Spring _01

学习目标

- [ ] 能够描述spring框架
- [ ] 能够编写spring的IOC的入门案例
- [ ] 能够说出spring的bean标签的配置
- [ ] 能够理解Bean的属性注入方法
- [ ] 能够理解复杂类型的属性注入
- [ ] 能够实现spring框架整合JUnit

第一章-Spring概述

知识点-Spring介绍

1.目标

- [ ] 能够描述spring框架

2.路径

1. 什么是Spring
2. Spring 的发展历程
3. Spring的优点
4. Spring的体系结构

3.讲解

3.1什么是Spring

Spring 是一个开源框架，是为了解决企业应用程序开发复杂性而创建的(解耦)。

框架的主要优势之一就是其分层架构，分层架构允许您选择使用哪一个组件，同时为 J2EE 应用程序开发提供集成的框架。

简单来说，Spring是一个分层的JavaSE/EE full-stack(一站式) 轻量级开源框架。

一站式:Spring提供了三层解决方案.

3.2Spring 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 spring 5.0 通用版（GA）

3.3 Spring的优点

1.方便解耦，简化开发(基础重要功能)

通过Spring提供的IoC容器，我们可以将对象之间的依赖关系交由Spring进行控制，避免硬编码所造成的过度程序耦合。有了Spring，用户不必再为单实例模式类、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。

2. AOP编程的支持(亮点)

通过Spring提供的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。

3.声明式事务的支持 (简化事务)

在Spring中, 我们可以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地进行事务的管理，提高开发效率和质量

4.方便程序的测试 (集成Junit测试框架)

可以用非容器依赖的编程方式进行几乎所有的测试工作，在Spring里，测试不再是昂贵的操作，而是随手可做的事情。例如：Spring对Junit4支持，可以通过注解方便的测试Spring程序。

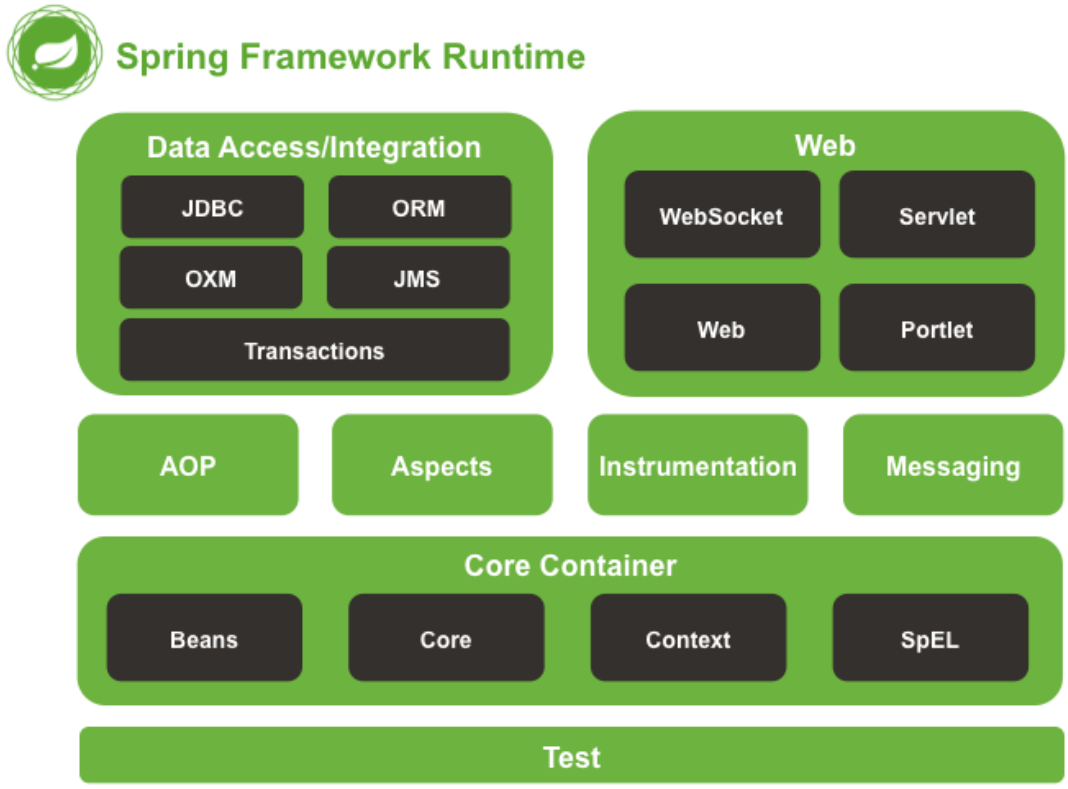
5.方便集成各种优秀框架 (亮点、优势)

Spring不排斥各种优秀的开源框架，相反，Spring可以降低各种框架的使用难度，Spring提供了对各种优秀框架（如Struts,Hibernate、Hessian、Quartz）等的直接支持。

6.降低Java EE API的使用难度(了解)

Spring对很多难用的Java EE API（如JDBC，JavaMail，远程调用等）提供了一个薄薄的封装层，通过Spring的简易封装，这些Java EE API的使用难度大为降低

3.4Spring的体系结构



4.小结

1. Spring是JavaSE/JavaEE 一系列的框架. 提供了三层开发里面的技术解决方案, 是一站式框架
2. 我们这三天学的是Spring的核心部分
    - IOC
    - AOP

- 声明式事务

第二章-IOC

知识点-工厂模式解耦

1.目标

- [ ] 能够使用工厂模式进行解耦

2.路径

1. 程序耦合的概述
2. 使用工厂模式解耦

3.讲解

3.1程序的耦合

3.1.1 程序耦合的概述

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、调用模块的方式以及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：
（1） 内容耦合。当一个模块直接修改或操作另一个模块的数据时，或一个模块不通过正常入口而转入另一个模块时，这样的耦合被称为内容耦合。内容耦合是最高程度的耦合，应该避免使用之。
（2） 公共耦合。两个或两个以上的模块共同引用一个全局数据项，这种耦合被称为公共耦合。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。
（3） 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，则称之为外部耦合。
（4） 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，这种耦合被称为控制耦合。
（5） 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。
（6） 数据耦合。模块之间通过参数来传递数据，那么被称为数据耦合。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。
（7） 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。

3.1.2在代码中体现

早期我们的 JDBC 操作(面向接口的编程)，注册驱动时，我们为什么不使用 DriverManager 的 register 方法，而是采
用 Class.forName 的方式?

原因就是： 我们的类依赖了数据库的具体驱动类（MySQL） ，如果这时候更换了数据库（比如 Oracle） ，
需要修改源码来重新数据库驱动。这显然不是我们想要的。

```
/**
* 程序的耦合
* 耦合：程序间的依赖关系
```

```
 * 包括：
 *         类之间的依赖
 *         方法间的依赖
 * 解耦：
 *         降低程序间的依赖关系
 * 实际开发中：
 *         应该做到：编译期不依赖，运行时才依赖。
 * 解耦的思路：
 *         第一步：使用反射来创建对象，而避免使用 new 关键字。
 *         第二步：通过读取配置文件来获取要创建的对象全限定类名
 */
public class JdbcDemo1 {
    public static void main(String[] args) throws Exception{
        //1.注册驱动
        //DriverManager.registerDriver(new Driver());//第一种注册驱动的方式
        //第一种注册驱动的方式，缺点是耦合性强

        Class.forName("com.mysql.jdbc.Driver");//这是第二种注册驱动的方式
        //第二种注册驱动的方式，缺点是"字符串硬编码"，可以使用配置文件解决

        //2.获取连接
        //3.获取操作数据库的预处理对象
        //4.执行 SQL，得到结果集
        //5.遍历结果集
        //6.释放资源
    }
}
```

第一种耦合:一个类直接依赖另外一个类

第二种耦合:将字符串直接写死在Java代码中(字符串的硬编码)

3.1.3解决程序耦合的思路

产生类与类之间的耦合的根本原因是:在一个类中new了另外一个类的对象

解决耦合的思路:不在类中创建另外一个类的对象，但是我们还需要另一个类的对象；由别人(Spring)把那个类的对象创建好之后给我用就可以了

当是我们讲解 jdbc 时，是通过反射来注册驱动的，代码如下：

`Class.forName("com.mysql.jdbc.Driver");//此处只是一个字符串`

此时的好处是，我们的类中不再依赖具体的驱动类，此时就算删除 mysql 的驱动 jar 包，依然可以编译（运行就不要想了没有驱动不可能运行成功的）。

同时，也产生了一个新的问题， mysql 驱动的全限定类名字符串是在 java 类中写死的，一旦要改还是要修改源码。解决这个问题也很简单，使用配置文件配置

3.2.自定义IOC(工厂模式解耦)

1. 原始方式
    - 方式: 创建类, 直接根据类new对象
    - 优点: 好写, 简单
    - 缺点: 耦合度太高, 不好维护
2. 接口方式
    - 方式: 定义接口, 创建实现类. 接口=子类的对象
    - 优点: 耦合度相对原始方式 减低了一点

    - 缺点: 多写了接口, 还是需要改源码 不好维护
3. 自定义IOC
    - 方式: 使用对象的话, 不直接new()了,直接从工厂里面取; 不需要改变源码

3.2.1原始方式

代码略

3.2.2接口方式

- AccountDao .java

```
public interface AccountDao {
    /**
     * 保存账户
     */
    void save();
}
```

- AccountDaoImpl.java

```
public class AccountDaoImpl implements AccountDao {
    /**
     * 保存账户
     */
    @Override
    public void save() {
        System.out.println("AccountDaoImpl... save()");
    }
}
```

- AccountService.java

```
public interface AccountService {
    /**
     * 保存账户
     */
    void save();
}
```

- AccountServiceImpl.java

```
public class AccountServiceImpl implements AccountService {
    /**
     * 保存账户
     */
    @Override
    public void save() {
        System.out.println("AccountServiceImpl... save()");
        AccountDao accountDao = new AccountDaoImpl();
        accountDao.save();
    }
}
```

- Client.java

```
public class Client {
    public static void main(String[] args) {
        AccountService accountService = new AccountServiceImpl();
        accountService.save();
    }

}
```

3.2.3自定义IOC(使用工厂模式解耦)

3.2.2.1工厂模式解耦思路

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

```
//1. 解析beans.xml文件，根据id获取要创建对象的类的全限定名
        SAXReader reader = new SAXReader();
        //将beans.xml文件转换成字节输入流
        InputStream is =
BeanFactory.class.getClassLoader().getResourceAsStream("beans.xml");

        //2. 读取配置文件，得到document对象
        try {
            Document document = reader.read(is);
            //3. 查找所有的bean标签
            List<Element> list = document.selectNodes("//bean");
            if (list != null && list.size() > 0) {
                //遍历出每一个bean标签
                for (Element element : list) {
                    //获取每一个bean标签的id属性和class属性
                    String attrId = element.attributeValue("id");
                    String attrClass = element.attributeValue("class");

                    //判断，传入的id是否和bean标签的id相同，如果相同，则创建对象
                    if (attrId.equals(id)) {
                        //找到了对应的bean标签,那么attrClass就是要创建对象的类的全限定名
                        return Class.forName(attrClass).newInstance();
                    }
                }
            }
            throw new RuntimeException("无法创建你传入的id的对象");
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException("创建对象出现异常");
        }
```

3.2.2.2实现

- 添加坐标依赖

```
  <dependencies>
    <!-- 解析 xml 的 dom4j -->
```

```
    <dependency>
      <groupId>dom4j</groupId>
      <artifactId>dom4j</artifactId>
      <version>1.6.1</version>
    </dependency>
    <!-- dom4j 的依赖包 jaxen -->
    <dependency>
      <groupId>jaxen</groupId>
      <artifactId>jaxen</artifactId>
      <version>1.1-beta-8</version>
    </dependency>
    <!--单元测试-->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.10</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

- AccountDao .java

```
public interface AccountDao {
    /**
     * 保存账户
     */
    void save();
}
```

- AccountDaoImpl.java

```
public class AccountDaoImpl implements AccountDao {
    /**
     * 保存账户
     */
    @Override
    public void save() {
        System.out.println("AccountDaoImpl... save()");
    }
}
```

- AccountService.java

```
public interface AccountService {
    /**
     * 保存账户
     */
    void save();
}
```

- AccountServiceImpl.java

```
public class AccountServiceImpl implements AccountService {

    private AccountDao accountDao = (AccountDao)
BeanFactory.getBean("accountDao");
    /**
     * 保存账户
     */
    @Override
    public void save() {
        System.out.println("AccountServiceImpl---saveAccount()");
        accountDao.save();
    }
}
```

- Client.java

```
public class Client {

    public static void main(String[] args) {
        AccountService accountService = (AccountService)
BeanFactory.getBean("accountService");
        accountService.save();
    }

}
```

- BeanFactory.java

  下面的通过 BeanFactory 中 getBean 方法获取对象就解决了我们代码中对具体实现类的依赖。

```
/**
 * @Description: 对象工厂
 * @Author: yp
 */
public class BeanFactory {
    public static Map<String, Object> beanMap = new HashMap<String, Object>();

    //程序初始化的时候, 把xml解析好 反射创建好 存到容器【map key是id, value是对应的对象】
里面
    static {
        InputStream is = null;
        try {
            //1.创建SAXReader对象
            SAXReader saxReader = new SAXReader();
            //2.读取xml 获得document对象
            is =
BeanFactoryOri.class.getClassLoader().getResourceAsStream("applicationContext.xm
l");
            Document document = saxReader.read(is);
            //3.获得所有的bean标签
            List<Element> beanEles =  document.selectNodes("//bean");

            //4.遍历所有的bean标签
            for (Element beanEle : beanEles) {
                //5.获得bean标签的id和class属性值
                String id = beanEle.attributeValue("id");
```

```
                String className = beanEle.attributeValue("class");

                //6.根据class属性值反射创建对象, 把id作为key,把反射创建的对象作为value存到beanMap
                Object obj  = Class.forName(className).newInstance();
                beanMap.put(id,obj);
            }

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (is != null) {
                try {
                    is.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

    }

    /**
     * 根据id获得对应的对象(Bean)
     *
     * @param id
     * @return 初级版本的问题:
     * 1.每次getBean都需要读取xml
     * 2.临时的时候才反射创建
     * <p>
     * 终极版本里面解决
     * 1.xml读取一次 (定义在static{})
     * 2.程序初始化的时候, 解析好 反射创建好 存到容器【map key是id, value是对应的对象】里面. getBean()的时候直接从容器里面取
     */
    public static Object getBean(String id) {  //eg: userDao
        return beanMap.get(id);
    }
}
```

- applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <!--id就是接口的名字, class实现类的全限定名  -->
    <bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"></bean>
    <bean id="userService" class="com.itheima.service.impl.UserServiceImpl">
</bean>
</beans>
```

