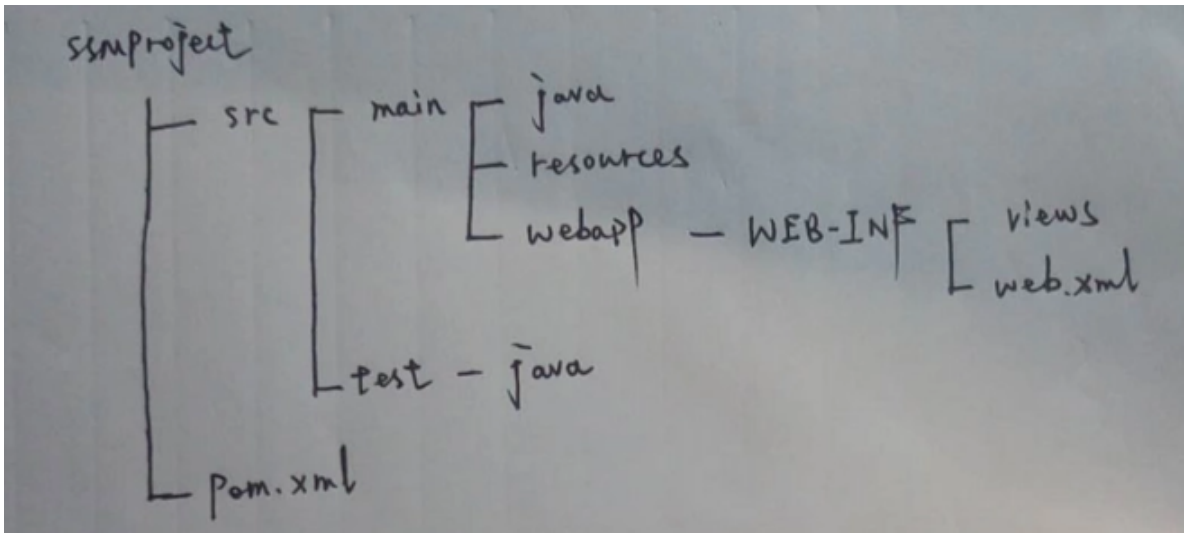
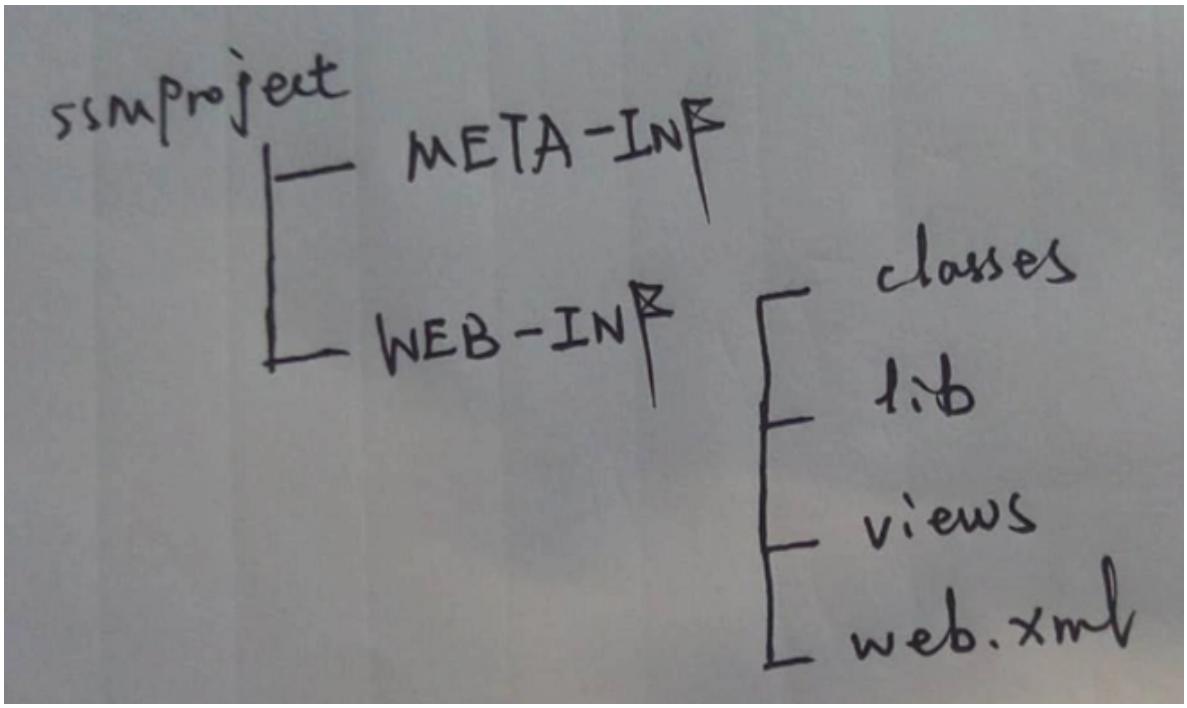
`java`项目中的`classPath`到底是什么

