注入QueryRunner -->
    </bean>

    <!-- 配service -->
    <bean id="accountService" class="com.aiqi.service.impl.AccountServiceImpl">
        <property name="accountDao" ref="accountDao"></property> <!-- 注入dao -->
    </bean>

    <!-- 配置dbutils的QueryRunner -->
    <bean id="runner" class="org.apache.commons.dbutils.QueryRunner" scope="prototype">
        <!-- 注意这里要使用多例prototype,因为当多个dao使用单个QueryRunner时,会产生线程不安全问题
-->
        <constructor-arg name="ds" ref="dataSource"></constructor-arg> <!-- 注入数据源 -->
    </bean>
```

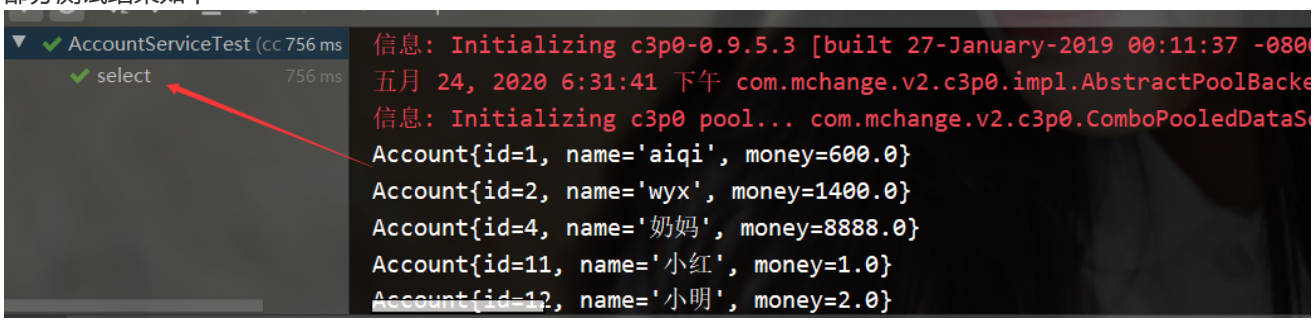
```
    <!-- 配c3p0数据源 -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <!-- 注入连接数据库的必备信息 -->
        <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
        <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/db_spring">
</property>
        <property name="user" value="root"></property>
        <property name="password" value="root"></property>
    </bean>
</beans>
```

- 测试类AccountServiceTest.java

```
package com.aiqi.test;

import com.aiqi.pojo.Account;
import com.aiqi.service.AccountService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.List;

/**
 * @description: 账户测试类
 * @author: aiqi
 * @create: 2020-05-03 22:32
 **/
/*
用spring整合junit测试
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:applicationContext.xml")
public class AccountServiceTest {

    @Autowired
    private AccountService accountService;

    //没和springjunit整合之前这么写
//    @Before
//    public void init(){
//        //1.根据xml配置获取核心容器对象
//        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
//        //2.根据id获取bean对象
//        accountService =(AccountService) applicationContext.getBean("accountService");
//    }

    @Test
    public void add(){
```

```
        Account account = new Account();
        account.setName("奶妈");
        account.setMoney(3000f);
        accountService.add(account);
    }
    @Test
    public void delete(){
        accountService.delete(3);
    }
    @Test
    public void update(){
        Account account = accountService.selectById(1);
        account.setMoney(1314f);
        accountService.update(account);
        System.out.println(account);
    }
    @Test
    public void select(){
        List<Account> accounts = accountService.selectAll();
        for (Account account : accounts) {
            System.out.println(account);
        }
    }
}
```

部分测试结果如下



2.6.2 基于注解

当完全不使用xml时,需要编写配置类,这时我们需要使用到新注解

> 1.@Configuration 指定当前类是一个配置类 当配置类作为AnnotationConfigApplicationContext对象创建的参数时,该注解可以不写 如:new AnnotationConfigApplicationContext(SpringConfig.class);

> 2.@ComponentScan 用于通过注解指定spring在创建IOC容器时要扫描的包 属性:value,它和basePackages的作用是一样的,指定要扫描的包 相当于xml配置中的 @ComponentScans 指定多个@ComponentScan

> 3.@Bean 将方法的返回值作为bean对象存入springIOC容器中 它有name和value属性,作用一样,指定该返回值bean对象的id,不写时,默认id是该方法的名称 注意:当该方法有参数时,spring会去IOC容器中找该bean对象,查找的方式和@Autowired的一样

> 4.@Import 用于导入其他配置类 属性:value 指定其他配置类的字节码文件 有@Import注解的类是父配置类,导入的配置类是子配置类

> 5.@PropertySource 用于指定properties文件的位置 属性:value 指定文件的名称和路径 classpath关键字,表示类路径下

> 6.@Qualifier 当有多个相同类型的bean对象时,指定传入的bean的id 详情见JdbcConfig

SpringConfig

```
package com.aiqi.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.PropertySource;

/**
 * @description: Spring的主配置类， 替代xml配置
 * @author: aiqi
 * @create: 2020-05-05 13:57
 **/
/*
 我们在实际开发中,如果项目没有要求的话,大多数时候,自己写的类用注解配置,其他jar包的类用xml配置,更方便
 */

@Configuration
@ComponentScan("com.aiqi")
@Import(JdbcConfig.class)
@PropertySource("classpath:jdbc.properties")
public class SpringConfig {}
```

JdbcConfig