4.小结

1. 程序初始化的时候, 解析xml, 全部解析好 存到容器【Map】里面
2. 把bean标签的id作为map的key, bean标签的class属性值反射创建对象之后作为map的value
3. 要使用对象, 不再直接new了, 而是从容器里面直接取

知识点-IOC概念

1.目标

- [ ] 能够理解IOC概念

2.路径

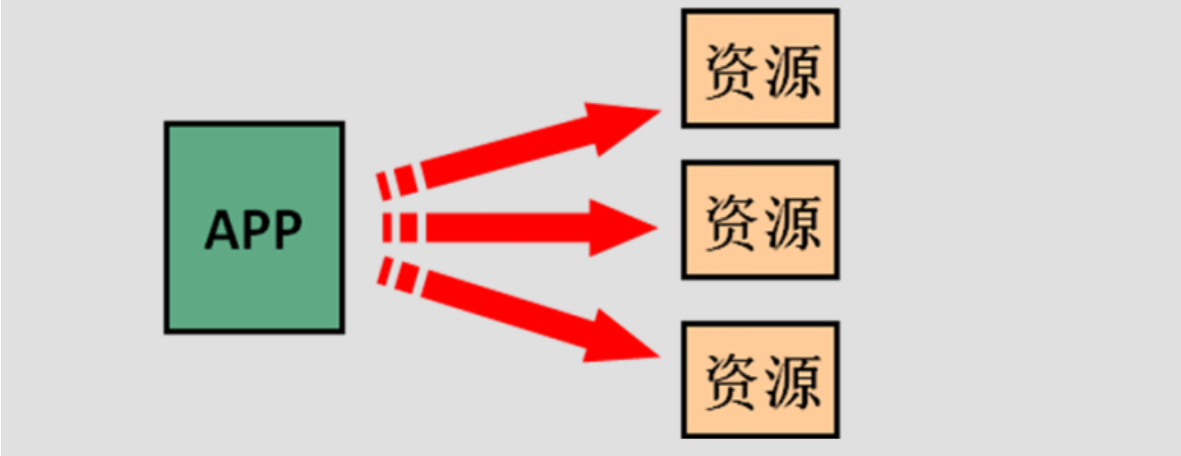
1. 分析和IOC的引入
2. IOC概述

3.讲解

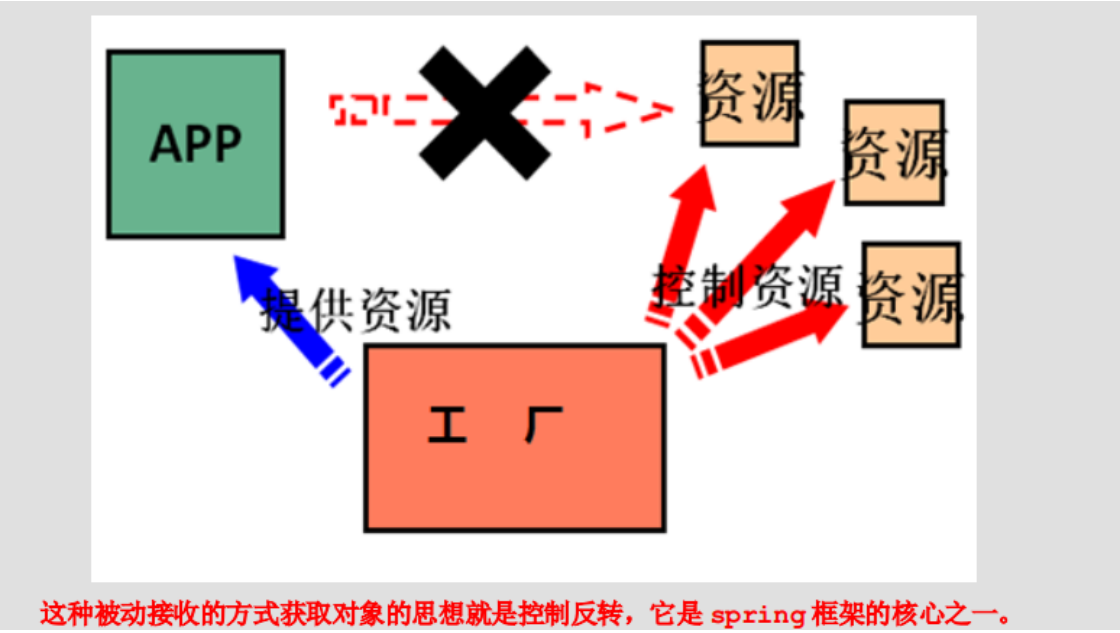
3.1分析和IOC的引入

上一小节我们通过使用工厂模式，实现了表现层——业务层以及业务层——持久层的解耦。
它的核心思想就是：
1、通过读取配置文件反射创建对象。
2、把创建出来的对象都存起来，当我们下次使用时可以直接从存储的位置获取。
这里面要解释两个问题：
第一个：存哪去?
分析：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。
所以我们的答案就是: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。
第二个： 什么是工厂?
工厂就是负责给我们从容器中获取指定对象的类。这时候我们获取对象的方式发生了改变。
原来：我们在获取对象时，都是采用 new 的方式。 是主动的。

- 老方式



- 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。



3.2IOC概述

IOC(inversion of control)的中文解释是"==控制反转"，对象的使用者不是创建者. 作用是将对象的创建 反转给spring框架来创建和管理。==

控制反转怎么去理解呢。 其实它反转的是什么呢，是对象的创建工作。 举个例子:平常我们在servlet或者service里面创建对象，都是使用new 的方式来直接创建对象，现在有了spring之后，我们就再也不new对象了，而是把对象创建的工作交给spring容器去维护。我们只需要问spring容器要对象即可

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)。

4.小结

1. IOC: 控制反转. 对象的使用者不是创建者. 把对象的创建和管理, 交给Spring框架. 要使用的时候, 直接从Spring工厂(容器)里面取
2. 作用: 解耦.

案例-Spring的IOC快速入门

1.需求

- [ ] 能够编写spring的IOC的入门案例

2.分析

本章我们使用的案例是， 账户的业务层和持久层的依赖关系解决。在开始 spring 的配置之前，我们要先准备一下环境。由于我们是使用 spring 解决依赖关系，并不是真正的要做增删改查操作，所以此时我们没必要写实体类。并且我们在此处使用的是 java 工程，不是 java web 工程。

3.实现

3.0步骤

1. 创建Maven工程, 添加坐标
2. 准备好接口和实现类
3. 创建spring的配置文件 (applicationContext.xml), 配置bean标签

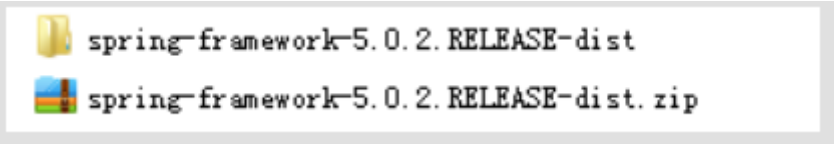
4. 创建工厂对象 获得bean 调用

3.1准备工作

3.1.1准备 spring 的开发包

官网： http://spring.io/
下载地址：http://repo.springsource.org/libs-release-local/org/springframework/spring
解压:(Spring 目录结构:)

- docs :API 和开发规范.
- libs :jar 包和源码.
- schema :约束.



我们上课使用的版本是 spring5.0.2。
特别说明：

spring5 版本是用 jdk8 编写的，所以要求我们的 jdk 版本是 8 及以上。

3.1.2接口和实现类

- AccountService.java

```
public interface AccountService {

    /**
     * 保存账户
     */
    void save();
}
```

- AccountServiceImpl.java

```
public class AccountServiceImpl implements AccountService {

    /**
     * 保存账户
     */
    @Override
    public void save() {
        System.out.println("AccountServiceImpl---saveAccount()");
    }
}
```

- Client.java

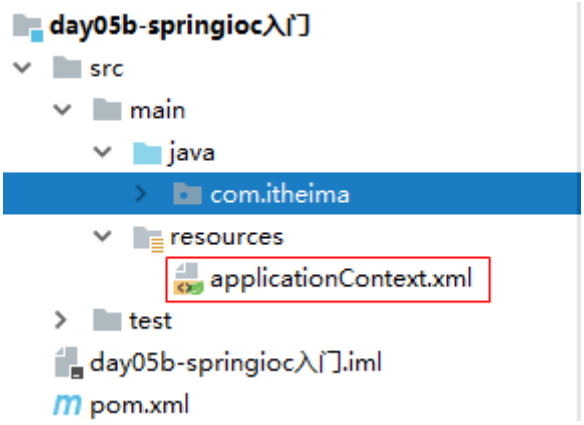
```
public class Client {
    public static void main(String[] args) {
        AccountService accountService = new AccountServiceImpl();//TODO 耦合待解决
        accountService.save();
    }
}
```

3.2代码实现

- 创建 maven 工程并导入坐标
- 坐标

```
<dependencies>
    <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.2.RELEASE</version>
    </dependency>
</dependencies>
```

- 在类的根路径下创建spring的配置文件
- 配置文件为任意名称的 xml 文件（不能是中文）,一般习惯名称为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">

    <!--注册Service-->
    <bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl"></bean>
</beans>
```

- 编写Java代码测试

```
public class DbTest {
    @Test
    public void fun01() throws Exception {
        //1. 创建工厂
        ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
        //2. 根据id获得对象
        AccountService accountService = (AccountService)
applicationContext.getBean("accountService");
        accountService.save();

    }
}
```

4.小结

4.1步骤

1. 创建Maven工程, 添加坐标
2. 准备接口和实现类
3. 创建applicationContext.xml 配置bean
4. 创建工厂, 从工厂获得对象

4.2注意事项

我们当前使用的Spring版本是5.0.2 jdk要求>=8的

####

知识点-Spring 基于 XML 的 IOC 细节

1.目标

- [ ] 能够说出spring的bean标签的配置

2.路径

1. 配置文件详解(Bean标签)
2. bean的作用范围和生命周期

3.讲解

3.1配置文件详解(Bean标签)

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

    <!--
        bean标签: 注册bean, 把对象交给Spring管理
        1.【掌握】属性:
            id/name属性: 随便写,作为bean的唯一标识,不要重复(建议写接口的名字,首字母小写)
```

```
            class属性:  类的全限定名
            scope属性:  singleton 单例,  不管获得多少次 都是同一个, 创建出来存到Spring容
器里面  【默认】
                        prototype 多例,  每获得一次 就创建一个新的对象,创建出来不会存到
Spring容器里面
       2. 【了解】的属性
            init-method属性: 指定初始化方法,写方法名
            destroy-method属性: 指定销毁的方法, 象征着当前对象从Spring容器里面移除了 写方
法名
    -->
    <bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl" scope="prototype"
        init-method="initMethod" destroy-method="destoryMethod"
    ></bean>

</beans>
```

- id/name属性

  用于标识bean， 其实id 和 name都必须具备唯一标识， 两种用哪一种都可以。但是一定要唯一、 一般开发中使用id来声明.

- class属性: 用来配置要实现化类的全限定名
- scope属性: 用来描述bean的作用范围

  singleton: 默认值，单例模式。spring创建bean对象时会以单例方式创建。(默认)

  prototype: 多例模式。spring创建bean对象时会以多例模式创建。

  request: 针对Web应用。spring创建对象时，会将此对象存储到request作用域。

  session: 针对Web应用。spring创建对象时，会将此对象存储到session作用域。

- init-method属性：spring为bean初始化提供的回调方法
- destroy-method属性：spring为bean销毁时提供的回调方法. 销毁方法针对的都是单例bean，如果想销毁bean , 可以关闭工厂

3.2bean的作用范围和生命周期

- 单例对象： scope="singleton"

  一个应用只有一个对象的实例。它的作用范围就是整个引用。
  生命周期:
  对象出生：当应用加载，创建容器时，对象就被创建了。
  对象活着：只要容器在，对象一直活着。
  对象死亡：当应用卸载，销毁容器时，对象就被销毁了。

- 多例对象： scope="prototype"

  每次访问对象时，都会重新创建对象实例。
  生命周期:
  对象出生：当使用对象时，创建新的对象实例。
  对象活着：只要对象在使用中，就一直活着。
  对象死亡：当对象长时间不用时，被 java 的垃圾回收器回收了.

4.小结

1. 配置结构

```
<bean id/name="" class="" scope="" init-method="" destory-method=""></bean>
```

2. scope取值
    - singleton:单例【默认】, 使用的都是同一个对象(同一个id获得的). 单例的bean存到Spring容器里面
    - prototype:多例, 使用的不是同一个对象(同一个id获得的), 使用一个 创建一个. 多例的bean不会存到Spring容器里面

知识点-Spring工厂详解

1.目标

- [ ] 掌握Spring常见的工厂类

2.路径

1. spring 中工厂的类结构图
2. BeanFactory 和 ApplicationContext 的区别

3.讲解

3.1spring 中工厂的类结构图

- ClassPathXmlApplicationContext：它是从类的根路径下加载配置文件
- 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。
- 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。

3.2BeanFactory 和 ApplicationContext 的区别(了解)

- ApplicationContext 是现在使用的工厂

```
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
```

- XmlBeanFactory是老版本使用的工厂,目前已经被废弃【了解】

```
BeanFactory beanFactory = new XmlBeanFactory(new
ClassPathResource("applicationContext.xml"));
```

两者的区别:

ApplicationContext加载方式是框架启动时就开始创建所有单例的bean,存到了容器里面

BeanFactory加载方式是用到bean时再加载(目前已经被废弃)

4.小结

1. 工厂类继承关系

- BeanFactory
  - ==ApplicationContext==
    - FileSystemXmlApplicationContext　xml方式 它是从磁盘路径上加载配置文件
    - ==ClassPathXmlApplicationContext　xml方式 它是从类的根路径下加载配置文件==
    - ==AnnotationConfigApplicationContext　注解方式的==

2. 新旧工厂
   - 新工厂: 在工厂初始化的话的时候, 就会创建配置的所有的单例bean,存到Spring容器里面
   - 旧工厂(已经废弃了): 等用到对象的时候, 再创建

知识点-实例化Bean的三种方式【了解】

1.目标

- [ ] 了解实例化 Bean 的三种方式

2.路径

1. 无参构造方法方式
2. 静态工厂方式
3. 实例工厂实例化的方法

3.讲解

3.1方式一:无参构造方法方式 （只要掌握这种）

需要实例化的类，提供无参构造方法

配置代码

```
 <bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">
</bean>
```

3.2方式二:静态工厂方式

需要额外提供工厂类， 工厂方法是静态方法

```
public class StaticFactory {
    public static Object getBean(){
        return new AccountServiceImpl02();
    }
}
```

配置代码

```
<!--方式二:静态工厂方式  -->
<bean  id="accountService02" class="com.itheima.utils.StaticFactory" factory-
method="getBean"/>
```

3.3方式三:实例工厂实例化的方法

创建实例化工厂类

```
public class InstanceFactory {
    public Object getBean(){
        return new AccountServiceImpl03();
    }
}
```

配置

```
<!-- 方式三: 实例化工厂 -->
<!--注册工厂  -->
<bean id="factory" class="com.itheima.utils.InstanceFactory"></bean>
<!--引用工厂  -->
<bean  id="accountService03" factory-bean="factory" factory-method="getBean"/>
```

4.小结

1. 一般情况下不会使用第二和三种, 主要使用第一种

第三章-依赖注入

知识点-依赖注入概念

1.目标

- [ ] 掌握什么是依赖注入

2.讲解

依赖注入全称是 dependency Injection 翻译过来是依赖注入.其实就是如果我们托管的某一个类中存在属性，需要spring在创建该类实例的时候，顺便给这个对象里面的属性进行赋值。 这就是依赖注入。

现在, Bean的创建交给Spring了, 需要在xml里面进行注册

我们交给Spring创建的Bean里面可能有一些属性(字段), Spring帮我创建的同时也把Bean的一些属性(字段)给赋值, 这个赋值就是注入.