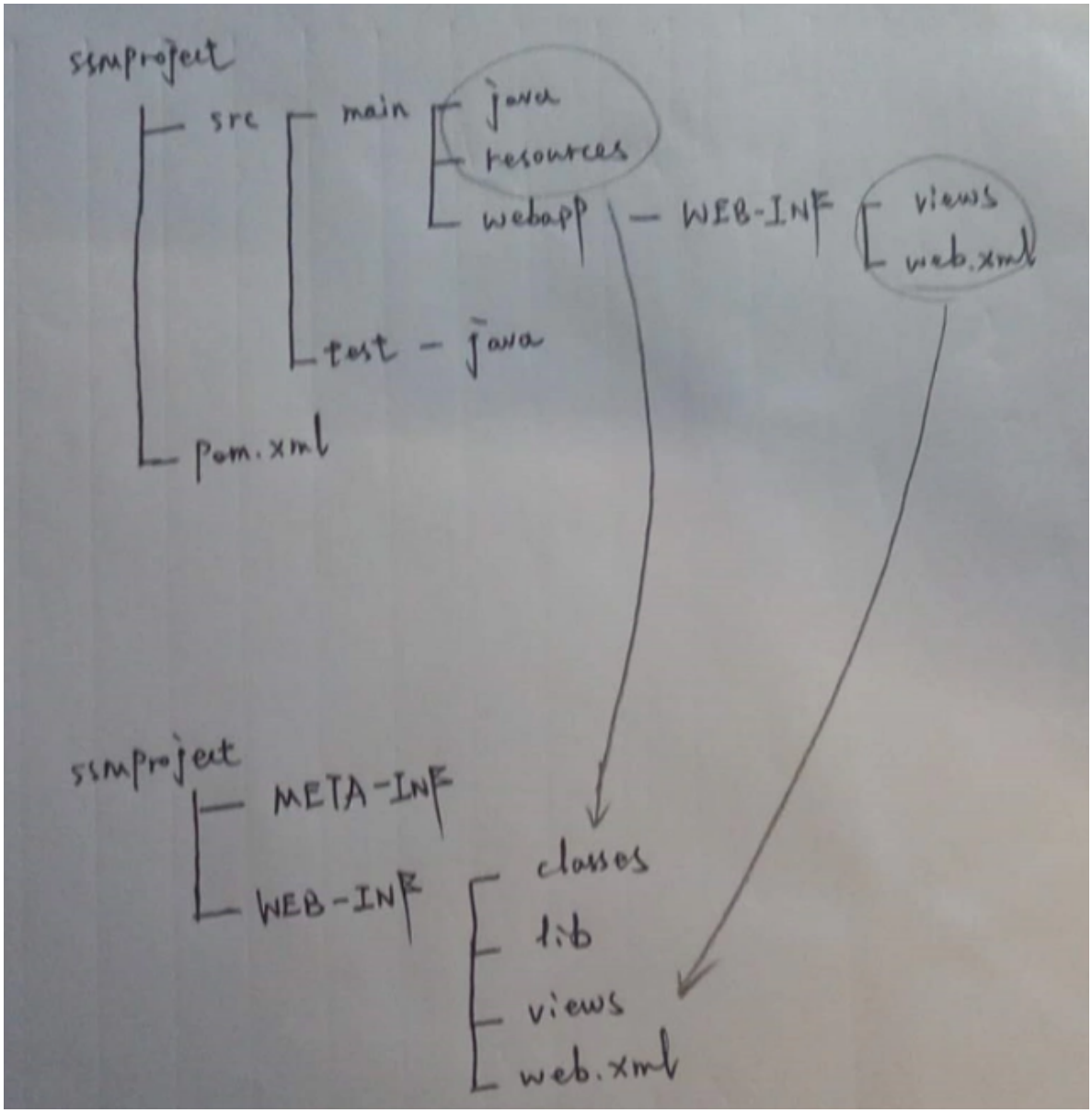
- 开发时期的`web`项目结构



- `web`项目发布后的目录结构



- 对比资源

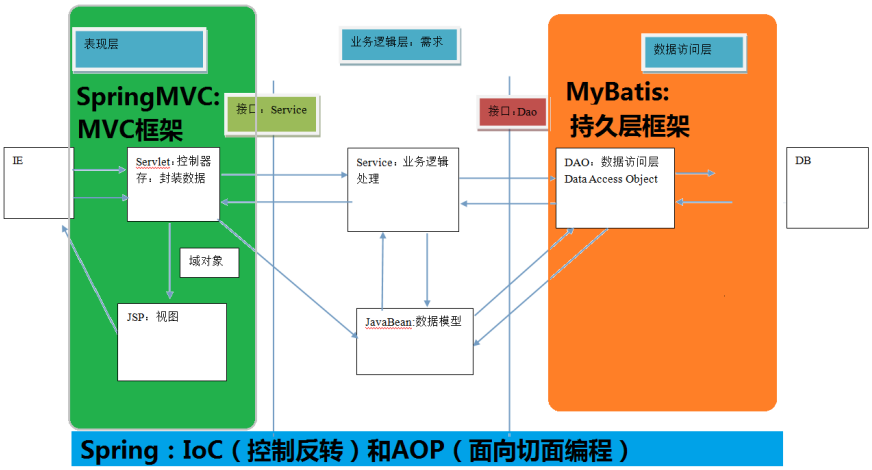


> - 开发时期的 `java` 和 `resource` ——>生产包时期： `classes`
> - 开发时期的 `views` 和 `web.xml` 仍然在 `WEB-INF` 下面

> 在打包后的项目中，根目录有两个：
>
> - `META-INF`
> - `WEB-INF`

> 我们常说的 `classes` 就是 `classpath`

Spring 讲解内容



- `spring` 框架的概述以及 `spring` 中基于 `xml` 的 `IOC` 配置
- `spring` 中基于注解的 `IOC` 和 `IOC` 的案例
- `Spring` 中的 `aop` 和基于 `XML` 以及注解的 `AOP` 配置
- `Spring` 中的 `JdbcTemplate` 以及 `Spring` 事务控制

第一天

第一天内容介绍

1. `Spring` 概述
    1. `Spring` 是什么
    2. `Spring` 的两大核心
    3. `Spring` 的发展历程和优思
    4. `Spring` 体系结构
2. 程序的耦合及解耦
    1. 曾经案例中的问题
    2. 工厂模式解耦
3. `IOC` 概念和 `Spring` 中的 `IOC`
    1. `Spring` 中基于 `ZML` 的 `IOC` 环境搭建
4. 依赖注入 `（Dependency Injection）`
5. 作业：

`Spring` 概述