```
package com.aiqi.config;

import com.mchange.v2.c3p0.ComboPooledDataSource;
import org.apache.commons.dbutils.QueryRunner;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;

import javax.sql.DataSource;

/**
 * @description: 与数据库连接相关的配置类
 * @author: aiqi
 * @create: 2020-05-05 17:11
 **/

@Configuration
public class JdbcConfig {

    @Value("${jdbc.driver}")
```

```
    private String driver;

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    /*
     * @Description: 用于创建QueryRunner对象,并把它放入springIOC容器中
     * @Param: [dataSource]
     * @return: org.apache.commons.dbutils.QueryRunner
     **/
    @Bean("runner")
    @Scope("prototype")   //设置为多例,线程安全
    public QueryRunner getQueryRunner(@Qualifier("ds1") DataSource dataSource){
        return new QueryRunner(dataSource);   //@Qualifier 当匹配多个相同类型的bean对象时,指定注入的bean对象的id
    }

    /*
    * @Description: 创建c3p0数据源对象,并传入Spring的IOC容器
    * @Param: []
    * @return: com.mchange.v2.c3p0.ComboPooledDataSource
    **/
    @Bean("ds1")
    public ComboPooledDataSource getDataSource(){
        try {
            ComboPooledDataSource cpds = new ComboPooledDataSource();
            cpds.setDriverClass(driver);
            cpds.setJdbcUrl(url);
            cpds.setUser(username);
            cpds.setPassword(password);
            return cpds;
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }

    @Bean("ds2")
    public ComboPooledDataSource getDataSource1(){
        try {
            ComboPooledDataSource cpds = new ComboPooledDataSource();
            cpds.setDriverClass(driver);
            cpds.setJdbcUrl("jdbc:mysql://localhost:3306/db_spring");
            cpds.setUser(username);
            cpds.setPassword(password);
            return cpds;
        }catch (Exception e){

            throw new RuntimeException(e);
```

```
        }
    }
}
```

AccountDaoImpl.java

```
package com.aiqi.dao.impl;

import com.aiqi.dao.AccountDao;
//...导包略

/**
 * @description: 账户持久层接口实现类
 * @author: aiqi
 * @create: 2020-05-02 18:49
 **/

@Repository("accountDao")
public class AccountDaoImpl implements AccountDao {
    @Autowired
    private QueryRunner runner;

    public void add(Account account) {
        try {
            runner.update("insert into t_test (name,money) values
(?,?)",account.getName(),account.getMoney());
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }

    //其他方法略,见2.6.1
}
```

AccountServiceImpl.java

```
package com.aiqi.service.impl;

import com.aiqi.dao.AccountDao;
//...导包略

/**
 * @description: 账户业务层接口实现类
 * @author: aiqi
 * @create: 2020-05-02 18:51
 **/
@Service("accountService")
public class AccountServiceImpl implements AccountService {
    @Autowired

    private AccountDao accountDao;
```

```
    public void add(Account account) {
        accountDao.add(account);
    }

    //....其他方法略
}
```

测试类AccountServiceTest.java

```
package com.aiqi.test;

import com.aiqi.config.SpringConfig;
import com.aiqi.pojo.Account;
import com.aiqi.service.AccountService;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.List;

/**
 * @description: Junit账户测试类
 * @author: aiqi
 * @create: 2020-05-03 22:32
 **/

/*
spring整合junit配置
    1.pom导入spring-test
    2.使用junit提供的一个注解(@RunWith)把原有的main方法替换了,替换成spring提供的
    3.告知spring的运行器,spring的IOC容器创建是基于xml还是注解,并且说明位置
        @ContextConfiguration
        属性:locations 指定xml文件的路径,要加上classpath关键词,表示在类路径下
        classes 指定注解配置类所在位置
    (注意:当我们使用spring5.x.x版本时,要求junit版本必须是4.12及以上)
 */

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = SpringConfig.class)
public class AccountServiceTest {

    @Autowired
    private AccountService accountService;

    @Test
    public void add(){
        Account account = new Account();
        account.setName("奶妈");
        account.setMoney(3000f);
        accountService.add(account);
```

```
    }

    //其他测试方法略...
}
```

测试结果正确,图略...

注意spring整合junit的配置,如上

三、AOP

3.1 银行转账案例

```
/*
     * @Description:  模拟转账
     * @Param: [sourceName, targetName, money] 转出账户名,转入账户名,金额
     * @return: void
     **/
    public void transfer(String sourceName, String targetName, Float money) {

        //1.根据名称查询转出账户
        Account sourceAccount = accountDao.selectByName(sourceName);
        //2.根据名称查询转入账户
        Account targetAccount = accountDao.selectByName(targetName);
        //3.转出账户减钱
        sourceAccount.setMoney(sourceAccount.getMoney() - money);
        //4.转入账户加钱
        targetAccount.setMoney(targetAccount.getMoney() + money);
        //5.更新转出账户
        accountDao.update(sourceAccount);
        //int i = 1/0;
        //6.更新转入账户
        accountDao.update(targetAccount);
    }
```

查看上面的AccountServiceImpl.java的一部分有关转账代码,我们发现执行是没有问题的,转账成功,但我们在5-6步中模拟一个异常,这将会导致一个严重错误,转出者的钱减少了,而转入者的钱并没有更新,即没有到账.

所以我们需要进行数据库事务控制 编写代码 ConnectionUtil.java

```
package com.aiqi.util;

import javax.sql.DataSource;
import java.sql.Connection;