3.小结

1. 注册: 把bean的创建交给Spring
2. 依赖注入: bean创建的同时, bean里面可能有一些字段需要赋值, 这个赋值交给Spring, 这个过程就是依赖注入

知识点-依赖注入实现

1.目标

- [ ] 能够理解Bean的属性注入方法
- [ ] 能够理解复杂类型的属性注入

2.路径

1. 构造方法方式注入
2. set方法方式的注入
3. P名称空间注入

3.讲解

3.1构造方法方式注入【掌握】

- Java代码

```
public class AccountServiceImpl implements AccountService {
    private String name;
    public AccountServiceImpl(String name) {
        this.name = name;
    }
    @Override
    public void save() {
        System.out.println("AccountServiceImpl... save()"+name);

    }
}
```

- 配置文件

```
<!--注册AccountService-->
 <bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl">
    <constructor-arg name="name" value="张三"></constructor-arg>
</bean>
```

3.2set方法方式的注入【重点】

3.2.1注入简单类型

- java代码

```
public class AccountServiceImpl implements AccountService {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public void save() {
        System.out.println("AccountServiceImpl... save()"+name);

    }
}
```

- 配置文件

```
 <bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl">
       <property name="name" value="李四"></property>
</bean>
```

3.2.2注入数组类型(了解一下)

- java代码

```
public class AccountServiceImpl implements AccountService {
    private String[] hobbys;

    public void setHobbys(String[] hobbys) {
        this.hobbys = hobbys;
    }

    @Override
    public void save() {
        System.out.println("AccountServiceImpl... save()"+
Arrays.toString(hobbys));

    }
}
```

- 配置文件

```
    <bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl">
        <property name="hobbys">
            <array>
                <value>篮球</value>
                <value>足球</value>
                <value>乒乓球</value>
                <value>排球</value>
            </array>
        </property>
    </bean>
```

3.2.3注入Map类型(了解)

- Java代码

```
public class AccountServiceImpl implements AccountService {
    private Map<String,String> map;

    public void setMap(Map<String, String> map) {
        this.map = map;
    }

    @Override
    public void save() {
        Set<Map.Entry<String, String>> set = map.entrySet();
        for (Map.Entry<String, String> entry : set) {
            System.out.println(entry.getKey()+":"+entry.getValue());
        }
        System.out.println("AccountServiceImpl... save()");
```

```
    }
}
```

- 配置文件

```
<bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl">
     <property name="map">
         <map>
              <entry key="akey" value="aaa"/>
              <entry key="bkey" value="bbb"/>
              <entry key="ckey" value="ccc"/>
         </map>
     </property>
</bean>
或者

<bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl">
    <property name="map">
        <props>
            <prop key="akey">aaa</prop>
            <prop key="bkey">bbb</prop>
            <prop key="ckey">ccc</prop>
        </props>
    </property>
</bean>
```

3.2.4注入Java对象类型(最重要的)

- Java代码

```
public class AccountServiceImpl implements AccountService {
    private AccountDao accountDao;

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }
    @Override
    public void save() {

        System.out.println("AccountServiceImpl... save()");
        accountDao.save();

    }
}
```

- 配置文件

```
<!--注册AccountService-->
<bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl">
    <property name="accountDao" ref="accountDao"></property>
</bean>
<!--注册accountDao-->
<bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl">
</bean>
```

3.3 P名称空间注入[了解] 其实底层也是调用set方法，只是写法有点不同

3.3.1 p名称空间的方式

- 提供属性的set方法
- 在applicationContext.xml引入p命令空间
  `xmlns:p="http://www.springframework.org/schema/p"`

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

- 使用

```
<bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl"
p:name="张三">
</bean>
```

4.小结

1. 构造方法方式
   - 定义变量 ,提供构造方法
   - 在bean标签里面配置子标签constructor-arg

```
<bean id="" class="">
    <constructor-arg name="" value=""></constructor-arg>
    <constructor-arg name="" ref=""></constructor-arg>
</bean>
```

2. Set方法方式
   - 2.1 步骤
     - 定义变量 ,提供Set方法
     - bean标签里面配置子标签property
   - 2.2 简单类型(基本类型, String)

```
<bean id="" class="">
    <property name="" value=""></property>
</bean>
```

- 2.3 数组类型

```
<bean id="" class="">
    <property name="">
        <array>
            <value></value>
        </array>
    </property>
</bean>
```

- 2.3 Map类型

```
<bean id="" class="">
    <property name="">
        <map>
            <entry key="" value=""></entry>
        </map>
    </property>
</bean>
```

- 2.4 对象类型

```
<bean id="" class="">
    <property name="" ref="">
    </property>
</bean>
```

第四章-IOC练习

使用Spring的IoC的实现账户的CRUD

1.目标

- [ ] 够实现spring基于XML的IoC案例

2.分析

1. 环境搭建
2. 创建applicationContext.xml
    - new的地方, 就注册
    - 需要用到的字段(对象), 就注入
3. 启动Spring工厂

3.实现

3.1环境搭建

1. 创建Maven工程,导入坐标

```
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>
  </dependencies>
```

实体

```
package com.itheima.pojo;

/**
 * 包名:com.itheima.pojo
 *
 * @author Leevi
 * 日期2020-06-26  15:02
 */
public class Account {
    private Integer id;
    private String name;
    private Double money;

    public Account() {
    }

    public Account(Integer id, String name, Double money) {
        this.id = id;
        this.name = name;
        this.money = money;
    }

    @Override
    public String toString() {
        return "Account{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", money=" + money +
                '}';
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getMoney() {
        return money;
    }

    public void setMoney(Double money) {
        this.money = money;
    }
}
```

3. 编写持久层代码

AccountDao.java

```
public interface AccountDao {

    void save(Account account) throws SQLException;

    void update(Account account) throws SQLException;

    void delete(Integer id) throws SQLException;

    Account findById(Integer id) throws SQLException;

    List<Account> findAll() throws SQLException;
}
```

AccountDaoImpl.java

```
public class AccountDaoImpl implements AccountDao {
     @Override
    public void save(Account account) throws SQLException {
        System.out.println("保存用户信息:"+account);
    }

    @Override
    public void update(Account account) throws SQLException {
        System.out.println("更新用户信息:"+account);
    }

    @Override
    public void delete(Integer id) throws SQLException {
        System.out.println("删除用户信息:"+id);
    }

    @Override
    public Account findById(Integer id) throws SQLException {
        System.out.println("查询用户信息");
        return new Account(1,"张三",1000.0);
```

```
    }

    @Override
    public List<Account> findAll() throws SQLException {
        //3. 执行sql语句
        System.out.println("查询左右用户信息");
        List<Account> accountList = new ArrayList<>();
        accountList.add(new Account(1,"张三",1000.0));
        accountList.add(new Account(2,"李四",2000.0));
        return accountList;
    }
}
```

4. 编写业务层代码

AccountService.java

```
public interface AccountService {
    void save(Account account) throws Exception;

    void update(Account account) throws Exception;

    void delete(Integer id) throws Exception;

    Account findById(Integer id) throws Exception;

    List<Account> findAll() throws Exception;

}
```

AccountServiceImpl.java

```
public class AccountServiceImpl implements AccountService {

    private AccountDao accountDao;

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    @Override
    public void save(Account account) throws Exception {
        accountDao.save(account);

    }

    @Override
    public void update(Account account) throws Exception {
        accountDao.update(account);

    }

    @Override
    public void delete(Integer id) throws Exception {
        accountDao.delete(id);
    }
```

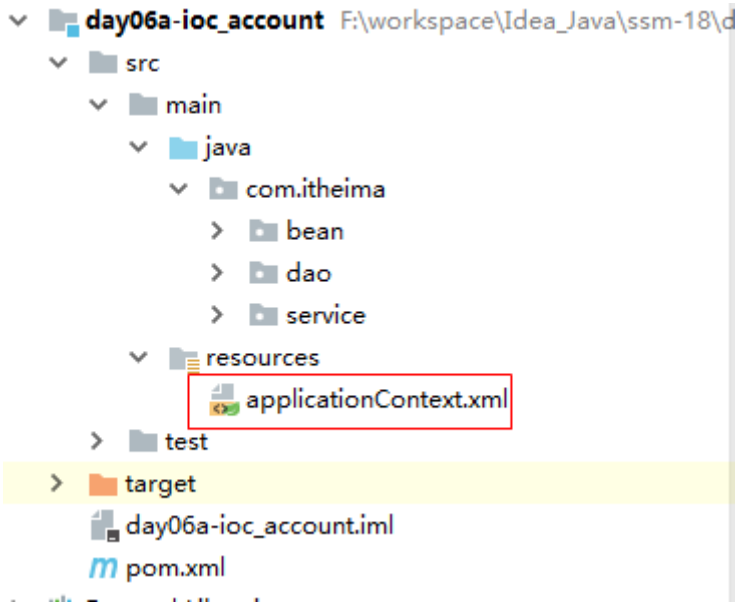
```
    @Override
    public Account findById(Integer id) throws Exception {
        return  accountDao.findById(id);
    }

    @Override
    public List<Account> findAll() throws Exception {
        return  accountDao.findAll();
    }
}
```

5. 编写控制层AccountController的代码

```
package com.itheima.controller;

import com.itheima.pojo.Account;
import com.itheima.service.AccountService;

import java.util.List;

/**
 * 包名:com.itheima.controller
 *
 * @author Leevi
 * 日期2020-06-26  15:08
 */
public class AccountController {
    private AccountService accountService;

    public void setAccountService(AccountService accountService) {
        this.accountService = accountService;
    }

    public List<Account> findAll(){
        return accountService.findAll();
    }

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

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

    public void deleteById(Integer id){
        accountService.deleteById(id);
    }
}
```

3.2配置步骤

- 创建applicationContext.xml




```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">
    <!--ioc-->
    <!--di-->
    <bean id="accountController"
class="com.itheima.controller.AccountController">
        <property name="accountService" ref="accountService"></property>
    </bean>
    <bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl">
        <property name="accountDao" ref="accountDao"></property>
    </bean>
    <bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl">
    </bean>
</beans>
```

3.3测试案例

```java
package com.itheima;

import com.itheima.controller.AccountController;
import com.itheima.pojo.Account;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.List;

/**
 * 包名:com.itheima
 *
 * @author Leevi
 * 日期2020-06-26  15:10
 * 1. 引入依赖
 * 2. 编写配置文件
 *    1. ioc: 创建对象
```

```
 *       1. AccountController
 *       2. AccountServiceImpl
 *       3. AccountDaoImpl
 *    2. 依赖注入
 *       1. AccountController中注入AccountServiceImpl的对象
 *       2. AccountServiceImpl中注入AccountDaoImpl的对象
 * 3. 创建spring的核心容器
 * 4. 从核心容器中获取AccountController,调用对象的方法
 */
public class TestSpring {
    @Test
    public void test01(){
        ApplicationContext act = new
ClassPathXmlApplicationContext("spring.xml");

        AccountController accountController = (AccountController)
act.getBean("accountController");

        List<Account> accountList = accountController.findAll();
        System.out.println(accountList);
    }
}
```

4.小结

第五章-Spring5 的新特性【了解】

知识点-Spring5 的新特性

1.目标

- [ ] 了解Spring5 的新特性

2.路径

1. 与 JDK 相关的升级
2. 核心容器的更新
3. JetBrains Kotlin 语言支持
4. 响应式编程风格
5. Junit5 支持
6. 依赖类库的更新

3.讲解

3.1与 JDK 相关的升级

3.1.1jdk 版本要求

spring5.0 在 2017 年 9 月发布了它的 GA（通用）版本。

该版本是基于 jdk8 编写的， 所以 jdk8 以下版本将无法使用。 同时，可以兼容 jdk9 版本。

注:

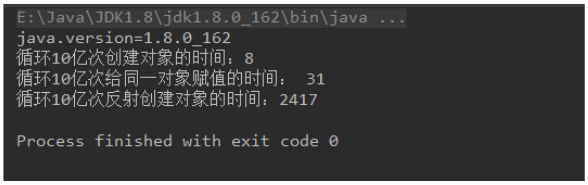
我们使用 jdk8 构建工程，可以降版编译。但是不能使用 jdk8 以下版本构建工程。

Spring 大量使用了反射

3.1.2利用 jdk8 版本更新的内容

- 基于 JDK8 的反射增强

```
public class Test {
    //循环次数定义： 10 亿次
    private static final int loopCnt = 1000 * 1000 * 1000;
    public static void main(String[] args) throws Exception {
        //输出 jdk 的版本
        System.out.println("java.version=" +
System.getProperty("java.version"));
        t1();
        t2();
        t3();
    }
    // 每次重新生成对象
    public static void t1() {
        long s = System.currentTimeMillis();
        for (int i = 0; i < loopCnt; i++) {
            Person p = new Person();
            p.setAge(31);
        }
        long e = System.currentTimeMillis();
        System.out.println("循环 10 亿次创建对象的时间： " + (e - s));
    }
    // 同一个对象
    public static void t2() {
        long s = System.currentTimeMillis();
        Person p = new Person();
        for (int i = 0; i < loopCnt; i++) {
            p.setAge(32);
        }
        long e = System.currentTimeMillis();
        System.out.println("循环 10 亿次给同一对象赋值的时间： " + (e - s));
    }
    //使用反射创建对象
    public static void t3() throws Exception {
        long s = System.currentTimeMillis();
        Class<Person> c = Person.class;
        Person p = c.newInstance();
        Method m = c.getMethod("setAge", Integer.class);
        for (int i = 0; i < loopCnt; i++) {
            m.invoke(p, 33);
        }
        long e = System.currentTimeMillis();
        System.out.println("循环 10 亿次反射创建对象的时间： " + (e - s));
    }
    static class Person {
        private int age = 20;
        public int getAge() {
            return age;
        }
        public void setAge(Integer age) {
            this.age = age;
        }
    }
}
```