> - `Spring` 是分层的 Java SE/EE 应用 `full-stack` （全栈）轻量级开源框架

- 以 IOC （Inverse Of Control： 反转控制）和 AOP （Aspect Oriented Programming：面向切面编程）为 内核 ，
- 提供了 展现层 Spring MVC 和 持久层 Spring JDBC 以及业务层事务管理等众多的企业级应用技术，还能整合开源世界众多 著名的第三方框架和类库，逐渐成为使用最多的 Java EE 企业应用开源框架。
- 官网： https://spring.io

Spring 的优势

- **方便解耦，简化开发**

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

- **AOP 编程的支持**

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

- **声明式事务的支持**

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

- **方便程序的测试**

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

- **方便集成各种优秀框架**

  Spring 可以降低各种框架的使用难度，提供了对各种优秀框架 （Struts、Hibernate、Hessian、Quartz

  等） 的直接支持。

- **降低 JavaEE API 的使用难度**

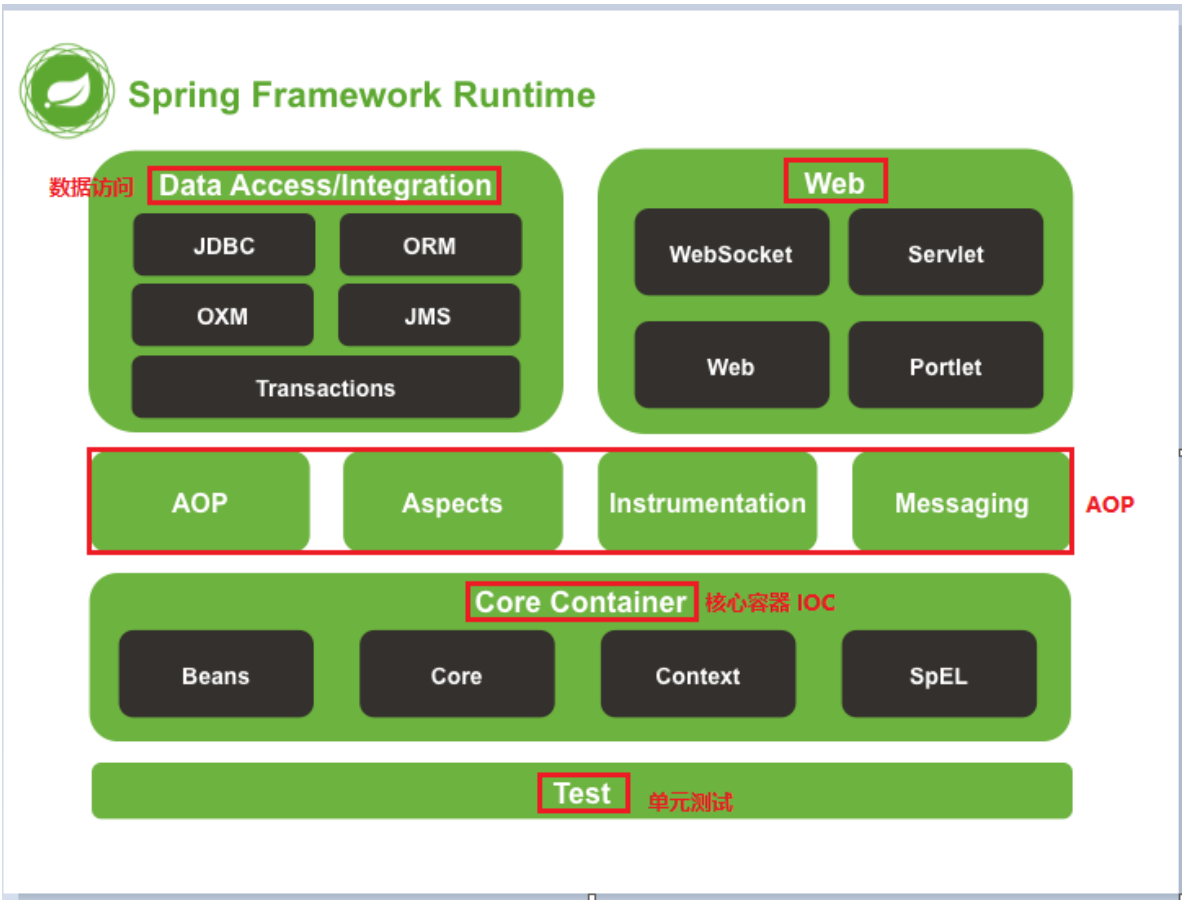
  Spring 对 JavaEE API（如 JDBC、JavaMail、远程调用等 进行了薄薄的封装层，使这些 API 的