/**
 * @description: 数据库连接事务管理工具类  用于从数据源中获取一个连接,并且实现和线程的绑定
 * @author: aiqi
 * @create: 2020-05-05 19:07
 **/
public class ConnectionUtil {

    private ThreadLocal<Connection> tl = new ThreadLocal<Connection>();
```

```
    private DataSource dataSource;

    //spring set方法依赖注入
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    /*
    获取当前线程上的连接
    */
    public Connection getThreadConnection(){
        try {
            //1.先从ThreadLocal上获取
            Connection conn = tl.get();
            //2.判断当前线程上是否有连接
            if(conn==null){
                //3.没有的话,先从数据源获取一个连接,然后存入tl中
                conn = dataSource.getConnection();
                tl.set(conn);
            }
            //4.返回当前线程上的连接
            return conn;
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }

    /*
        将连接与线程解绑
     */
    public void removeConnection(){
        tl.remove();
    }
}
```

TransactionManager.java

```
package com.aiqi.util;

/**
 * @description: 与事务管理相关的工具类
 * @author: aiqi
 * @create: 2020-05-05 19:24
 **/
public class TransactionManager {
    private ConnectionUtil connectionUtil;

    //spring set方法注入
    public void setConnectionUtil(ConnectionUtil connectionUtil) {
        this.connectionUtil = connectionUtil;
    }
```

```
    //开启事务
    public void open(){
        try {
            connectionUtil.getThreadConnection().setAutoCommit(false);
        }catch (Exception e){
            e.printStackTrace();
        }
    }
    //提交事务
    public void commit(){
        try {
            connectionUtil.getThreadConnection().commit();
        }catch (Exception e){
            e.printStackTrace();
        }
    }
    //回滚事务
    public void rollback(){
        try {
            connectionUtil.getThreadConnection().rollback();
        }catch (Exception e){
            e.printStackTrace();
        }
    }
    //释放连接
    public void release(){
        try {
            connectionUtil.getThreadConnection().close();  //这个不是真的把连接关闭了,而是将
连接返回连接池中
            connectionUtil.removeConnection();
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

当写完这两个有关事务管理的工具类后,我们修改下业务层的转账代码

```
private AccountDao accountDao;

private TransactionManager tm;

//set方法依赖注入
public void setAccountDao(AccountDao accountDao) {
  this.accountDao = accountDao;
}
public void setTm(TransactionManager tm) {
  this.tm = tm;
}

public void transfer(String sourceName, String targetName, Float money) {
        try {
```

```
            //1.开启事务
            tm.open();
            //2.执行操作
                //1.根据名称查询转出账户
                Account sourceAccount = accountDao.selectByName(sourceName);
                //2.根据名称查询转入账户
                Account targetAccount = accountDao.selectByName(targetName);
                //3.转出账户减钱
                sourceAccount.setMoney(sourceAccount.getMoney()-money);
                //4.转入账户加钱
                targetAccount.setMoney(targetAccount.getMoney()+money);
                //5.更新转出账户
                accountDao.update(sourceAccount);
                //int i = 1/0;
                //6.更新转入账户
                accountDao.update(targetAccount);
            //3.提交事务
            tm.commit();
            //4.返回结果
        }catch (Exception e){
            //5.回滚操作
            tm.rollback();
            e.printStackTrace();
        }finally {
            //6.释放连接
            tm.release();
        }
    }
```

当我们业务层还有其他方法也需要实现事务控制时,需要跟上面的写法一样,这样我们发现太麻烦了,于是想到了**代理模式**,抽取重复代码,进行代码的增强.

3.2 动态代理模式

> 动态代理: 特点:字节码随用随创建，随用随加载 作用:不修改源码的基础上对方法增强 分类: 基于接口的动态代理 基于子类的动态代理

3.2.1 基于接口

> 基于接口的动态代理: 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 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 newProxyInstance方法的参数: Classloader:类加载器 它是用于加载代理对象字节码的。和被代理对象使用相同的类加载器。固定写法。 Class[]:字节码数组 它是用于让代理对象和被代理对象有相同方法。固定写法。 InvocationHandler:用于提供增强的代码 它是让我们写如何代理。我们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，但不是必须的 此接口的实现类都是谁用谁写。

例子:

```
IProducer proxyProducer =
(IProducer)Proxy.newProxyInstance(producer.getClass().getClassLoader(),
producer.getClass().getInterfaces(),
new InvocationHandler(){
    /**
```

```
    *
    作用:执行被代理对象的任何接口方法都会经过该方法
    *
    方法参数的含义
    *@param proxy 代理对象的引用
    *@param method 当前执行的方法
    *@param args  当前执行方法所需的参数
    * @return  和被代理对象方法有相同的返回值
    * @throws Throwable
    */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        return method.invoke(producer,args);
    }
);
```

缺陷:被代理的类必须是一个接口,那么普通的java类能不能被代理呢,答案是可以的,如下

3.2.2 基于子类

要实现基于子类的代理,需要导入第三方jar包 cglib

```
<dependency>
  <groupId>cglib</groupId>
  <artifactId>cglib</artifactId>
  <version>2.1_3</version>
</dependency>
```

> 基于子类的动态代理: 涉及的类: Enhancer 提供者: 第三方jar包 cglib 如何创建代理对象: 使用Enhancer类中的creat方法 创建代理对象的要求: **被代理类不能是最终类**(最终类不能创建子类,因此也无法创建代理对象) creat方法的参数: Class:字节码 它是用于指定被代理对象的字节码。固定写法。 Callback:用于提供增强的代码 它是让我们写如何代理。我们一般都是些一个该接口的实现类，通常情况下都是匿名内部类，但不是必须的, 此接口的实现类都是谁用谁写。我们一般是写该接口的子接口实现类 MethodInterceptor