- @NonNull 注解和@Nullable 注解的使用

  用 @Nullable 和 @NotNull 注解来显示表明可为空的参数和以及返回值。这样就够在编译的时候处理空值而不是在运行时抛出 NullPointerExceptions。

- 日志记录方面

  Spring Framework 5.0 带来了 Commons Logging 桥接模块的封装, 它被叫做 spring-jcl 而不是标准的 Commons Logging。当然，无需任何额外的桥接，新版本也会对 Log4j 2.x, SLF4J, JUL(java.util.logging) 进行自动检测。

3.2.核心容器的更新

Spring Framework 5.0 现在支持候选组件索引作为类路径扫描的替代方案。该功能已经在类路径扫描器中添加，以简化添加候选组件标识的步骤。应用程序构建任务可以定义当前项目自己的 META-INF/spring.components 文件。在编译时，源模型是自包含的， JPA 实体和 Spring 组件是已被标记的。从索引读取实体而不是扫描类路径对于小于 200 个类的小型项目是没有明显差异。但对大型项目影响较大。加载组件索引开销更低。因此，随着类数的增加，索引读取的启动时间将保持不变。 加载组件索引的耗费是廉价的。因此当类的数量不断增长，加上构建索引的启动时间仍然可以维持一个常数,不过对于组件扫描而言，启动时间则会有明显的增长。 这个对于我们处于大型 Spring 项目的开发者所意味着的，是应用程序的启动时间将被大大缩减。虽然 20 或者 30 秒钟看似没什么，但如果每天要这样登上好几百次，加起来就够你受的了。使用了组件索引的话，就能帮助你每天过的更加高效。

你可以在 Spring 的 Jira 上了解更多关于组件索引的相关信息。

3.3.JetBrains Kotlin 语言支持

Kolin概述：谷歌公司研发的，是一种支持函数式编程编程风格的面向对象语言。 Kotlin 运行在 JVM 之上，但运行环境并不
限于 JVM。

- Kolin 的示例代码：

```
{

  ("/movie" and accept(TEXT_HTML)).nest {GET("/", movieHandler::findAllView)

              GET("/{card}", movieHandler::findOneView)
        }

  ("/api/movie" and accept(APPLICATION_JSON)).nest {

  GET("/", movieApiHandler::findAll)

  GET("/{id}", movieApiHandler::findOne)
  }
}
```

- Kolin 注册 bean 对象到 spring 容器：

```
val context = GenericApplicationContext {
  registerBean()
  registerBean { Cinema(it.getBean()) }
}
```

3.4.响应式编程风格

此次 Spring 发行版本的一个激动人心的特性就是新的响应式堆栈 WEB 框架。这个堆栈完全的响应式且非阻塞，适合于事件循环风格的处理，可以进行少量线程的扩展。

Reactive Streams 是来自于 Netflix, Pivotal, Typesafe, Red Hat, Oracle, Twitter 以及Spray.io 的工程师特地开发的一个 API。它为响应式编程实现 的实现提供一个公共 的 API，好实现Hibernate 的 JPA。这里 JPA 就是这个 API, 而 Hibernate 就是实现。Reactive Streams API 是 Java 9 的官方版本的一部分。在 Java 8 中, 你会需要专门引入依赖来使用 Reactive Streams API。Spring Framework 5.0 对于流式处理的支持依赖于 Project Reactor 来构建, 其专门实现了Reactive Streams API。

Spring Framework 5.0 拥有一个新的 spring-webflux 模块，支持响应式 HTTP 和 WebSocket 客户端 Spring Framework 5.0 还提供了对于运行于服务器之上，包含了 REST, HTML, 以及 WebSocket 风格交互的响应式网页应用程序的支持。在 spring-webflux 中包含了两种独立的服务端编程模型：基于注解：使用到了@Controller 以及 Spring MVC 的其它一些注解；使用 Java 8 lambda 表达式的函数式风格的路由和处理。有 了 Spring Webflux, 你 现 在 可 以 创 建 出 WebClient, 它 是 响 应 式 且 非 阻 塞 的 ， 可 以 作 为RestTemplate 的一个替代方案。

- 这里有一个使用 Spring 5.0 的 REST 端点的 WebClient 实现：

```
WebClient webClient = WebClient.create();
Movie movie =
webClient.get().uri("http://localhost:8080/movie/42").accept(MediaType.APPLICATI
ON_JSON).exchange().then(response -> response.bodyToMono(Movie.class));
```

3.5. Junit5 支持

完全支持 JUnit 5 Jupiter，所以可以使用 JUnit 5 来编写测试以及扩展。此外还提供了一个编程以及扩展模型， Jupiter 子项目提供了一个测试引擎来在 Spring 上运行基于 Jupiter 的测试。

另外， Spring Framework 5 还提供了在 Spring TestContext Framework 中进行并行测试的扩展。针对响应式编程模型， spring-test 现在还引入了支持 Spring WebFlux 的 WebTestClient 集成测试的支持，类似于 MockMvc，并不需要一个运行着的服务端。使用一个模拟的请求或者响应，WebTestClient就可以直接绑定到 WebFlux 服务端设施。

你可以在这里找到这个激动人心的 TestContext 框架所带来的增强功能的完整列表。当然， Spring Framework 5.0 仍然支持我们的老朋友 JUnit! 在我写这篇文章的时候， JUnit 5 还只是发展到了 GA 版本。对于 JUnit4， Spring Framework 在未来还是要支持一段时间的。

3.6.依赖类库的更新

- 终止支持的类库

  Portlet.
  Velocity.
  JasperReports.
  XMLBeans.
  JDO.
  Guava.

- 支持的类库

Jackson 2.6+
EhCache 2.10+ / 3.0 GA
Hibernate 5.0+
JDBC 4.0+
XmlUnit 2.x+
OkHttp 3.x+
Netty 4.1+

Spring 02

学习目标

- [ ] 能够实现spring基于注解的IoC案例
- [ ] 能够编写spring的IOC的注解代码
- [ ] 能够编写spring使用注解实现组件扫描
- [ ] 能够说出spring的IOC的相关注解的含义
- [ ] 能够理解AOP相关概念
- [ ] 能够说出AOP相关术语的含义
- [ ] 能够掌握基于XML的AOP配置

第一章-Spring的IOC注解开发

学习基于注解的 IoC 配置，大家脑海里首先得有一个认知，即注解配置和 xml 配置要实现的功能都是一样的，都是要降低程序间的耦合。只是配置的形式不一样。

关于实际的开发中到底使用 xml 还是注解，每家公司有不同的习惯。所以这两种配置方式我们都需要掌握。

案例-注解开发入门

1.目标

- [ ] 能够编写spring的IOC的注解代码(代替xml里面配置的Bean标签)

2.分析

1. 创建工程, 添加坐标
2. 在类上面添加@Component
3. 在applicationContext.xml 开启包扫描

3.实现

3.1创建Maven工程,添加依赖

```
<dependencies>
    <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.2.RELEASE</version>
    </dependency>
 </dependencies>
```

3.2使用@Component注解配置管理的资源

- eg: 需要给AccountServiceImpl

```
@Component("accountService")
public class AccountServiceImpl implements AccountService {} // 等同于 在xml里面,
<bean id="accountService" class="具体的类">
```

3.3引入context的命名空间

- applicationContext.xml中需要引入context的命名空间,可在xsd-configuration.html中找到

```
<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">


</beans>
```

3.4 配置扫描

在beans标签内部，使用context:component-scan ， 让spring扫描该基础包下的所有子包注解

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

4.小结

1. 在类上面添加@Component("id")
2. 在applicationContext.xml 开启包扫描

案例-注解开发进阶

1.目标

- [ ] 能够编写spring使用注解实现组件扫描
- [ ] 能够说出spring的IOC的相关注解的含义

2.路径

1. 用于创建对象的注解
2. 用于改变作用范围的注解
3. 和生命周期相关的注解

3.讲解

3.1用于创建对象的

相对于`相当于： <bean id="" class="">`

3.1.1@Component

作用:
　把资源让 spring 来管理。相当于在 xml 中配置一个 bean。
属性:
　value：指定 bean 的 id。如果不指定 value 属性，默认 bean 的 id 是当前类的类名。首字母小写。

3.1.2 @Controller @Service @Repository

　web里面的三层结构 中的所有类，在spring里面都称之为 Component （组件） ， 但是它也提供了更详细的注解来针对不同的层级声明 。

　三个衍生注解如下:

@Controller	:修饰WEB层类	--->action | SpringMVC
@Service	:修饰业务层类	--->service
@Repository	:修饰DAO层类	--->dao

　在需要spring创建对象的类上面使用注解 @Component(“us”) 即可.Spring看类上是否有该注解，如果有该注解，生成这个类的实例。如果不指定value值， 那么默认的id值就是类名的名字， 第一个字母小写.

3.2用于改变作用范围的@scope

@scope

　singleton: 单例(默认)

　prototype:多例

　@Scope注解用来描述类的作用范围的，默认值singleton。如同xml中bean标签的属性scope `<bean scope=""/>`.如果配置成多例的使用prototype。

```
@Scope("prototype")
@Component("accountService")
public class AccountServiceImpl implements AccountService {}
```

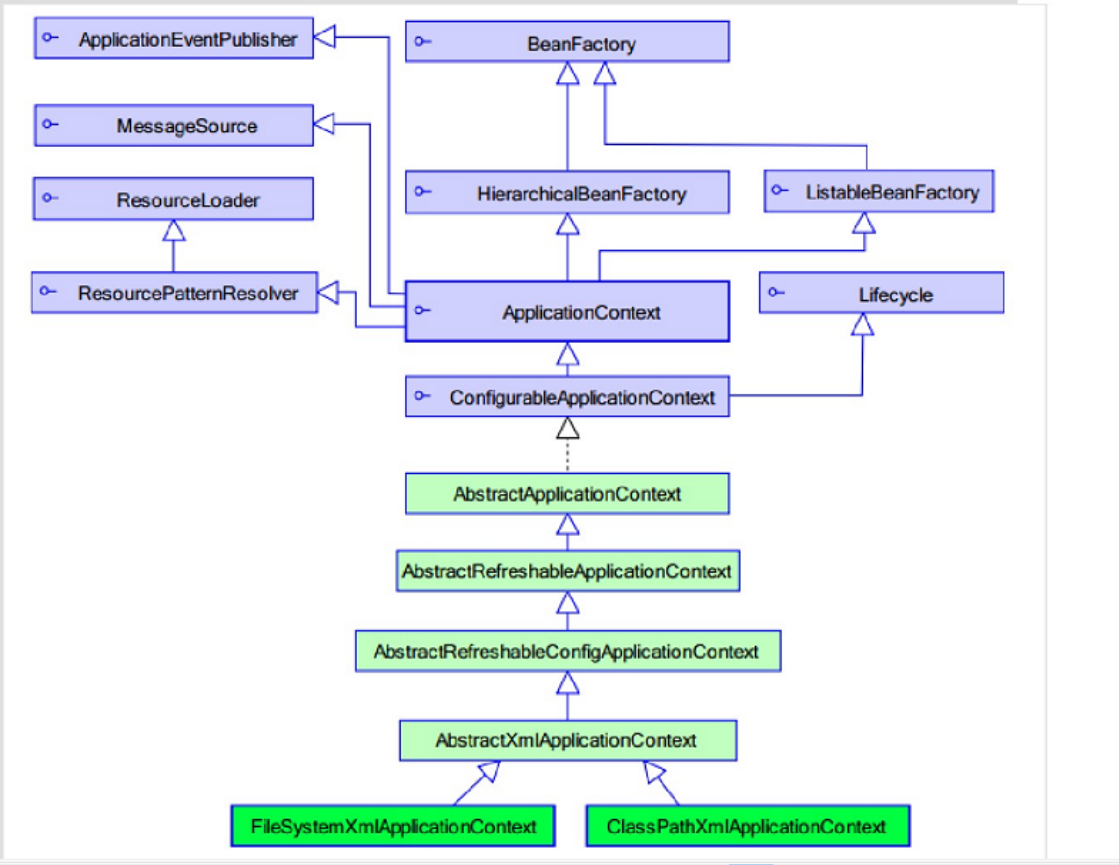
3.3和生命周期相关的【了解】

- 初始化和销毁回调方法对应的注解

　　@PostConstrut:如同xml中bean标签的属性init-method `<bean init-method=""/>`,用来设置spring框架初始化此类实例时调用的初始化方法,标注在此类的初始化方法上

　　@PreDestroy:如同xml中bean标签的属性init-method `<bean destroy-method=""/>`,用来设置spring框架销毁此类实例时调用的销毁方法,标注在此类的销毁方法上

> 注意:这两个注解都是配在方法上的



4.小结

1. 注册Bean的 相当于配置了 `<bean id="" class=""/>`
    - @Component("id")
    - @Controller("id") 配置web层
    - @Service("id") 配置Service层

- @Repository("id") 配置持久层

> 如果id不配, 默认就是类的名字,首字母小写

2. 配置bean的作用范围 `<bean scope=""></bean>`
    - @Scope("类型")
        - singleton 单例【默认】
        - prototype 多例
3. 配置和生命周期相关的 `<bean  init-method="" destroy-method=""></bean>`
    - @PostConstruct `init-method=""`
    - @PreDestroy `destroy-method=""`

知识点-使用注解注入属性

1.目标

- [ ] 掌握使用注解注入属性

2.路径

1. @Value : 注入简单类型的数据

2. @Autowired
3. @Qualifier
4. @Resource

3.讲解

3.1@Value

- 作用:

  注入基本数据类型和 String 类型数据的

- 属性:

  value：用于指定值 , 可以通过表达式动态获得内容再赋值

- 实例:

```
@Value("奥巴马")
private String name;
```

3.2@Autowired

- 作用:

  自动按照类型注入。当使用注解注入属性时， set 方法可以省略。它只能注入其他 bean 类型。

  如果只有一个实现类, 可以自动注入成功

  如果有两个或者两个以上的实现类, 找到变量名一致的id对象给注入进去, 如果找不到,就报错

- 实例:

```
@Component("accountService")
public class AccountServiceImpl implements AccountService {
    @Autowired
    //当有多个AccountDao实现类时候, @Autowired会在在Spring容器里面找id为accountDao的对象注入,找不到就报错
    private AccountDao accountDao;
    @Override
    public void save() {
        System.out.println("AccountServiceImpl---save()");
        accountDao.save();
    }
}
```

3.3@Qualifier

- 作用

  在自动按照类型注入(@Autowired)的基础之上，再按照Bean的id注入。

  它在给字段注入时不能独立使用，必须和@Autowire一起使用;

  但是给方法参数注入时，可以独立使用。

- 属性

  value：指定bean的id。

- 实例

```
@Component("accountService")
public class AccountServiceImpl implements AccountService {
    @Autowired
    @Qualifier(value = "accountDao02")
    private AccountDao accountDao;
    @Override
    public void save() {
        System.out.println("AccountServiceImpl---save()");
        accountDao.save();
    }
}
```

3.4@Resource

如果上面一个接口有多种实现，那么现在需要指定找具体的某一个实现，那么可以使用@Resource

```
@Component("accountService")
public class AccountServiceImpl implements AccountService {
    @Resource(name = "accountDao02")
    private AccountDao accountDao;
    @Override
    public void save() {
        System.out.println("AccountServiceImpl---save()");
        accountDao.save();
    }
}
```

4.小结

1. 注入简单类型 `@Value("值/表达式")`
2. 注入对象类型
    - 如果只有一个实现类, 建议使用@Autowired
    - 如果有多个实现类, 建议使用@Resource("name="id")

知识点-混合开发

1.目标

- [ ] 掌握混合(xml和注解)开发

2.路径

1. 注解和XML开发比较
2. 混合开发特点
3. 混合开发环境搭建

3.讲解

3.1注解和XML比较

	基于XML配置	基于注解配置
Bean定义	<bean id="..." class="..." />	@Component 衍生类@Repository @Service @Controller
Bean名称	通过 id或name 指定	@Component("person")
Bean注入	<property> 或者通过p命名空间	@Autowired 按类型注入 @Qualifier按名称注入
生命过程、Bean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，使用其它	Bean的实现类由用户自己开发

- xml
  - 优点: 方便维护, 改xml文件
  - 缺点: 相对注解而言, 麻烦一点
- 注解
  - 优点: 开发简洁方便
  - 缺点: 维护没有xml那么方便, 需要改源码

3.2混合开发特点

使用xml(==注册==bean)来管理bean

使用注解来==注入==属性

3.3混合开发环境搭建

- 创建spring配置文件，编写头信息配置

```
<?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
     ">
</beans>
```

- 配置扫描

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

- 在xml配置中添加Bean的管理
- 在被管理bean对应的类中，为依赖添加注解配置

4.小结

1. xml注册(管理bean)
2. 注解注入(给对象里面字段/属性赋值)

知识点-spring整合mybatis

目标

能够使用spring整合mybatis

步骤

1. 引入mybatis以及整合的相关依赖
2. 在spring的IOC容器中创建DataSource对象(可以使用spring内置的DataSource)
3. 在spring的IOC容器中创建SqlSessionFactoryBean对象，并且可以指定要配置别名的包或者是加载核心配置文件
4. 在spring的IOC容器中创建MapperScannerConfigurer对象，用于扫描Dao接口创建代理对象
5. 通过注解的方式注入Dao代理对象

实现

1. 引入依赖

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <!--spring-->
    <spring.version>5.0.2.RELEASE</spring.version>
    <!--日志打印框架-->
    <slf4j.version>1.6.6</slf4j.version>
    <log4j.version>1.2.12</log4j.version>
    <!--mysql-->
    <mysql.version>5.1.6</mysql.version>
    <!--mybatis-->
    <mybatis.version>3.4.5</mybatis.version>
</properties>

<dependencies>
    <!--引入依赖-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
```

```
        <version>${spring.version}</version>
    </dependency>
    <!--mysql依赖-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>${mysql.version}</version>
    </dependency>
    <!-- log start -->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>${log4j.version}</version>
    </dependency>

    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>${slf4j.version}</version>
    </dependency>

    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>${slf4j.version}</version>
    </dependency>
    <!-- log end -->

    <!--mybatis的依赖-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>${mybatis.version}</version>
    </dependency>

    <!--
      mybatis整合spring的依赖
    -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>1.3.0</version>
    </dependency>

    <!--lombok-->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.10</version>
    </dependency>

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

```
</dependencies>
```

2. spring.xml文件配置

```
<?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">
    <!--包扫描，扫描注解-->
    <context:component-scan base-package="com.itheima"/>
    <!--spring整合mybatis的配置文件-->
    <!--
        要进行ssm的整合，第一步:将sqlSessionFactory对象交给spring核心容器管理
        sqlSessionFactoryBean对象，就是管理SQLSession创建Dao代理对象的
        创建代理对象的目的是执行SQL语句

    -->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--将数据源注入进来-->
        <property name="dataSource" ref="dataSource"/>
        <!--注入核心配置文件的路径-->
        <property name="configLocation" value="classpath:SqlMapConfig.xml"/>
    </bean>
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="username" value="root"/>
        <property name="password" value="123"/>
        <property name="url" value="jdbc:mysql:///day11?characterEncoding=utf-
8"/>
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    </bean>
    <!--
        第二步:一定要让spring核心容器去扫描dao接口，去加载dao接口的代理对象
    -->
    <bean id="mapperScannerConfigurer"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <!--指定要扫描的dao接口的包-->
        <property name="basePackage" value="com.itheima.dao"/>
    </bean>
</beans>
```

案例-使用mybatis和注解改造增删改查案例

1.需求

- [ ] 使用注解改造练习

2.分析

1. 在业务类上面添加@Service
2. 在业务类的accountDao字段上面添加@Autowird注解
3. 开启包扫描

3.实现

- AccountController的代码

```
@Controller
public class AccountController {
    @Autowired
    private AccountService accountService;
    public List<Account> findAll(){
        return accountService.findAllAccount();
    }

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

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

    public void deleteById(Integer id){
        accountService.deleteById(id);
    }
}
```

- 使用注解配置业务层

```
@Service
public class AccountServiceImpl implements AccountService {
    @Autowired
    private AccountDao accountDao;

    @Override
    public List<Account> findAllAccount() {
        return accountDao.findAllAccount();
    }

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

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

    @Override
    public void deleteById(Integer id) {
        accountDao.deleteById(id);
    }
```

```
}
```

- AccountDao的代码

```
public interface AccountDao {
    List<Account> findAllAccount();

    void add(Account account);

    void update(Account account);

    void deleteById(Integer id);
}
```

- AccountDao.xml配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.itheima.dao.AccountDao">
    <select id="findAllAccount" resultType="Account">
        select * from account
    </select>

    <insert id="add" parameterType="Account">
        insert into account values (null,#{name},#{money})
    </insert>

    <update id="update" parameterType="Account">
        update account set name=#{name},money=#{money} where id=#{id}
    </update>

    <delete id="deleteById" parameterType="int">
        delete from account where id=#{id}
    </delete>
</mapper>
```

- 修改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"
       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">
    <!--配置文件加上注解的混合开发方式-->
    <!--1. 包扫描-->
    <context:component-scan base-package="com.itheima"/>

    <!--导入其它的xml配置文件-->
    <import resource="classpath:application-mybatis.xml"/>
</beans>
```

- application-mybatis.xml的代码

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       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
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">
    <!--spring整合mybatis-->
    <!--引入外部的properties属性文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!--1. 创建DataSource对象-->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="driverClassName" value="${jdbc.driver}"></property>
    </bean>

    <!--2. 创建SqlSessionFactoryBean对象-->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"></property>
        <!--加载mybatis的核心配置文件-->
        <property name="configLocation" value="classpath:SqlMapConfig.xml">
</property>
    </bean>

    <!--3. 创建MapperScannerConfigurer对象-->
    <bean id="mapperScannerConfigurer"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="com.itheima.dao"></property>
    </bean>
</beans>
```

- jdbc.properties配置文件

```
jdbc.username=root
jdbc.password=123
jdbc.url=jdbc:mysql:///spring_db?characterEncoding=utf-8
jdbc.driver=com.mysql.jdbc.Driver
```

- mybatis的核心配置文件SqlMapConfig.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <typeAliases>
        <package name="com.itheima.pojo"/>
    </typeAliases>
</configuration>
```

4.小结

案例-使用纯注解改造增删改查(了解)

1.需求

- [ ] 使用纯注解改造作业

2.分析

基于注解的IoC配置已经完成，但是大家都发现了一个问题：我们依然离不开spring的xml配置文件，那么能不能不写这个bean.xml，所有配置都用注解来实现呢?

当然，同学们也需要注意一下，我们选择哪种配置的原则是简化开发和配置方便，而非追求某种技术

学习纯注解开发原因:

1. 方便大家学习SpringBoot
2. 以防万一真遇到了Spring纯注解开发(**不可能**)

```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"
       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">
    <!--包扫描，扫描注解-->
    <context:component-scan base-package="com.itheima"/>
    <!--spring整合mybatis的配置文件-->
    <!--
        要进行ssm的整合，第一步:将sqlSessionFactory对象交给spring核心容器管理
        sqlSessionFactoryBean对象，就是管理SQLSession创建Dao代理对象的
        创建代理对象的目的是执行SQL语句

    -->
    <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
        <!--将数据源注入进来-->
        <property name="dataSource" ref="dataSource"/>
        <!--注入核心配置文件的路径-->
        <property name="configLocation" value="classpath:SqlMapConfig.xml"/>
    </bean>
```

```
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="username" value="root"/>
        <property name="password" value="123"/>
        <property name="url" value="jdbc:mysql:///day11?characterEncoding=utf-
8"/>
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    </bean>
    <!--
        第二步:一定要让spring核心容器去扫描dao接口，去加载dao接口的代理对象
    -->
    <bean id="mapperScannerConfigurer"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <!--指定要扫描的dao接口的包-->
        <property name="basePackage" value="com.itheima.dao"/>
    </bean>
</beans>
```

3.实现

3.1 @Configuration

- 作用:

  用于指定当前类是一个spring配置类，当创建容器时会从该类上加载注解。

  获取容器时需要使用AnnotationApplicationContext(类.class)。

- 属性:

  value:用于指定配置类的字节码

- 示例代码:

```
@Configuration
public class SpringConfiguration {

}
```

3.2 @ComponentScan

- 作用:

  用于指定spring在初始化容器时要扫描的包。作用和在spring的xml配置文件中的:
  `<context:component-scan base-package="com.itheima"/>`是一样的。

- 属性:

  basePackages：用于指定要扫描的包。和该注解中的value属性作用一样。

- 示例代码:

```
@Configuration
@ComponentScan("com.itheima")
public class SpringConfiguration {

}
```

3.3@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。

- 属性：

  name：给当前@Bean注解方法创建的对象指定一个名称(即bean的id）。

- 示例代码：

```
@Configuration
@ComponentScan("com.itheima")
public class SpringConfiguration {
    private String driver = "com.mysql.jdbc.Driver";
    private String url = "jdbc:mysql:///spring_day02";
    private String username = "root";
    private String password = "123456";
    @Bean("dataSource")
    public DataSource createDataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        dataSource.setUrl(url);
        dataSource.setDriverClassName(driver);
        return dataSource;
    }
    @Bean
    public SqlSessionFactoryBean createSqlSessionFactory(DataSource dataSource){
        SqlSessionFactoryBean sqlSessionFactory = new SqlSessionFactoryBean();
        //设置dataSource
        sqlSessionFactory.setDataSource(dataSource);
        //设置要加载的mybatis的核心配置文件的路径
        Resource resource = new ClassPathResource("SqlMapConfig.xml");
        sqlSessionFactory.setConfigLocation(resource);
        return sqlSessionFactory;
    }
    @Bean
    public MapperScannerConfigurer createMapperScannerConfigurer(){
        MapperScannerConfigurer mapperScannerConfigurer = new
MapperScannerConfigurer();
        mapperScannerConfigurer.setBasePackage("com.itheima.dao");
        return mapperScannerConfigurer;
    }
}
```

3.4@Import（了解）

- 作用：

  用于导入其他配置类，在引入其他配置类时，可以不用再写@Configuration注解。当然，写上也没问题。

- 属性：

  value[]：用于指定其他配置类的字节码。

- 示例代码：

```
/**
 * 该类是一个配置类，它的作用和bean.xml是一样的
 */
@Configuration
@ComponentScan("com.itheima")
@Import({JdbcConfig.class})
public class SpringConfiguration {


}
```

```
public class JdbcConfig {
    private String driver = "com.mysql.jdbc.Driver";
    private String url = "jdbc:mysql:///spring_day02";
    private String username = "root";
    private String password = "123456";
    @Bean("dataSource")
    public DataSource createDataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        dataSource.setUrl(url);
        dataSource.setDriverClassName(driver);
        return dataSource;
    }
    @Bean
    public SqlSessionFactoryBean createSqlSessionFactory(DataSource dataSource){
        SqlSessionFactoryBean sqlSessionFactory = new SqlSessionFactoryBean();
        //设置dataSource
        sqlSessionFactory.setDataSource(dataSource);
        //设置要加载的mybatis的核心配置文件的路径
        Resource resource = new ClassPathResource("SqlMapConfig.xml");
        sqlSessionFactory.setConfigLocation(resource);
        return sqlSessionFactory;
    }
    @Bean
    public MapperScannerConfigurer createMapperScannerConfigurer(){
        MapperScannerConfigurer mapperScannerConfigurer = new
MapperScannerConfigurer();
        mapperScannerConfigurer.setBasePackage("com.itheima.dao");
        return mapperScannerConfigurer;
    }
}
```

3.5@PropertySource

- 作用：

  用于加载.properties文件中的配置。例如我们配置数据源时，可以把连接数据库的信息写到properties配置文件中，就可以使用此注解指定properties配置文件的位置。

- 属性：

  value[]：用于指定properties文件位置。如果是在类路径下，需要写上classpath:

- 示例代码：

jdbc.properties

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_day02
jdbc.username=root
jdbc.password=123456
```

JdbcConfig.java

```
@PropertySource(value = {"classpath:jdbc.properties"})
public class JdbcConfig {
    @Value("${jdbc.username}")
    private String username;

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

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

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

    @Bean("dataSource")
    public DataSource createDataSource(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        dataSource.setUrl(url);
        dataSource.setDriverClassName(driver);
        return dataSource;
    }