  使用难度大为降低。

- **Java 源码是经典学习范例**

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

`Spring`的体系结构



耦合和解耦合的思路分析

- 程序的耦合
  - 耦合：程序之间的依赖关系
    - 类之间的依赖
    - 程序之间的依赖
  - 解耦合
    - 降低程序间的依赖关系
- 实际开发中：
  - 应该做到：编译器不依赖，运行时才依赖

- 解耦的思路：
  1. 使用反射来创建对象，而避免使用`new`关键字

2. 通过读取配置文件来获取要创建对象的全限定类名

问题分析

> 现在编写一个模拟保存用户的操作来讲解一下问题

> 持久层
>
> - 当持久层被业务层调用的时候会出现类的耦合

- `IAccountDao.interface`

```
package com.bean.dao;

/**
 * 持久层接口
 */
public interface IAccountDao {

    void saveAccount();
}
```

- `IAccountDaoImpl.class`

```
 package com.bean.dao.impl;

 import com.bean.dao.IAccountDao;

 /**
  * 持久层实现类，这个类将会在业务层调用持久层的时候造成类的耦合
  */
 public class AccountDaoImpl implements IAccountDao {
 //    模拟保存操作
     public void saveAccount() {
         System.out.println("模拟已经保存了账户");
     }
 }
```

> 业务层
>
> - 当业务层调用持久层的时候会出现类的耦合
> - 当业务层被表现层调用的时候会出现类的耦合

- `IAccountService.interface`

```java
package com.bean.service;

/**
 * 业务层接口
 */
public interface IAccountService {
    /**
     * 模拟保存账户
     */
    void saveAccount();
}
```

- `IAccountServiceImpl.class`

```java
package com.bean.service.impl;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.service.IAccountService;

/**
 * 业务层实现类，用于调用持久层，当业务层调用持久层的时候就会产生类的耦合
 * 当业务层被表现层进行调用的时候就会出现类的耦合
 */
public class AccountServiceImpl implements IAccountService {
//    注意这里，new 了一个新的AccountDaoImpl对象，这正是类的耦合，要实现解耦合必须要解决这个问题
    private IAccountDao accountDao = new AccountDaoImpl();

    /**
     * 进行保存操作
     */
    public void saveAccount() {
        accountDao.saveAccount();
    }
}
```

> 表现层
>
> - 当表现层调用业务层的时候会出现类的耦合

- `cline.class`

```java
package com.bean.ui;

import com.bean.service.IAccountService;
import com.bean.service.impl.AccountServiceImpl;

/**
 * 这里是表现层，用于调用业务层，实际开发中这里应该是servlet，但是这只是模拟
 */
public class cline {
    public static void main(String[] args) {
//        注意这里，这里也使用了new关键字，也就是当表现层调用业务层的时候也有类的耦合
        IAccountService accountService = new AccountServiceImpl();
        accountService.saveAccount();
    }
```

```
15      }
```