例子:

```
Producer cglibProducer = (Producer)Enhancer.create(producer.getClass(), new
MethodInterceptor() {
    /**
    执行被代理对象的任何方法都会经过该方法
    @param proxy  代理对象的引用
    @param method 当前执行的方法
    @param args  当前执行方法所需的参数
    @param methodProxy  当前执行方法的代理对象
    @return
    @throws Throwable
    **/
    @Override
    public Object intercept (0bject proxy, Method method, Object[] args, MethodProxy
methodProxy) throws Throwable {

        return method.invoke(producer,args);
```

```
    }
});
```

3.2.3 使用

了解完动态代理后,我们完善下转账案例(使用基于接口的动态代理) 编写ServiceProxy.java

```
package com.aiqi.proxy;

import com.aiqi.service.AccountService;
import com.aiqi.util.TransactionManager;

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

/**
 * @description: 动态代理 service对象
 * @author: aiqi
 * @create: 2020-05-06 18:04
 **/
public class ServiceProxy {
    private AccountService accountService;
    private TransactionManager tm;

    //set方法依赖注入
    public final void setAccountService(AccountService accountService) {
        this.accountService = accountService;
    }

    public void setTm(TransactionManager tm) {
        this.tm = tm;
    }

    //获取动态代理对象AccountService
    public AccountService getAccountService() {
        return (AccountService)
Proxy.newProxyInstance(accountService.getClass().getClassLoader(),
                accountService.getClass().getInterfaces(), new InvocationHandler() {
                    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                        Object returnValue = null;
                        try {
                            //1.开启事务
                            tm.open();
                            //2.执行操作
                            returnValue = method.invoke(accountService,args);
                            //3.提交事务
                            tm.commit();
                            //4.返回结果
                            return returnValue;
                        }catch (Exception e){
```

```
                            //5.回滚操作
                            tm.rollback();
                            throw new RuntimeException(e);
                        }finally {
                            //6.释放连接
                            tm.release();
                        }
                    }
                });
    }
}
```

3.3 AOP概念

3.3.1 AOP是什么

> AOP:全称是Aspect Oriented Programming 即:面向切面编程
>
> 百度知道上这样描述: 在软件业，AOP为Aspect Oriented Programming的缩写;意为: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，是软件开发中的一个热点，也是Spring框架中的一一个重要内容,是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离,从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。
>
> 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。

3.3.2 AOP的作用及优势

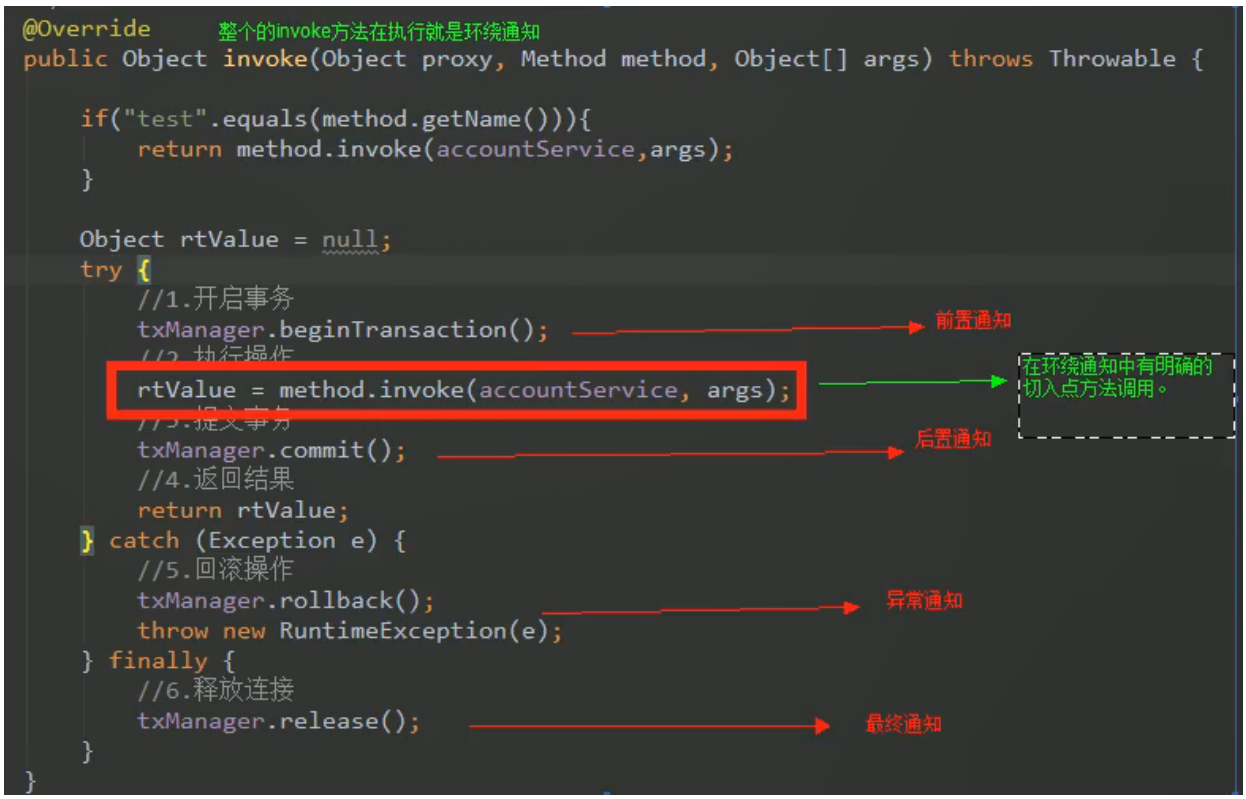
> 作用: 在程序运行期间，不修改源码对已有方法进行增强。
>
> 优势: 减少重复代码 提高开发效率 维护方便