    @Bean
    public SqlSessionFactoryBean createSqlSessionFactory(DataSource dataSource){
        SqlSessionFactoryBean sqlSessionFactory = new SqlSessionFactoryBean();
        //设置dataSource
        sqlSessionFactory.setDataSource(dataSource);
        //设置要加载的mybatis的核心配置文件的路径
        Resource resource = new ClassPathResource("SqlMapConfig.xml");
        sqlSessionFactory.setConfigLocation(resource);
        return sqlSessionFactory;
    }
    @Bean
    public MapperScannerConfigurer createMapperScannerConfigurer(){
        MapperScannerConfigurer mapperScannerConfigurer = new
MapperScannerConfigurer();
        mapperScannerConfigurer.setBasePackage("com.itheima.dao");
        return mapperScannerConfigurer;
    }
}
```

3.6通过注解获取容器

```
ApplicationContext ac = new
AnnotationConfigApplicationContext(SpringConfiguration.class);
```

3.小结

第二章-Spring整合测试

知识点-Spring整合测试(简单的了解)

1.目标

在测试类中，每个测试方法都有以下两行代码：

ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");

AccountService as= ac.getBean("accountService",AccountService.**class**);

这两行代码的作用是获取容器，如果不写的话，直接会提示空指针异常。所以又不能轻易删掉。

2.分析

早前我们测试业务逻辑，一般都使用Junit 框架来测试， 其实在spring框架里面它也做出来了自己的一套测试逻辑， 里面是对junit进行了整合包装。让我们在执行spring的单元测试上面，可以少写一些代码.

1. 导入Spring整合单元测试的坐标
2. 在测试类上面添加注解(指定运行的环境, 加载配置文件或者配置类)

3.实现

- 导入spring整合Junit的坐标

```
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.0.2.RELEASE</version>
    </dependency>
```

- 在测试类上面标记注解

```
@RunWith(SpringJUnit4ClassRunner.class)//指明运行的测试环境
//@ContextConfiguration("classpath:applicationContext.xml")//指明spring框架的加载的配置文件【有applicationContext.xml解使用这种】
```

```java
@ContextConfiguration(classes = SpringConfiguration.class)//指明spring框架的加载的配置类【纯注解使用这种】
public class DbTest
{
    @Autowired
    private AccountService accountService;
    @Test
    public void testSaveAccount() throws Exception {
        Account account = new Account();
        account.setName("王二麻子");
        account.setMoney(100f);
        accountService.save(account);
    }

}
```

4.小结

1. 添加Spring整合单元测试的坐标
2. 在测试类上面添加注解
    - @RunWith(SpringJUnit4ClassRunner.class) 指定运行的环境
    - @ContextConfiguration() 加载配置文件或者配置类的

第三章-混合开发方式补充

知识点-import标签

目标

使用import标签导入外部的xml配置文件

实现

```xml
<import resource="classpath:application-mybatis.xml"/>
```

知识点-引入外部properties文件

目标

引入外部的properties属性文件

实现

```
<context:property-placeholder location="classpath:jdbc.properties"/>

<!--1. 创建DataSource对象-->
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="username" value="${jdbc.username}"></property>
    <property name="password" value="${jdbc.password}"></property>
    <property name="url" value="${jdbc.url}"></property>
    <property name="driverClassName" value="${jdbc.driver}"></property>
</bean>
```

第四章-AOP相关的概念

知识点-AOP概述

1.目标

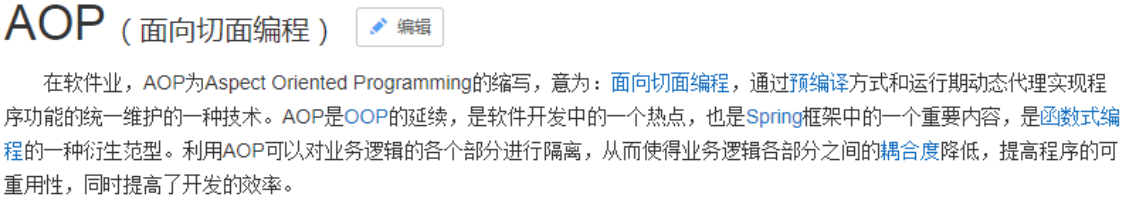
- [ ] 能够理解AOP相关概念

2.路径

1. 什么是AOP
2. AOP的作用和优势
3. AOP实现原理

3.讲解

3.1什么是AOP



AOP：全称是AspectOriented Programming, 即面向切面编程。在不修改源码的基础上，对我们的已有方法进行增强。

说白了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，进行增强

3.2AOP的作用和优势

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

3.3AOP实现原理

使用动态代理技术

- JDK动态代理: 必须需要接口的
- Cglib动态代理: 不需要接口的,只需要类就好了

4.小结

1. AOP: 面向切面编程
2. 作用: 不需要改变源代码 对目标方法进行增强
3. 底层实现: 动态代理

知识点-AOP的具体应用

1.需求

在所有的dao层的save()方法逻辑调用之前,进行权限的校验

2.分析

在AOP 这种思想还没有出现的时候，我们解决 切面的问题思路无非有以下两种：

1. 方式一:通过静态方法实现(缺点:需要修改源码,后期不好维护)

   把需要添加的代码抽取到一个地方，然后在需要添加那些方法中引用

2. 方式二:通过继承方案来解决(缺点:需要修改源码,继承关系复杂,后期不好维护)

   抽取共性代码到父类， 子类在需要的位置，调用父类方法。

上述两种方式的缺点

都会打破原来代码的平静（也就是必须要修改代码， 或者就是必须事先固定好。） 如果我们想在原有代码基础上扩展。 并且不改动原来的代码, 就可以使用AOP了。

其实AOP 字面直译过来是面向切面编程，其实通俗一点它就是对我们的具体某个方法进行了增强而已。在之前我们对某个方法进行增强无非是两种手段 ， 一种是装饰者模式 、 一种是代理模式 （静态代理 & 动态代理） 。 AOP 的底层使用的是动态代理方式 。

3.实现

AOP 的底层动态代理实现有两种方案。

一种是使用JDK的动态代理。 这种是早前我们在前面的基础增强说过的。 这一种主要是针对有接口实现的情况。 它的底层是创建接口的实现代理类， 实现扩展功能。也就是我们要增强的这个类，实现了某个接口，那么我就可以使用这种方式了. 而另一种方式是使用了cglib 的动态代理，这种主要是针对没有接口的方式，那么它的底层是创建被目标类的子类，实现扩展功能.

3.1JDK方式

==要求: 必须有接口==

```
public class B_JDKProxy {
```

```
    @Test
    public void fun01(){
         AccountDao accountDao = new AccountDaoImpl();

        AccountDao proxyDao =  (AccountDao)
Proxy.newProxyInstance(accountDao.getClass().getClassLoader(),
accountDao.getClass().getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                if("save".equals(method.getName())){
                    System.out.println("权限校验...");
                    return method.invoke(accountDao,args);
                }
                return method.invoke(accountDao, args);
            }
        });
        proxyDao.save();
    }
}
```

3.2 CgLib方式【了解】

- 添加坐标

```
  <dependencies>
    <!--Spring核心容器-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>
    <!--SpringAOP相关的坐标-->
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.8.7</version>
    </dependency>

    <!--Spring整合单元测试-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>
    <!--单元测试-->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

- 使用CgLib方式实现: 第三方的代理机制，不是jdk自带的. 没有实现接口的类产生代理，使用的是字节码的增强技术，其实就是产生这个类的子类。

==不需要有接口==

```
public class C_CglibProxy {
    @Test
    public void fun01(){
         AccountDaoImpl accountDao = new AccountDaoImpl();

        //创建enhancer对象
        Enhancer enhancer  = new Enhancer();
        //设置代理的父类
        enhancer.setSuperclass(AccountDaoImpl.class);

        enhancer.setCallback(new MethodInterceptor() {

            @Override
            public Object intercept(Object obj, Method method, Object[] args,
MethodProxy methodProxy) throws Throwable {
                if("save".equals(method.getName())){
                    System.out.println("权限校验...");
                    return method.invoke(accountDao, args);
                }
                return method.invoke(accountDao, args);
            }
        });

        AccountDaoImpl accountDaoProxy = (AccountDaoImpl) enhancer.create();
        accountDaoProxy.save();
    }
}
```

4.小结

1. AOP的底层就是封装了动态代理
    - JDK的动态代理
    - CgLib的动态代理
2. JDK的动态代理: 依赖接口的, 必须有接口

   CgLib的动态代理: 如果要代理的类没有实现接口，那么就只能使用cglib的动态代理

知识点-AOP术语

1.目标

- [ ] 能够说出AOP相关术语的含义

2.路径

1. Spring中的AOP说明
2. AOP中的术语

3.讲解

3.1 学习spring中的AOP要明确的事

- 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，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。

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。

3.2.AOP中的术语

- JoinPoint: 连接点(所有可以被增强的方法)

  类里面哪些方法可以被增强，这些方法称为连接点. 在spring的AOP中，指的是业务层的类的所有现有的方法。

- Pointcut: 切入点(具体项目中真正已经被增强的方法)

  在类里面可以有很多方法被增强，但是实际开发中，我们只对具体的某几个方法而已，那么这些实际增强的方法就称之为切入点

- Advice: 通知/增强 (具体用于增强方法的代码)

  增强的逻辑、称为增强，比如给某个切入点(方法) 扩展了校验权限的功能，那么这个校验权限即可称之为增强 或者是通知

  通知分为:

  前置通知： 在原来方法之前执行.

  后置通知： 在原来方法之后执行. 特点: 可以得到被增强方法的返回值

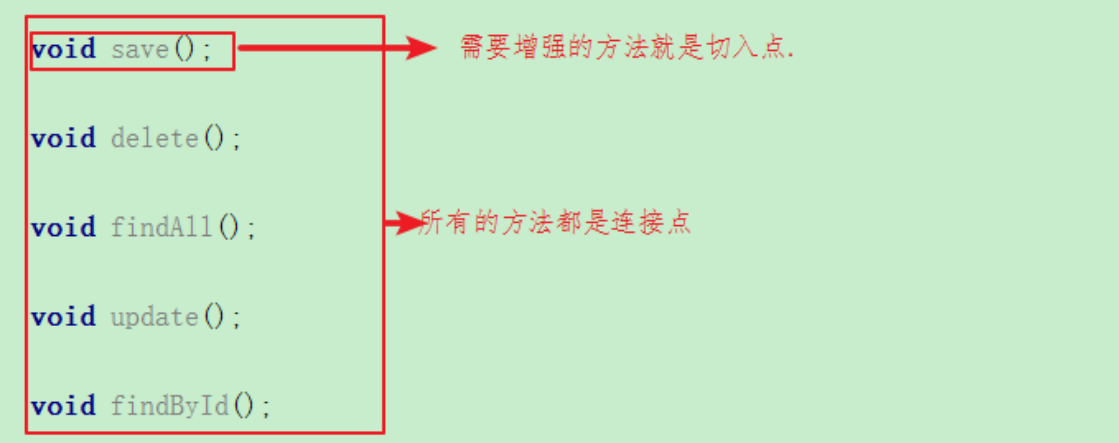
  环绕通知：在方法之前和方法之后执行. 特点:可以阻止目标方法执行

  异常通知： 目标方法出现异常执行. 如果方法没有异常,不会执行. 特点:可以获得异常的信息

  最终通知： 指的是无论是否有异常，总是被执行的。

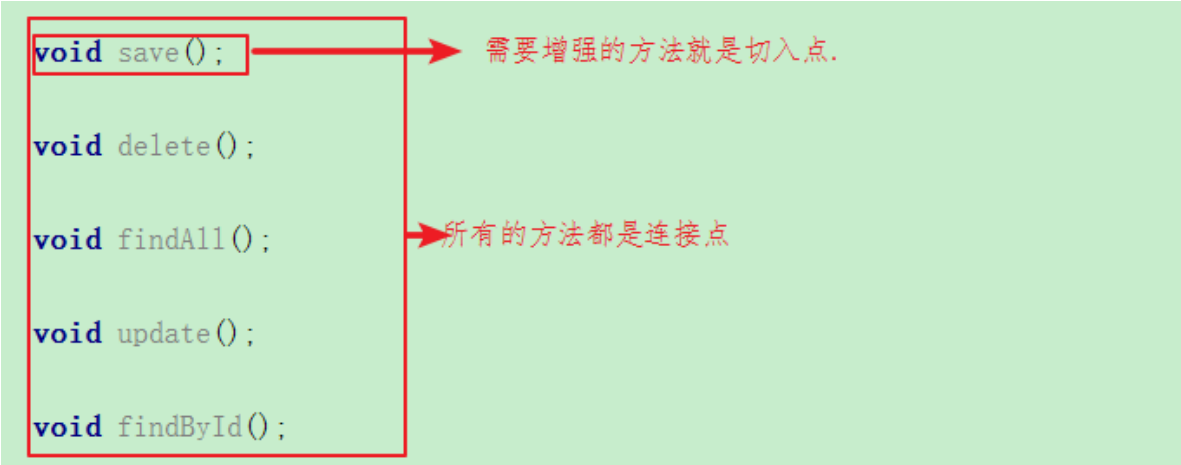
- Aspect: 切面(所有的通知都是在切面中的)

  切入点和通知的结合。



4.小结

1. 使用AOP: 逻辑需要我们写的, 怎么切是Spring做(运行阶段), 源码阶段看不到
2. 术语:




```
public class MyAspect {

    public void checkPrivilege(){
        System.out.println("权限校验...");
    }

}
```

通知

切面: 让通知和切入点进行结合

Spring_03

学习目标

- [ ] 能够掌握基于注解的AOP配置
- [ ] 能够使用spring的jdbc的模板
- [ ] 能够配置spring的连接池
- [ ] 能够使用jdbc模板完成增删改查的操作
- [ ] 能够应用声明式事务
- [ ] 能够理解事务的传播行为

第一章-Spring中的AOP

知识点-基于XML的AOP配置

1.目标

- [ ] 能够编写Spring的AOP中不同通知的代码

2.路径

1. 前置通知
2. 后置通知
3. 环绕通知

4. 异常通知
5. 最终通知

3.讲解

3.1前置通知

需求: 在dao的save()方法调用之前进行权限的校验

实现步骤:

1. 导入坐标
2. 定义Dao和增强的逻辑, 进行注册
3. 配置AOP
    - 配置切入点
    - 配置切面(切入点和通知结合)

实现:

- 导入坐标

```
<dependencies>
  <!--Spring核心容器-->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.2.RELEASE</version>
  </dependency>
  <!--SpringAOP相关的坐标-->
  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.7</version>
  </dependency>

  <!--Spring整合单元测试-->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.0.2.RELEASE</version>
  </dependency>
  <!--单元测试-->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

- 导入约束

```
<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">
```

- 定义被增强的业务逻辑类和切面类(增强的)

```
package com.itheima.service.impl;

import com.itheima.service.UserService;