编写工厂类和配置文件

工厂

工厂模式创建可重用组件

> 我们需要解耦合就需要工厂模式

- `Bean`：在计算机英语中，`Bean`有可重用组件的含义
  - `JavaBean`：用`JAVA`语言编写的可重用组件

> 所以，前面我们一直以为 JavaBean == 实体类，但是其实是`JavaBean >> 实体类`，`JavaBean`是可重用组件的一部分

> 知道了`Bean`对象的含义，我们就要使用工厂模式来写一个用来创建Bean对象的工厂，主要用于创建这里的可重用组件：
>
> `service`组件和`dao`对象

如何创建可重用组件

- 需求
  1. 需要一个配置文件来配置我们的`service`和`dao`
     - 配置的内容：`唯一标志 = 全限定类名`，`(key = value)`的形式
  2. 通过读取配置文件中的内容，反射创建对象
     1. 配置文件的内容：`xml`或者`properties`
- `bean.properties`

```
1   # 在resouce下面的bean.properties
2   accountService=com.bean.service.impl.AccountServiceImpl
3   accountDao=com.bean.dao.impl.AccountDaoImpl
```

创建并使用工厂

1. 首先是我们刚才创建的`properties`
2. 然后是我们的`Bean`工厂
   1. 定义`properties`对象
   2. 使用静态代码块为`Properties`赋值
      1. 实例化对象
      2. 获取`properties`文件的流对象
   3. 根据名称获取对象
3. 更改业务层和展示层的代码

1. 创建 `Properties`

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

2. 实现 `Bean` 工厂

```
package com.bean.factory;

import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

/**
 * 使用工厂模式创建Bean对象，以用来创建复用组件service和dao
 */
public class BeanFactory {

    //创建properties
    private static Properties properties;
    //使用静态代码块为properties赋值
    static{
        try {
            //实例化对象
            properties = new Properties();
            //获取properties文件的流对象
            InputStream inputStream =
BeanFactory.class.getClassLoader().getResourceAsStream("bean.properties");
            properties.load(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 根据名称获取对象
     * @param beanName  类型名称
     * @return 对象实体类
     */
    public static Object getBean(String beanName){
        Object bean = null;
        try {
            //根据key获取value
            String beanPath = properties.getProperty(beanName);
            //根据全限定类名来获取Class对象，然后进行实例化
            bean = Class.forName(beanPath).newInstance();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return bean;
    }
}
```

3. 更改业务层和展示层的代码

- 业务层

```
package com.bean.service.impl;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.factory.BeanFactory;
import com.bean.service.IAccountService;

/**
 * 业务层实现类，用于调用持久层，当业务层调用持久层的时候就会产生类的耦合
 * 当业务层被表现层进行调用的时候就会出现类的耦合
 */
public class AccountServiceImpl implements IAccountService {

//    private IAccountDao accountDao = new AccountDaoImpl();
    /*进行解耦合操作*/
    private IAccountDao accountDao = (IAccountDao) BeanFactory.getBean("accountDao");

    public void saveAccount() {
        accountDao.saveAccount();
    }
}
```

- 表现层

```
package com.bean.ui;

import com.bean.factory.BeanFactory;
import com.bean.service.IAccountService;
import com.bean.service.impl.AccountServiceImpl;

/**
 * 这里是表现层，用于调用业务层，实际开发中这里应该是servlet，但是这只是模拟
 */
public class cline {
    public static void main(String[] args) {
//        IAccountService accountService = new AccountServiceImpl();
        IAccountService accountService = (IAccountService)
BeanFactory.getBean("accountService");
        accountService.saveAccount();
    }
}
```

分析工厂模式中的问题并改造

- 现在的工厂模式是有问题的：

> - 多例：工厂每次启动都是创建的不同的实例，那么在调用的时候实例就都不相同
>
> > 多例会出现问题：
> >
> > - 大幅度消耗内存资源

- 我们要把工厂模式给改为单例工厂，就是只让工厂只创建一个实例，而且每次调用的都是同一个实例
- 我们的 `servlet` 就是单例的
    - 因为单例的时候只有一个实例，所以当我们将属性值定义在方法外部，类的内部的时候就会出现线程问题
    - 这就是之前讲 `servlet` 的时候讲的：尽量不要定义 `servlet` 的类成员
    - 多例对象虽然没有这个问题，但是由于对象被创建多次，执行效率肯定不高
    - 但是单例对象如果把变量定义在方法内部，也会进行重新初始化，这么看来，多例对象没有存在的必要

- 为了解决这个问题，我们对代码进行仔细的分析

```
    public static Object getBean(String beanName){
        Object bean = null;
        try {
            String beanPath = properties.getProperty(beanName);
            bean = Class.forName(beanPath).newInstance();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return bean;
    }
```