3.3.3 AOP的相关术语

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

> Pointcut (切入点): 所谓切入点是指我们要对哪些Joinpoint进行拦截的定义。 (**注意**:所有切入点都是连接点,而连接点不一定是切入点,切入点是被动态代理增强的方法)

> Advice (通知/增强) : 所谓通知是指拦截到Joinpoint之后所要做的事情就是通知。 通知的类型:前置通知，后置通知，异常通知，最终通知，环绕通知。



> 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一一些方法或Field字段。

> Target(目标对象) : 代理的目标对象。

> 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。 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。

> 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。

3.4 spring-AOP的使用

3.4.1 基于XML

配置步骤

> 1.把通知bean对象也交给ioc容器管理

> 2.使用aop:config标签表示开始aop的配置

> 3.使用aop:aspect标签表示开始切面的配置

```
属性: id 表示给切面提供一个唯一标识
ref 指定通知类的bean对象id
```

> 4.在aop:aspect标签内部使用对应标签来配置通知类型

```
aop:before 前置通知:在切入点方法执行之前执行
aop:after-returning 后置通知:切入点方法执行成功后,才会执行后置通知
aop:after-throwing 异常通知:切入点方法执行产生异常,才会执行异常通知
aop:after 最终通知:无论切入点方法是否正常执行,最终通知都会执行
aop:around 环绕通知:spring给我们提供的自己控制增强方法在哪执行(见Logger里面的注释)
method属性  用于指定该通知类中哪个方法是前置通知
pointcut属性 用于指定切入点表达式 该表达式的含义是指对哪些切入点(方法)进行增强
pointcut-ref属性 引用公共切入点表达式
```

注意: 其中后置通知和异常通知只会执行其中一个

> 5.aop:pointcut标签 定义公共切入点表达式 注意 当该标签写在aop:aspect标签内部时,只能在当前切面使用
> 当写在外部时,需要在所有aop:aspect标签之前定义

例子:

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

<!-- 配置AccountService -->
<bean id="accountService" class="com.aiqi.service.impl.AccountServiceImpl"></bean>

<!-- 配置Logger通知类 -->
<bean id="logger" class="com.aiqi.util.Logger"></bean>
<!-- 配置AOP -->
<aop:config>
    <!--切面表达式写在切面外部时,需要在所有aop:aspect标签之前定义 -->
    <!--<aop:pointcut id="pt1" expression="execution(* com.aiqi.service.impl.*.*
(..))"/>-->
    <!-- 配置切面 -->
    <aop:aspect id="loggerAdvice" ref="logger">
    <!-- 配置通知类型,并且配置通知方法和切入点方法的关联 -->
    <!-- 前置通知 -->
    <!--<aop:before method="beforeLog" pointcut="execution(* com.aiqi.service.impl.*.*
(..))"></aop:before>-->
    <!--后置通知-->
    <!--<aop:after-returning method="afterLog" pointcut-ref="pt1"></aop:after-
returning>-->
    <!--异常通知-->
    <!--<aop:after-throwing method="errorLog" pointcut-ref="pt1"></aop:after-throwing>--
>
    <!--最终通知-->
    <!--<aop:after method="finalLog" pointcut-ref="pt1"></aop:after>-->
    <!-- 环绕通知 -->
    <aop:around method="aroundLog" pointcut-ref="pt1"></aop:around>
    <!-- 配置公共切入点表达式 id为该表达式的唯一标识  expression是切入点表达式内容-->
    <aop:pointcut id="pt1" expression="execution(* com.aiqi.service.impl.*.*(..))"/>
```

```
    </aop:aspect>
</aop:config>
</beans>
```

```
package com.aiqi.util;

import org.aspectj.lang.ProceedingJoinPoint;

/**
 * @description: 打印日志的类(通知类)
 * @author: aiqi
 * @create: 2020-05-06 20:16
 **/
public class Logger {
    //前置通知
    public void beforeLog(){
        System.out.println("执行方法前");
    }

    //后置通知
    public void afterLog(){
        System.out.println("执行方法后");
    }
    //异常通知
    public void errorLog(){
        System.out.println("出现了异常");
    }

    //最终通知
    public void finalLog(){
        System.out.println("方法执行结束");
    }

    //环绕通知
    /*
       1.出现的问题:按照之前的写法
        public void aroundLog(){
            System.out.println("环绕通知");
        }
        执行测试方法后,发现aroundLog执行了,但切入点方法没有执行
        2.分析:在之前学的动态代理中,我们通过method.invoke()执行了切入点方法,而这里并没有执行切入点
方法

        3.解决:
            Spring给我们提供了一个接口ProceedingJoinPoint,该接口可作为环绕通知的方法参数.
            在程序执行时,Spring会为我们提供该接口的实现类供我们使用