/**
 * 包名:com.itheima.service.impl
 *
 * @author Leevi
 * 日期2020-06-27  16:04
 * 目标:
 * 目标1:在执行增删改查方法之前，都做权限校验    前置通知: before
 * 目标2:在执行增删改查方法之后(没有出现异常)，都打印日志    后置通知: after-returning
 * 目标3:在执行增删改查方法出现异常之后，都进行友好的通知    异常通知: after-throwing
 * 目标4:在执行增删改查方法之后，无论出没出现异常，均打印一句"执行完毕"  最终通知: after
 * 目标5:计算add、update方法的执行时间  环绕通知: around
 */
public class UserServiceImpl implements UserService {
    @Override
    public void add() throws InterruptedException {
        System.out.println("添加用户....");
        Thread.sleep(3000);
    }

    @Override
    public void delete() {
        System.out.println("删除用户....");
        int i = 10/0;
    }

    @Override
    public void update() throws InterruptedException {
        System.out.println("修改用户...");
        Thread.sleep(5000);
    }

    @Override
    public void findAll() {
        System.out.println("查询用户....");
    }

}
```

```
package com.itheima.aspect;

import org.aspectj.lang.ProceedingJoinPoint;
```

```java
/**
 * 包名:com.itheima.aspect
 *
 * @author Leevi
 * 日期2020-06-27  16:06
 */
public class MyAspect {
    /**
     * 权限校验
     */
    public void checkPermission(){
        System.out.println("进行权限校验...");
    }

    /**
     * 打印日志
     */
    public void printLog(){
        System.out.println("打印日志...");
    }

    /**
     * 友好通知
     */
    public void say(){
        System.out.println("服务器出现异常，请稍后...");
    }


    public void end(){
        System.out.println("执行完毕....");
    }

    /**
     * 计算执行时间
     * @param joinPoint
     * @throws Throwable
     */
    public void time(ProceedingJoinPoint joinPoint) throws Throwable {
        long startTime = System.currentTimeMillis();
        //执行被增强的那个方法
        joinPoint.proceed();
        long endTime = System.currentTimeMillis();
        System.out.println(endTime - startTime);
    }
}
```

- applicationContext.xml配置文件中进行aop配置

```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"
       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">
    <!--ioc的配置-->
    <bean id="userService" class="com.itheima.service.impl.UserServiceImpl">
</bean>
    <bean id="myAspect" class="com.itheima.aspect.MyAspect"></bean>
    <!--
        aop的配置:
            1. 指定切入点 (要增强谁)
            2. 指定切面以及切面里面的通知  （用什么来增强）
               通知又分为:前置、后置、最终、异常、环绕 (从哪增强)
    -->
    <aop:config>
        <!--1.切入点的配置-->
        <aop:pointcut id="pt1" expression="execution(*
com.itheima.service.impl.UserServiceImpl.*(..))"/>
        <aop:pointcut id="pt2" expression="execution(*
com.itheima.service.impl.UserServiceImpl.add(..))"/>
        <!--
            2.配置切面以及切面里面的通知
                ref:指定哪个对象作为切面
        -->
        <aop:aspect id="asp1" ref="myAspect">
            <!--前置通知-->
            <aop:before method="checkPermission" pointcut-ref="pt1">
</aop:before>
            <!--
                异常通知: after-throwing
            -->
            <aop:after-throwing method="say" pointcut-ref="pt1"></aop:after-
throwing>
            <!--
                最终通知: after
            -->
            <aop:after method="end" pointcut-ref="pt1"></aop:after>

            <!--
                后置通知 : after-returning
            -->
            <aop:after-returning method="printLog" pointcut-ref="pt1">
</aop:after-returning>

            <!--
                环绕通知:
            -->
            <aop:around method="time" pointcut-ref="pt2"></aop:around>
        </aop:aspect>
    </aop:config>
```

知识点-基于注解的AOP配置

1.目标

- [ ] 能够理解spring的AOP的注解

2.分析

2.1路径

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

2.2步骤

1. 创建工程, 导入坐标
2. 创建Dao和增强的类, 注册
3. 开启AOP的注解支持
4. 在增强类上面添加@Aspect
5. 在增强类的增强方法上面添加
    - @Before()
    - @AfterReturning()
    - @Around()
    - @AfterThrowing()
    - @After()

3.讲解

3.1前置通知

- 添加坐标

```
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.8.7</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
    </dependency>
```

```
</dependencies>
```

- 定义被增强的业务逻辑类

```
package com.itheima.service.impl;

import com.itheima.service.UserService;

/**
 * 包名:com.itheima.service.impl
 *
 * @author Leevi
 * 日期2020-06-27  16:04
 * 目标:
 * 目标1:在执行增删改查方法之前，都做权限校验    前置通知: before
 * 目标2:在执行增删改查方法之后(没有出现异常)，都打印日志    后置通知: after-returning
 * 目标3:在执行增删改查方法出现异常之后，都进行友好的通知    异常通知: after-throwing
 * 目标4:在执行增删改查方法之后，无论出没出现异常，均打印一句"执行完毕"  最终通知: after
 * 目标5:计算add、update方法的执行时间  环绕通知: around
 */
public class UserServiceImpl implements UserService {
    @Override
    public void add() throws InterruptedException {
        System.out.println("添加用户....");
        Thread.sleep(3000);
    }

    @Override
    public void delete() {
        System.out.println("删除用户....");
        int i = 10/0;
    }

    @Override
    public void update() throws InterruptedException {
        System.out.println("修改用户...");
        Thread.sleep(5000);
    }

    @Override
    public void findAll() {
        System.out.println("查询用户....");
    }

}
```

- 定义切面类,在切面增强类上面使用注解 @Aspect并且在切面类中定义切入点方法

```
package com.itheima.aspect;

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

/**
 * 包名:com.itheima.aspect
```

```
 *
 * @author Leevi
 * 日期2020-06-27  16:06
 *
 */
@Component
@Aspect
public class MyAspect {
    @Pointcut("execution(* com.itheima.service.impl.UserServiceImpl.*(..))")
    public void pt1(){

    }

    @Pointcut("execution(*
com.itheima.service.impl.UserServiceImpl.add(..))")
    public void pt2(){

    }
    /**
     * 权限校验
     */
    @Before("pt1()")
    public void checkPermission(){
        System.out.println("进行权限校验...");
    }

    /**
     * 打印日志
     */
    @AfterReturning("pt1()")
    public void printLog(){
        System.out.println("打印日志...");
    }

    /**
     * 友好通知
     */
    @AfterThrowing("pt1()")
    public void say(){
        System.out.println("服务器出现异常，请稍后...");
    }

    @After("pt1()")
    public void end(){
        System.out.println("执行完毕....");
    }

    /**
     * 计算执行时间
     * @param joinPoint
     * @throws Throwable
     */
    @Around("pt2()")
    public void time(ProceedingJoinPoint joinPoint) throws Throwable {
        long startTime = System.currentTimeMillis();
        //执行被增强的那个方法
        joinPoint.proceed();
        long endTime = System.currentTimeMillis();
```

```
        System.out.println(endTime - startTime);
    }
}
```

- 创建配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">
    <!--包扫描-->
    <context:component-scan base-package="com.itheima"/>
    <!--加载aop注解驱动-->
    <aop:aspectj-autoproxy />
</beans>
```

第二章-Spring中的JdbcTemplate 【了解】

知识点-JdbcTemplate基本使用

1.目标

- [ ] 能够使用JdbcTemplate

2.路径

1. JdbcTemplate介绍
2. JdbcTemplate入门
3. 使用JdbcTemplate完成CRUD

3.讲解

3.1JdbcTemplate介绍

Spring除了IOC 和 AOP之外，还对Dao层的提供了支持。 就算没有框架，也对原生的JDBC提供了支持。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。

持久化技术	模版类
JDBC	org.springframework.jdbc.core.JdbcTemplate
Hibrenate	org.springframework.orm.hibernate5.HibernateTemplate
MyBatis	org.mybatis.spring.SqlSessionTemplate

3.2JDBC模版入门案例

- 创建Maven工程,导入坐标

```
  <dependencies>
    <!--Spring核心容器-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>
    <!--SpringJdbc-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.6</version>
    </dependency>
    <!--Spring整合单元测试-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>
    <!--单元测试-->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

- 表结构创建和JavaBean

```
create table account(
    id int primary key auto_increment,
    name varchar(40),
    money double
)character set utf8 collate utf8_general_ci;

insert into account(name,money) values('zs',1000);
insert into account(name,money) values('ls',1000);
insert into account(name,money) values('ww',1000);
```

- 实体

```
public class Account implements Serializable{
    private Integer id;
    private String name;
    private Double money;
  ...
}
```

- 使用JDBC模版API

```
    //使用JDBC模版保存
    public void save(Account account) {
        //1. 创建数据源
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/spring_day03");
        dataSource.setUsername("root");
        dataSource.setPassword("123456");

        //2. 创建JDBC模版
        JdbcTemplate jdbcTemplate = new JdbcTemplate();
        jdbcTemplate.setDataSource(dataSource);

        //3.操作数据库
        String sql = "insert into account values(?,?,?)";
        jdbcTemplate.update(sql, null,account.getName(),account.getMoney())

    }
```

3.3.使用JDBC模版完成CRUD

1. 增加

```
String sql = "insert into account values(?,?,?)";
jdbcTemplate.update(sql, null,account.getName(),account.getMoney());
```

2. 修改

```
String sql ="update account set name = ? where id = ?";
Object[] objects = {user.getName(), user.getId()};
jdbcTemplate.update(sql,objects);
```

3. 删除

```
String sql = "delete from account where id=?";
jdbcTemplate.update(sql, 6);
```

4. 查询单个值

```
String sql = "select count(*) from t_user";
Long n = jdbcTemplate.queryForObject(sql, Long.class);
```

```
String sql = "select name from t_user where id=?";
String name = jdbcTemplate.queryForObject(sql, String.class, 4);
System.out.println(name);
```

5. 查询单个对象

```
String sql = "select * from account where id = ?";
User user = jdbcTemplate.queryForObject(sql,new BeanPropertyRowMapper<>
(Account.class),id);
```

6. 查询集合

```
String sql = "select * from account";
List<Map<String, Object>> list = jdbcTemplate.queryForList(sql);
System.out.println(list);
```

```
String sql = "select * from account";
List<Account> list = List<User> list = jdbcTemplate.query(sql, new
BeanPropertyRowMapper<>(Account.class));
```

4.小结

1. JdbcTemplate是Spring里面的持久层一个工具包, 封装了Jdbc
2. 使用步骤
    - 创建DataSource
    - 创建JDBCTemplate
    - 调用update(), query(), queryForObject()

知识点-在dao中使用JdbcTemplate

1.目标

- [ ] 能够使用spring的jdbc的模板

2.路径

1. 方式一在dao中定义JdbcTemplate
2. 方式二dao继承JdbcDaoSupport

3.讲解

3.1方式一

这种方式我们采取在dao中定义JdbcTemplate

- AccountDaoImpl.java

```
public class AccountDaoImpl implements AccountDao {

    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override
    public Account findAccountById(Integer id) {
        List<Account> list =  jdbcTemplate.query("select * from account where id
= ? ",new AccountRowMapper(),id);
        return list.isEmpty()?null:list.get(0);
    }
```

```
    @Override
    public Account findAccountByName(String name) {
        List<Account> list =  jdbcTemplate.query("select * from account where
name = ? ",new AccountRowMapper(),name);
        if(list.isEmpty()){
            return null;
        }
        if(list.size()>1){
            throw new RuntimeException("结果集不唯一，不是只有一个账户对象");
        }
        return list.get(0);
    }

    @Override
    public void updateAccount(Account account) {
        jdbcTemplate.update("update account set money = ? where id = ?
",account.getMoney(),account.getId());
    }

}
```

- applicationContext.xml

```
    <!-- 配置一个dao -->
    <bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl">
        <!-- 注入jdbcTemplate -->
        <property name="jdbcTemplate" ref="jdbcTemplate"></property>
    </bean>

    <!-- 配置一个数据库的操作模板：JdbcTemplate -->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!-- 配置数据源 -->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
        <property name="url" value="jdbc:mysql:///spring_day04"></property>
        <property name="username" value="root"></property>
        <property name="password" value="123456"></property>
    </bean>
```

3.2方式二

这种方式我们采取让dao继承JdbcDaoSupport

- AccountDaoImpl.java

```
public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {

    @Override
    public Account findAccountById(Integer id) {
        //getJdbcTemplate()方法是从父类上继承下来的。
        List<Account> list = getJdbcTemplate().query("select * from account
where id = ? ",new AccountRowMapper(),id);
```

```
        return list.isEmpty()?null:list.get(0);
    }

    @Override
    public Account findAccountByName(String name) {
        //getJdbcTemplate()方法是从父类上继承下来的。
        List<Account> list =  getJdbcTemplate().query("select * from account
where name = ? ",new AccountRowMapper(),name);
        if(list.isEmpty()){
            return null;
        }
        if(list.size()>1){
            throw new RuntimeException("结果集不唯一，不是只有一个账户对象");
        }
        return list.get(0);
    }

    @Override
    public void updateAccount(Account account) {
        //getJdbcTemplate()方法是从父类上继承下来的。
        getJdbcTemplate().update("update account set money = ? where id = ?
",account.getMoney(),account.getId());
    }
}
```

- 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">

<!-- 配置dao2 -->
<bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl">
    <!-- 注入dataSource -->
    <property name="dataSource" ref="dataSource"></property>
</bean>
<!-- 配置数据源 -->
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
    <property name="url" value="jdbc:mysql:///spring_day04"></property>
        <property name="username" value="root"></property>
        <property name="password" value="123456"></property>
    </bean>
</beans>
```

4.小结

1. 方式一: 自己注册JDBCTemplate, 自己在Dao注入JDBCTemplate
2. 方式二: 继承JdbcDaoSupport, 不需要自己注册JDBCTemplate, 也不需要自己注入JDBCTemplate
   需要注入DataSource

第三章-Spring配置第三方连接池

知识点-Spring配置第三方连接池

1.目标

- [ ] 能够配置spring的连接池

2.路径

1. Spring配置C3P0连接池
2. Spring配置Druid连接池
3. Spring配置HikariCP连接池【扩展】

3.讲解

3.1Spring配置C3P0连接池

官网:http://www.mchange.com/projects/c3p0/

步骤:

1. 导入c3p0的坐标
2. 注册连接池(修改DataSource的class以及四个基本项)

实现:

- 导入坐标

```
<dependency>
      <groupId>c3p0</groupId>
      <artifactId>c3p0</artifactId>
      <version>0.9.1.2</version>
</dependency>
```