- 其中有一段：`Class.forName(beanPath).newInstance()`
- 这里说明了每次调用方法的时候都会根据默认构造函数进行重新初始化，那么我们只要初始化一次
- 要初始化一次，就要想办法把初始化的内容存起来，要不然会重新初始化，所以需要一个容器存起来
- 使用 `Map`，将初始化的对象储存起来

解决问题之后的单例工厂

```
package com.bean.factory;

import java.io.IOException;
import java.io.InputStream;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

/**
 * 使用工厂模式创建Bean对象，以用来创建复用组件service和dao
 */
public class BeanFactory {

```

```
15      private static Properties properties;
16
17      private static Map<String,Object> beans;
18
19      static{
20          try {
21              //实例化对象
22              properties = new Properties();
23              //获取properties文件的流对象
24              InputStream inputStream =
    BeanFactory.class.getClassLoader().getResourceAsStream("bean.properties");
25              properties.load(inputStream);
26
27              //初始化容器
28              beans = new HashMap<String, Object>();
29              //获取配置文件中的所有key属性，封装成一个枚举类型
30              Enumeration<Object> enumeration = properties.keys();
31              while (enumeration.hasMoreElements()){
32                  //获取key
33                  String key = enumeration.nextElement().toString();
34                  //获取value,这里是获取全类名
35                  String value = properties.getProperty(key);
36                  //根据全类名进行反射创建对象
37                  Object object = Class.forName(value).newInstance();
38                  beans.put(key,object);
39              }
40          } catch (IOException e) {
41              e.printStackTrace();
42          } catch (IllegalAccessException e) {
43              e.printStackTrace();
44          } catch (InstantiationException e) {
45              e.printStackTrace();
46          } catch (ClassNotFoundException e) {
47              e.printStackTrace();
48          }
49      }
50
51      //这里就改造成为直接返回容器里的内容了，成功改造成了多例
52      public static Object getBean(String beanName){
53          return beans.get(beanName);
54      }
55  }
```

IOC

- 那么 `IOC` 究竟是什么

> - `IOC`：控制反转，把创建对象的权力交给工厂，是框架的重要特征
> - `IOC` 包括：依赖注入，依赖查找

> 举个例子，我们要买房子，以前我们决定要找哪个房子，有绝对的控制权，现在交给中介了，这就是把控制权交出去，这就是控制反转

- `IOC` 的作用

> 只能削减计算机的耦合，解除我们代码中的依赖关系，别的都干不了

> 注意，耦合不可能完全消除，只能尽量削减

使用 `Spring` 的 `IOC` 解决程序耦合

准备 `Spring` 的开发包

- 官网： https://spring.io/

`Spring` 基于 `XML` 的开发环境

1. 编写 `pom.xml`
2. 使用 `Spring` 实现 `Bean` 工厂
3. 获取 `IOC` 核心容器并使用
    1. 获取 `IOC` 核心容器 `ApplicationContext` 的，而它有实现类：
        1. 使用实现类 `ClassPathXmlApplicationContext`
        2. 使用实现类 `FileSystemXmlApplicationContext`
        3. 使用实现类 `AnnotationConfigApplicationContext`
    2. 根据 `id` 获取 `Bean` 对象并使用