        环绕通知:spring为我们提供的一种可以在代码中手动控制增强方法何时执行的方式
     */
    public Object aroundLog(ProceedingJoinPoint pjp){
        Object returnValue = null;
        try {
            System.out.println("前置通知");
```

```
            //得到切入点方法运行所需参数
            Object[] args = pjp.getArgs();
            //执行切入点方法
            returnValue = pjp.proceed(args);

            System.out.println("后置通知");

            return returnValue;
        }catch (Throwable t){   //这里要用Throwable关异常,Exception关不住,(需要再去了解下)
            System.out.println("异常通知");
            throw new RuntimeException(t);
        }finally {
            System.out.println("最终通知");
        }

    }
}
```

知识点:

> 切入点表示式语法 pom导入依赖

```
<dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjweaver</artifactId>
<version>1.8.13</version>
</dependency>
```

> 关键字:execution(表达式) 表达式: 访问修饰符 返回值 包名.类名.方法名(参数列表) 标准的表达式写法 例如:execution(public void com.aiqi.service.impl.AccountServiceImpl.add()) 其中 访问修饰符可以省略 返回值可以写通配符 ***** 表示任意返回值 包名可以使用通配符 ***.** 表示任意包 有几级包就写几个通配符 例如 com.aiqi.util ==> ..*. 也可以使用通配符 ***..** 表示任意包及其子包 类名和方法名可以使用通配符 ***** 表示任意类名和方法名 参数列表 可以直接写数据类型 基本类型直接写名称 如 int 引用类型写 包名.类名 如 java.lang.String 可以使用通配符 ***** 表示任意类型参数 **..** 表示有无参数均可,当有参数时参数可以是任意类型 全通配型写法(实际开发中尽量不用) **execution(* ...*(..))** 在实际开发中,切入点表达式的通常写法: 切到业务层实现类下的所有方法 例如 *** com.aiqi.service.impl..(..)**

3.4.2 基于注解

这节我们用注解AOP完善下银行转账案例

```
<aop:aspectj-autoproxy></aop:aspectj-autoproxy> <!--表示开启spring对注解AOP的支持-->
```

在完善案例前首先学几个有关AOP的注解:

> @Component 将该类注册到springIOC容器中 @Aspect 表明该类是一个切面(通知)类 @Pointcut 定义一个切面表达式 @Before() 前置通知 @AfterReturning() 后置通知 @AfterThrowing() 异常通知 @After() 最终通知 @Around() 环绕通知 @EnableAspectJAutoProxy 开启spring对注解AOP的支持

> **注意**: 1.spring的通过注解配置的通知方法执行顺序有点问题: 前置通知-->执行切入点方法-->最终通知-->后置通知/异常通知 而用环绕通知,我们自己写通知执行的顺序就没有任何问题了(所以注解AOP,一般环绕通知用得多) 2.传入切入点表达式时,需要打括号执行 例如:@Before("pt1()") 必须打括号,不然会报错

完善代码: 1.编写通知类

```
package com.aiqi.util;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

/**
 * @description: 事务通知类
 * @author: aiqi
 * @create: 2020-05-08 15:53
 **/
@Component("tn")
@Aspect  //指定该类是一个切面类
public class TransactionNotice {
    @Autowired
    private TransactionManager tm;

    //定义切面表达式
    @Pointcut("execution(* com.aiqi.service.impl.*.*(..))")
    private void pt1(){}

    //环绕通知
    @Around("pt1()")
    public Object aroundLog(ProceedingJoinPoint pjp){
        Object returnValue = null;
        try {
            //1.开启事务
            tm.open();
            //2.执行切入点方法
            //得到切入点方法运行所需参数
            Object[] args = pjp.getArgs();
            //运行方法
            returnValue = pjp.proceed(args);
            //3.提交事务
            tm.commit();
            //4.返回结果
            return returnValue;
        }catch (Throwable t){
            //5.回滚操作
            tm.rollback();
            throw new RuntimeException(t);
        }finally {
            //6.释放连接
            tm.release();
        }

    }
```

```
}
```

四、JdbcTemplate

4.1 简介

JdbcTemplate简介

> Spring对数据库的操作在jdbc上面做了深层次的封装，使用spring的注入功能，可以把DataSource注册到JdbcTemplate之中。

> JdbcTemplate位于spring-jdbc包中。其全限定命名为org.springframework.jdbc.core.JdbcTemplate。要使用JdbcTemlate还需一个spring-tx这个包包含了一下事务和异常控制

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
```

4.2 使用

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- 配JdbcTemplate -->
    <bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"></property>
    </bean>
    <!-- 配spring内置数据源 -->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!-- 注入连接数据库的必备信息 -->
        <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
        <property name="url" value="jdbc:mysql://localhost:3306/db_spring"></property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
    </bean>
    <!-- 配dao层 -->
    <bean id="accountDao" class="com.aiqi.dao.impl.AccountDaoImpl">
        <property name="jt" ref="jt"></property>

    </bean>
```

```
</beans>
```

AccountDaoImpl.java

```
package com.aiqi.dao.impl;

import com.aiqi.dao.AccountDao;
import com.aiqi.pojo.Account;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

import java.util.List;