- 配置数据源连接池

```
<!-- 配置C3P0数据库连接池 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
   <property name="jdbcUrl" value="jdbc:mysql:///spring_jdbc" />
   <property name="driverClass" value="com.mysql.jdbc.Driver" />
   <property name="user" value="root" />
   <property name="password" value="123456" />
</bean>
<!-- 管理JDBC模版 -->
<bean id="jdbcTemplete" class="org.springframework.jdbc.core.JdbcTemplate">
   <property name="dataSource" ref="dataSource" />
</bean>
```

> 核心类：`com.mchange.v2.c3p0.ComboPooledDataSource

3.2.Spring配置Druid连接池

官网:http://druid.io/

步骤:

1. 导入Druid坐标
2. 修改DataSource的class(四个基本项)

实现:

- 导入坐标

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.9</version>
</dependency>
```

- 配置数据源连接池

```
    <!--注册dataSource -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver">
</property>
        <property name="url" value="jdbc:mysql://localhost:3306/spring_day03">
</property>
        <property name="username" value="root"></property>
        <property name="password" value="123456"></property>
    </bean>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg name="dataSource" ref="dataSource"></constructor-arg>
    </bean>
```

3.3.Spring配置HikariCP连接池【扩展】

官网:https://github.com/brettwooldridge/HikariCP

步骤:

1. 添加HikariCP坐标
2. 修改DataSource的class(四个基本项)

实现:

- 导入HikariCP坐标

```
<dependency>
   <groupId>com.zaxxer</groupId>
   <artifactId>HikariCP</artifactId>
   <version>3.1.0</version>
</dependency>
```

- 配置数据源连接池

```
 <!-- 配置HikariCP数据库连接池 -->
<bean id="dataSource" class="com.zaxxer.hikari.HikariDataSource">
    <property name="jdbcUrl" value="jdbc:mysql:///spring_jdbc" />
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="username" value="root" />
    <property name="password" value="123456" />
</bean>

 <!-- 管理JDBC模版 -->
<bean id="jdbcTemplete" class="org.springframework.jdbc.core.JdbcTemplate">
<property name="dataSource" ref="dataSource" />
</bean>
```

```
  >  核心类：`com.zaxxer.hikari.HikariDataSource`


+ 为什么HikariCP为什么越来越火?
  + 效率高
  + 一直在维护
  + ==SpringBoot2.0 现在已经把HikariCP作为默认的连接池了==


+ 为什么HikariCP被号称为性能最好的Java数据库连接池?

  + 优化并精简字节码: HikariCP利用了一个第三方的Java字节码修改类库Javassist来生成委托实现动态代理

  + 优化代理和拦截器：减少代码，代码量越少。一般意味着运行效率越高、发生bug的可能性越低.




    ![1540557295765](img/1540557295765.png)

  + 自定义集合类型ConcurrentBag：ConcurrentBag的实现借鉴于C#中的同名类，是一个专门为连接池设计的lock-less集合，实现了比LinkedBlockingQueue、LinkedTransferQueue更好的并发性能。ConcurrentBag内部同时使用了ThreadLocal和CopyOnWriteArrayList来存储元素，其中CopyOnWriteArrayList是线程共享的。ConcurrentBag采用了queue-stealing的机制获取元素：首先尝试从ThreadLocal中获取属于当前线程的元素来避免锁竞争，如果没有可用元素则再次从共享的CopyOnWriteArrayList中获取。此外，ThreadLocal和CopyOnWriteArrayList在ConcurrentBag中都是成员变量，线程间不共享，避免了伪共享(false sharing)的发生。

  + 使用FastList替代ArrayList: FastList是一个List接口的精简实现，只实现了接口中必要的几个方法。JDK ArrayList每次调用get()方法时都会进行rangeCheck检查索引是否越界，FastList的实现中去除了这一检查，只要保证索引合法那么rangeCheck就成为了不必要的计算开销(当然开销极小)。




### 4.小结
```

```
1. 导入连接池坐标
2. 改DataSource的class属性值，配置属性名



## 知识点-Spring引入Properties配置文件

### 1.目标

- [ ] 掌握Spring引入Properties配置文件

### 2.步骤

1. 定义jdbc.properties
2. 把jdbc.properties引入applicationContext.xml
3. 使用表达式根据key获得value



### 3.实现

#### 3.1jdbc.properties

```properties
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/spring_day03
jdbc.username=root
jdbc.password=123456
```
```

3.2.在applicationContext.xml页面使用

- 引入配置文件方式一(不推荐使用)

```
 <!-- 引入properties配置文件：方式一 (繁琐不推荐使用) -->
<bean id="properties"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="classpath:jdbc.properties" />
</bean>
```

- 引入配置文件方式二【推荐使用】

```
 <!-- 引入properties配置文件：方式二 (简单推荐使用) -->
<context:property-placeholder location="classpath:jdbc.properties" />
```

classpath表示的是类路径，对于配置文件而言也就是那个resources根目录

- bean标签中引用配置文件内容

```
 <!-- hikariCP 连接池 -->
<bean id="dataSource" class="com.zaxxer.hikari.HikariDataSource">
    <property name="jdbcUrl" value="${jdbc.url}" />
    <property name="driverClassName" value="${jdbc.driver}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>
```

> 注意:在使用`<context:property-placeholder/>`标签时，properties配置文件中的key一定要是带点分隔的。例如`jdbc.url`

4.小结

1. 定义jdbc.properties
2. 把jdbc.properties引入到applicationContext.xml
3. 根据key读取value

第四章-Spring管理事务

知识点-Spring管理事务概述(了解即可)

1.目标

- [ ] 知道Spring管理事务相关的API

2.路径

1. 概述
2. 相关的API

3.讲解

3.1概述

由于Spring对持久层的很多框架都有支持， Hibernate 、 jdbc 、JdbcTemplate , MyBatis ， 由于使用的框架不同，所以使用事务管理操作API 也不尽相同。 为了规范这些操作， Spring统一定义一个事务的规范 ， 这其实是一个接口 。这个接口的名称： PlatformTrasactionManager.并且它对已经做好的框架都有支持.

如果dao层使用的是JDBC，JdbcTemplate 或者mybatis,那么 可以使用 DataSourceTransactionManager 来处理事务

如果dao层使用的是 Hibernate, 那么可以使用HibernateTransactionManager 来处理事务

3.2相关的API

3.2.1PlatformTransactionManager

平台事务管理器是一个接口，实现类就是Spring真正管理事务的对象。

常用的实现类:

DataSourceTransactionManager :JDBC开发的时候(JDBCTemplate,MyBatis)，使用事务管理。

HibernateTransactionManager :Hibernate开发的时候，使用事务管理。

3.2.2TransactionDefinition

事务定义信息中定义了事务的隔离级别，传播行为，超时信息，只读。

3.2.3 TransactionStatus:事务的状态信息

事务状态信息中定义事务是否是新事务，是否有保存点。

4.小结

1. Spring管理事务是通过事务管理器. 定义了一个接口PlatformTransactionManager
    - **DataSourceTransactionManager** JDBC,MyBatis,JDBCTemplate
    - HibernateTransactionManager Hibernate

知识点-编程式(通过代码)(硬编码)事务【了解】

1.目标

- [ ] 了解Spring编程式事务

2.步骤

1. 创建DataSource对象
2. 创建JDBCTemplate对象
3. 创建事务管理器, 传入DataSource. DataSourceTransactionManager
4. 创建事务模版
5. 通过事务模版进行事务处理
6. 在事务里面操作数据库

3.实现

- 添加依赖

```
<dependencies>
  <!--Spring核心容器-->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.2.RELEASE</version>
  </dependency>
  <!--SpringJdbc-->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.0.2.RELEASE</version>
  </dependency>
  <!--事务相关的-->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.0.2.RELEASE</version>
  </dependency>
  <!--SpringAOP相关的坐标-->
  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.7</version>
  </dependency>
<!--数据库驱动-->
  <dependency>
```

```
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.6</version>
    </dependency>
    <!--Spring整合单元测试-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-test</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>
    <!--单元测试-->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
     <!--连接池-->
    <dependency>
      <groupId>com.zaxxer</groupId>
      <artifactId>HikariCP</artifactId>
      <version>3.1.0</version>
    </dependency>
  </dependencies>
```

- 代码

```
public class JDBCTempleTest {
     @Test
     public void fun01() throws PropertyVetoException{
            //1. 创建数据源
             ComboPooledDataSource dataSource = new ComboPooledDataSource();
             dataSource.setDriverClass("com.mysql.jdbc.Driver");
             dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/spring_day03");
             dataSource.setUser("root");
             dataSource.setPassword("123456");
             //2. 创建jdbc模版
             final JdbcTemplate jdbcTemplate = new JdbcTemplate();
             jdbcTemplate.setDataSource(dataSource);

             //开始编程式事务
             //3.创建事务管理器
             DataSourceTransactionManager transactionManager = new
DataSourceTransactionManager();
             transactionManager.setDataSource(dataSource);
             //4.创建事务模版
             TransactionTemplate transactionTemplate = new
TransactionTemplate();

             //给事务模板配置事务管理者
             transactionTemplate.setTransactionManager(transactionManager);

                //4. 进行事务操作
             transactionTemplate.execute(new TransactionCallback<Object>() {
                @Override
                public Object doInTransaction(TransactionStatus status) {
                    //操作数据库
```

```
                    jdbcTemplate.update("update t_account set money = money - ?
where name = ?  ",  100.0,"zs");

                    int i = 1/0;

                    jdbcTemplate.update("update t_account set money = money + ?
where name = ?  ", 100.0,"ls");
                    return null;
                }
            });
        }
}
```

4.小结

1. 我们项目开发 一般不会使用编程式(硬编码)方式. 一般使用声明式(配置)事务
    - xml方式
    - 注解方式

知识点-Spring声明式事务-xml配置方式【重点】

Spring的声明式事务的作用是: 无论spring集成什么dao框架，事务代码都不需要我们再编写了

声明式的事务管理的思想就是AOP的思想。面向切面的方式完成事务的管理。声明式事务有两种,==xml配置方式和注解方式.==

1.目标

- [ ] 使用xml方式配置事务

2.步骤

1. 注册事务管理器, 配置数据源
2. 配置事务建议(事务规则: 事务隔离级别, 遇到什么异常回滚,是否只读...)
3. 配置AOP
    - 配置切入点
    - 配置切面

3.讲解

3.1实现

```
<dependencies>
    <!--SpringJdbc-->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>5.0.2.RELEASE</version>
    </dependency>
```

```
    <!--SpringAOP相关的坐标-->
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjweaver</artifactId>
      <version>1.8.7</version>
    </dependency>
  </dependencies>
```

- 配置事务管理器

```
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"></property>
</bean>
```

- 配置事务建议(事务的规则)

```
<tx:advice id="adviceId" transaction-manager="transactionManager">
  <!--在tx:advice标签内部 配置事务的属性 -->
  <tx:attributes>
  <!-- 指定方法名称：是业务核心方法
       read-only：是否是只读事务。默认false，不只读。
       isolation：指定事务的隔离级别。默认值是使用数据库的默认隔离级别。
       propagation：指定事务的传播行为。
       timeout：指定超时时间。默认值为：-1。永不超时。
       rollback-for：用于指定一个异常，当执行产生该异常时，事务回滚。产生其他异常，事务不回滚。没有默认值，任何异常都回滚。
       no-rollback-for：用于指定一个异常，当产生该异常时，事务不回滚，产生其他异常时，事务回滚。没有默认值，任何异常都回滚。
       -->
       <tx:method name="*" read-only="false" propagation="REQUIRED"/>
       <tx:method name="find*" read-only="true" propagation="SUPPORTS"/>
  </tx:attributes>
</tx:advice>
```

- 配置事务的AOP

```
<aop:config>
    <!-- 定义切入点 ，也就是到底给那些类中的那些方法加入事务的管理 -->
    <aop:pointcut expression="execution(* com.itheima.service.impl.*.*(..))"
id="pointCat"/>
    <!-- 让切入点和事务的建议绑定到一起 -->
    <aop:advisor advice-ref="txAdvice" pointcut-ref="pointCat"/>
</aop:config>
```

3.2事务的传播行为

3.2.1事务的传播行为的作用

我们一般都是将事务设置在Service层,那么当我们调用Service层的一个方法的时候, 它能够保证我们的这个方法中,执行的所有的对数据库的更新操作保持在一个事务中， 在事务层里面调用的这些方法要么全部成功，要么全部失败。

如果你的Service层的这个方法中，除了调用了Dao层的方法之外， 还调用了本类的其他的Service方法，那么在调用其他的Service方法的时候， 我必须保证两个service处在同一个事务中，确保事物的一致性。

事务的传播特性就是解决这个问题的。

3.2.2 事务的传播行为的取值

保证在同一个事务里面:

- **PROPAGATION_REQUIRED:默认值，也是最常用的场景.**

  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。

- PROPAGATION_SUPPORTS：

  如果当前没有事务，就以非事务方式执行。

  如果已经存在一个事务中，加入到这个事务中。

- PROPAGATION_MANDATORY

  如果当前没有有事务，就抛出异常;

  如果已经存在一个事务中，加入到这个事务中。

保证不在同一个事物里:

- **PROPAGATION_REQUIRES_NEW**

  如果当前有事务，把当前事务挂起,创建新的事务但独自执行

- PROPAGATION_NOT_SUPPORTED

  如果当前存在事务，就把当前事务挂起。不创建事务

- PROPAGATION_NEVER

  如果当前存在事务，抛出异常

4.小结

1. 注册事务管理器, 需要注入DataSource
2. 配置事务建议(规则: 隔离级别, 事务传播行为...)
3. 配置AOP
   - 配置切入点
   - 配置切面

知识点-spring声明式事务-注解方式【重点】

1.目标

- [ ] 使用注解方式配置事务

2.步骤

1. 注册事务管理器
2. 开启事务注解支持
3. 在业务类上面添加@Transactional

3.实现

- 在applicationContext里面打开注解驱动

```
<!--一,配置事务管理器  -->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

<!--二配置事务注解  -->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

- 在业务逻辑类上面使用注解

  @Transactional： 如果在类上声明，那么标记着该类中的所有方法都使用事务管理。也可以作用于方法上， 那么这就表示只有具体的方法才会使用事务。

4.小结

4.1步骤

1. 注册事务管理器
2. 开启事务注解支持
3. 在业务类上面添加@Transactional

4.2两种方式比较

- xml方式
  - 优点: 只需要配一次, 全局生效, 容易维护
  - 缺点: 相对注解而言 要麻烦一点
- 注解方式
  - 优点: 配置简单
  - 缺点: 写一个业务类就配置一次, 事务细节不好处理, 不好维护