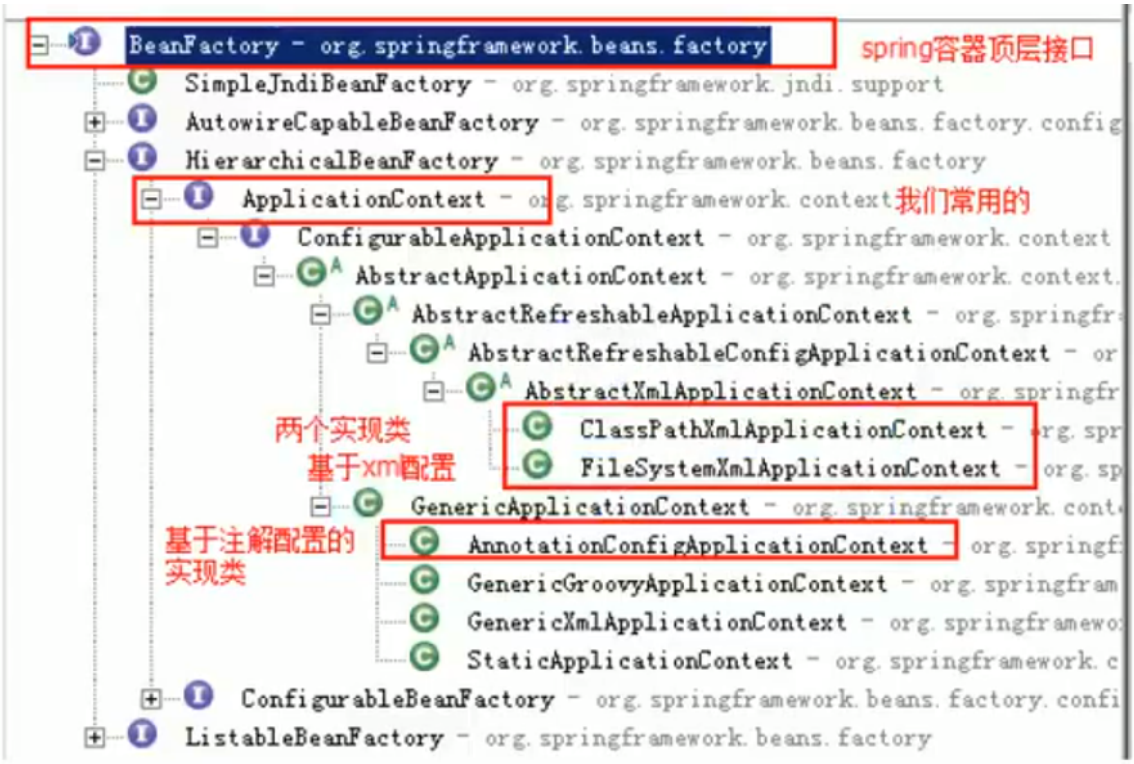
- `pom.xml`

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

    <groupId>com.bean</groupId>
    <artifactId>SpringIOCProject</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

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

</project>
```

- `bean.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">

    <!--把对象的创建交给spring来管理
        id：配置文件中的key值，与在讲工厂模式的时候的key值相同
        class：配置文件中的value值，是全限定类名，与在讲工厂模式的时候的value值相同
    -->
    <bean id="accountService" class="com.bean.service.impl.AccountServiceImpl"></bean>
    <bean id="accountDao" class="com.bean.dao.impl.AccountDaoImpl"></bean>

</beans>
```

- 首先来一张`ApplicationContext`的体系结构图



- `实现类`

```java
package com.bean.ui;

import com.bean.dao.IAccountDao;
import com.bean.service.IAccountService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 这里是表现层，用于调用业务层，实际开发中这里应该是servlet，但是这只是模拟
 */
public class cline {

    /**
     * 获取spring的IOC核心容器
```

```
     * @param args
     */
    public static void main(String[] args) {
        /*1. 获取核心容器，就是在工厂模式讲到的最后取到的容器Map对象，这里叫做ApplicationContext
        * ApplicationContext是个抽象类，有两个实现类：我们先用第一个
        *       1. ClassPathXmlApplicationContext
        *       2. FileSystemXmlApplicationContext
        * */
        ApplicationContext applicationContext = new ClassPathXmlApplicationContext("bean.xml");

        /*根据id获取bean对象，可以自己强转也可以传类型让他强转
        * */
        IAccountService accountService = (IAccountService) applicationContext.getBean("accountService");
        IAccountDao accountDao = applicationContext.getBean("accountDao", IAccountDao.class);


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

    }
}
```

`ApplicationContext` 三个实现类的区别

- `ClassPathXmlApplicationContext` 和 `FileSystemXmlApplication` 和 `AnnotationConfigApplicationContext`

1. `ClassPathXmlApplicationContext`：只可以加载类路径下的配置文件（实际项目更常用，但不如注解）
2. `FileSystemXmlApplicationContext`：可以加载磁盘任意路径下的文件(必须要有访问权限)
3. `AnnotationConfigApplicationCntext`：读取注解创建容器的，明天内容

```
package com.bean.ui;

import com.bean.dao.IAccountDao;
import com.bean.service.IAccountService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

/**
 * 这里是表现层，用于调用业务层，实际开发中这里应该是servlet，但是这只是模拟
 */
public class cline {

    /**
     * 获取spring的IOC核心容器
     * @param args
     */
    public static void main(String[] args) {
            //获取核心容器的方式1
//        ApplicationContext applicationContext = new ClassPathXmlApplicationContext("bean.xml");
```

```
21              //获取核心容器的方式2
22          ApplicationContext applicationContext  = new FileSystemXmlApplicationContext(
23   "E:\\Project\\java\\java\\JavaEE\\Frame\\Spring\\SpringIOCProject\\src\\main\\resources\\bean.xml");
24
25          IAccountService accountService = (IAccountService) applicationContext.getBean("accountService");
26          IAccountDao accountDao = applicationContext.getBean("accountDao", IAccountDao.class);
27
28
29          System.out.println(accountService);
30          System.out.println(accountDao);
31
32      }
33  }
```