/**
 * @description: 账户持久层接口实现类
 * @author: aiqi
 * @create: 2020-05-08 18:42
 **/
public class AccountDaoImpl implements AccountDao {
    private JdbcTemplate jt;
    //依赖注入
    public void setJt(JdbcTemplate jt) {
        this.jt = jt;
    }

    public void add(Account account) {
        jt.update("insert into t_test (name,money) values
(?,?)",account.getName(),account.getMoney());
    }

    public void delete(Integer id) {
        jt.update("delete from t_test where id =?",id);
    }

    public void update(Account account) {
        jt.update("update t_test set name=?,money=? where
id=?",account.getName(),account.getMoney(),account.getId());
    }

    public Account findById(Integer id) {
        List<Account> accounts = jt.query("select * from t_test where id = ?",new
BeanPropertyRowMapper<Account>(Account.class),id);
        return accounts.isEmpty()?null:accounts.get(0);
    }

    public List<Account> findAll() {
        return jt.query("select * from t_test",new BeanPropertyRowMapper<Account>
(Account.class));
    }

    public Long findCount() {
        return jt.queryForObject("select count(*) from t_test",Long.class);
    }
```

```
}
```

测试下Demo3.java

```
package com.aiqi.jdbctemplate;

import com.aiqi.pojo.Account;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * @description: JdbcTemplate的单表CRUD
 * @author: aiqi
 * @create: 2020-05-08 17:30
 **/
/*
增删改update方法返回的是受到影响的行数
查query方法用得最多的有两个重载方法,返回List集合
query(String sql,RowMapper<T> rowMapper,Object...args)  在jdk1.5版本后使用(因为可变参数在
jdk1.5才出来)
query(String sql,Object[]args,RowMapper<T> rowMapper) 任意版本都能使用
 */
public class Demo3 {
    public static void main(String[] args) {
        //1.获取容器
        ApplicationContext ac = new
ClassPathXmlApplicationContext("applicationContext.xml");
        //2.获取对象
        JdbcTemplate jt = ac.getBean("jt",JdbcTemplate.class);
        //3.执行操作
        //增
        //jt.update("insert into t_test(name,money) values (?,?)","小红",1111);
        //删
        //jt.update("delete from t_test where name like ?","%小%");
        //改
        //jt.update("update t_test set name=?,money=? where id=?","奶妈",8888,4);
        //查
        //查多个
//        List<Account> accounts = jt.query("select * from t_test where name like ?",new
AccountRowMapper(),"%小%");
//        List<Account> accounts1 = jt.query("select * from t_test where name like
?",new BeanPropertyRowMapper<Account>(Account.class),"%小%");
//        for (Account account : accounts1) {
//            System.out.println(account);
//        }
```

```
        //查一个
    /*Account account = jt.query("select * from t_test where id = ?",new
AccountExtractor(),1);
    System.out.println(account);*/
        //查询返回一行一列(使用聚合函数)
        long len = jt.queryForObject("select count(*) from t_test",Long.class);
        System.out.println(len);
    }

    //使用注解开发时,DaoImpl实现类可以通过Autowired注入JdbcTemplate
    //使用xml开发时,DaoImpl实现类可以继承JdbcDaoSupport类,减少set方法依赖注入 代码的耦合
}


/*
自定义Account集合的封装策略
 */
class AccountRowMapper implements RowMapper<Account> {

    public Account mapRow(ResultSet resultSet, int i) throws SQLException {
        Account account = new Account();
        account.setId(resultSet.getInt("id"));
        account.setName(resultSet.getString("name"));
        account.setMoney(resultSet.getFloat("money"));
        return account;
    }
}
/*
定义Account的封装策略  //有点问题  实际开发中查询一个还是查list,然后get(0)
 */
class AccountExtractor implements ResultSetExtractor<Account>{

    public Account extractData(ResultSet resultSet) throws SQLException,
DataAccessException {
        Account account = new Account();
        account.setId(resultSet.getInt("id"));
        account.setName(resultSet.getString("name"));
        account.setMoney(resultSet.getFloat("money"));
        return account;
    }
}
```

五、事务

环境搭建

> - 本节使用JdbcTemplate
> - 还是使用上面的账户类及其转账案例

5.1 基于XML

```
<?xml version="1.0" encoding="UTF-8"?>
```

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">
    <!-- 配JdbcTemplate -->
    <bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"></property>
    </bean>
    <!-- 配spring内置数据源 -->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <!-- 注入连接数据库的必备信息 -->
        <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
        <property name="url" value="jdbc:mysql://localhost:3306/db_spring"></property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
    </bean>
    <!-- 配dao层 -->
    <bean id="accountDao" class="com.aiqi.dao.impl.AccountDaoImpl">
        <property name="jt" ref="jt"></property>
    </bean>
    <!--配service层-->
    <bean id="accountService" class="com.aiqi.service.impl.AccountServiceImpl">
        <property name="accountDao" ref="accountDao"></property>
    </bean>

    <!-- 基于xml的声明式事务控制 -->
    <!--
        导入xml约束(tx的名称空间和约束,同时事务也需要aop的支持,所以这里也导入了aop)
        1.配置事务管理器,注入dataSource
        2.配置事务的通知
            通过tx:advice标签配置
            属性
            id:事务通知的唯一标识
            transaction-manager:给通知设置一个事务管理器
        3.配置AOP的通用切入点表达式
        4.建立事务通知和切入点表达式的联系
        5.配置事务的属性
            在<tx:advice>标签内通过<tx:attributes>配置

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

    </bean>
```

```
    <!--配置事务的通知-->
    <tx:advice id="txAdvice" transaction-manager="txManager">
        <!--配置事务的属性-->
        <tx:attributes>
            <tx:method name="*" propagation="REQUIRED" read-only="false"/>
            <tx:method name="find*" propagation="SUPPORTS" read-only="true"></tx:method>
        </tx:attributes>
    </tx:advice>
    <!--配置AOP-->
    <aop:config>
        <!--配置AOP切入点表达式-->
        <aop:pointcut id="pt1" expression="execution(* com.aiqi.service.impl.*.*(..))"/>
        <!--建立AOP切入点表达式和事务通知的关系-->
        <aop:advisor advice-ref="txAdvice" pointcut-ref="pt1"></aop:advisor>
    </aop:config>
</beans>
```

知识点:**事务的属性**

> isolation 用于指定事务的隔离级别,默认值default,表示使用数据库的默认隔离级别

> propagation 用于指定事务的传播行为,默认值REQUIRED,表示一定会有事务,增删改的选择;查询可以选择SUPPORTS

> read-only 用于指定事务是否只读,默认false,表示可以读写;查询方法才能设置为true,只读.

> timeout 用于指定事务的超时时间,默认值-1,表示永不超时;如果指定了数值,以秒(s)为单位

> rollback-for 用于指定一个异常,当异常产生时,事务回滚,产生其他异常时,事务不回滚;不指定时,默认所有异常都回滚

> no-rollback-for 用于指定一个异常,当异常产生时,事务不回滚,产生其他异常时,事务回滚;不指定时,默认所有异常都回滚

5.2 基于注解

```
<!-- 基于注解的声明式事务控制 -->
    <!--
        1.配置事务管理器,注入dataSource
        2.开启spring对注解事务的支持
        3.在需要事务支持的地方使用@Transactional
    -->

    <!--配置事务管理器-->
    <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"></property>
    </bean>
    <!--开启注解事务的支持-->
    <tx:annotation-driven transaction-manager="txManager"></tx:annotation-driven>
```

```
@Transactional(propagation = Propagation.REQUIRED,readOnly = false)

public void transfer(Integer sourceId, Integer targetId, Float money) {
```

```
        //1.根据id查询转出账户
        Account sourceAccount = accountDao.findById(sourceId);
        //2.根据id查询转入账户
        Account targetAccount = accountDao.findById(targetId);
        //3.转出账户减钱
        sourceAccount.setMoney(sourceAccount.getMoney() - money);
        //4.转入账户加钱
        targetAccount.setMoney(targetAccount.getMoney() + money);
        //5.更新转出账户
        accountDao.update(sourceAccount);
        //int i = 1/0;
        //6.更新转入账户
        accountDao.update(targetAccount);
}
```

上面还小小的依赖了下xml配置,完全不用xml的话,还需要编写配置类

> SpringConfig.java

```
package com.aiqi.config;

import org.springframework.context.annotation.*;
import org.springframework.transaction.annotation.EnableTransactionManagement;

/**
 * @description: Spring的主配置类, 替代xml配置
 * @author: aiqi
 * @create: 2020-05-05 13:57
 **/

/*
当我们用注解开发,完全不用xml配置时,需要写配置类
 */

@Configuration  //说明该类是一个配置类
@ComponentScan("com.aiqi")  //配置IOC注解要扫描的包
@Import({JdbcConfig.class,TxConfig.class})  //引入其他配置类
@PropertySource("classpath:jdbc.properties")  //引入properties文件
@EnableAspectJAutoProxy  //配置spring开启注解AOP的支持
@EnableTransactionManagement  //配置spring开启注解事务的支持
public class SpringConfig {}
```

> JdbcConfig.java

```
package com.aiqi.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
```

```java
import javax.sql.DataSource;

/**
 * @description: 与数据库连接相关的配置类
 * @author: aiqi
 * @create: 2020-05-05 17:11
 **/

@Configuration
public class JdbcConfig {

    @Value("${jdbc.driver}")
    private String driver;

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    /*
     * @Description: 用于创建JdbcTemplate对象,并把它放入springIOC容器中
     * @Param: [dataSource]
     * @return: org.springframework.jdbc.core.JdbcTemplate
     **/
    @Bean("jt")
    @Scope("prototype")   //设置为多例,线程安全
    public JdbcTemplate getJdbcTemplate(DataSource dataSource){
        return new JdbcTemplate(dataSource);
    }

    /*
    * @Description: 创建spring内置数据源对象,并传入Spring的IOC容器
    * @Param: []
    * @return: org.springframework.jdbc.datasource.DriverManagerDataSource
    **/
    @Bean("dataSource")
    public DriverManagerDataSource getDataSource(){
        try {
            DriverManagerDataSource dmds = new DriverManagerDataSource();
            dmds.setDriverClassName(driver);
            dmds.setUrl(url);
            dmds.setUsername(username);
            dmds.setPassword(password);
            return dmds;
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }

}
```

> TxConfig.java

```
package com.aiqi.config;

import org.springframework.context.annotation.Bean;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.PlatformTransactionManager;

import javax.sql.DataSource;

/**
 * @description: 有关事务的配置类
 * @author: aiqi
 * @create: 2020-05-10 21:48
 **/
public class TxConfig {
    /*
    * @Description: 用于创建数据源事务管理器,并传入springIOC容器
    * @Param: [dataSource]
    * @return: org.springframework.transaction.PlatformTransactionManager
    **/
    @Bean("txManager")
    public PlatformTransactionManager getTransactionManager(DataSource dataSource){
        return new DataSourceTransactionManager(dataSource);
    }
}
```

5.3 编程式事务控制

> **再去了解**

```
    by aiqi
```