使用两个构建核心容器的接口时出现的问题

构建核心容器的的两个接口：

- `ApplicationContext`

  > 在构建核心容器的时候，创建对象所采取的策略是立即加载。
  >
  > 也就是说，只要一读取完配置文件就立马创建配置文件中配置的对象

- `BeanFactory`

  > 在构建核心容器时，创建对象采取的的策略是延迟加载的方式。
  >
  > 也就是说，什么时候根据 `id` 创建对象了，什么时候才真正的创建对象

- 什么时候适合立即加载，什么时候适合延迟加载
- `ApplicationContext`

  > 单例模式适用，因为单例模式只创建一次，所以越早创建越好

- `BeanFactory`

  > 多例模式适用，因为多例模式多次创建，所以越晚创建越好

> 虽然上面两种分析的很到位，但是 `Spring` 是绝对强大的框架，他会根据你的配置来进行更改，那么从这方面分析，推荐使用 `ApplcationContext` ，因为 `BeanFactory` 是顶层，功能没那么强

`Spring` 中的 `bean` 细节

- `spring`
  - `id` ：唯一标识

- `class`：映射对应的类
- `factory-bean`：指定工厂
- `factory-method`：指定工厂中的方法
- `init-method`：初始化要执行的方法
- `destory-method`：销毁时要执行的方法
- `scope`：作用范围
  - `singleton`：单例
  - `prototype`：多例
  - `request`：作用于`web`应用的请求范围
  - `session`：作用于`web`应用的会话范围
  - `global-session`：作用于集群环境的会话范围（全局会话范围），当不是集群环境时，它就是`session`

---

1. 创建`bean`的三种方式
2. `bean`对象的作用范围
3. `bean`对象的生命周期

三种创建Bean对象的方式

1. 使用默认构造函数创建

> - 在`Spring`的配置文件中使用`bean`标签，配以`id`和`class`属性之后，且没有其他的属性和标签
> - 这就是采用默认构造函数创建`bean`对象，此时假如类中没有默认构造函数，则对象无法创建

2. 使用普通工厂中的方法创建对象

> - 其实我们的`bean.xml`中实现的就是直接返回给你一个对象，也就是没有写工厂类之前的`new 对象`
> - 但是假如代码也实现了工厂类，我们的`Spring`也可以根据我们自己实现的这个工厂类来获取对象
> - 也就是说我们不再以`class`得到对象了，而是以工厂类来得到的了

- `BeanFactory.class`

```
package com.bean.factory;

import com.bean.service.impl.AccountServiceImpl;

import java.io.IOException;
import java.io.InputStream;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class BeanFactory {

    public AccountServiceImpl getAccountService(){
```

```
            return new AccountServiceImpl();
        }
    }
```

- `bean.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">

    <!--假如我们现在也实现了工厂类，那么我们也可以使用工厂类来获取对象
        1. 配置bean工厂：像配置其他对象一样配置工厂
            1. 配置id
            2. 配置class
        2. 配置bean工厂产生的对象
            1. 配置id
            2. 指定是哪个工厂
            3. 指定工厂中的方法
    -->

    <!--配置bean工厂
        id：指定唯一标识
        class：指定类的路径
        factory-bean：指定是哪个工厂
        factory-method：指定是哪个方法
    -->
    <bean id="beanFactory" class="com.bean.factory.BeanFactory"></bean>
    <!--配置bean工厂产生的对象-->
    <bean id="accountService" factory-bean="beanFactory" factory-method="getAccountService">
</bean>

</beans>
```

3. 使用工厂中的静态方法创建对象

> - 我们之前写过工厂的静态类，第三种方法就是使用工厂中的静态方法创建对象，并存入 `Spring` 容器

```
<?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">

    <!--假如我们现在也实现了工厂类，那么我们也可以使用工厂类来获取对象
        1. 配置bean工厂和方法：像配置其他对象一样配置工厂
            1. 配置id
            2. 配置class
            3. 指定是工厂中的什么方法（必须是静态方法）
    -->

    <!--配置bean工厂
        id：指定唯一标识
        class：指定类的路径
        factory-bean：指定是哪个工厂
```

```
18          factory-method：指定是哪个方法，只不过如果并在一起，这个方法就不是普通方法了，而是静态方法
19      -->
20      <bean id="beanFactory" class="com.bean.factory.BeanFactory" factory-
    method="getAccountService"></bean>
21
22  </beans>
```

- 第三种方法和第二种方法差不不是很大，区别就在于一个是静态方法，一个不是静态方法
- 那么可能有这个问题：这个不是已经 `new` 了么？这不是耦合了吗？

  > 原因在于：
  >
  > - 我们第二种和第三种解决的问题是存在 `jar` 包中的代码， `jar` 包中的代码是 `.class` 文件，不能更改的，
  > - 所以我们有些时候没办法像第一种方式一样直接来创建对象的时候，就需要这两种方法

- 所以解决的问题是：如果你想通过 `Spring` 容器来获得 `jar` 包中的某个方法的话就可以采用这两种方法

作用范围

- `bean` 标签的 `scope` 属性：用于指定 `bean` 的作用范围
  - `singleton` ：单例的（默认值），常用
  - `prototype` ：多例，常用
  - `request` ：作用于 `web` 应用的请求范围
  - `session` ：作用于 `web` 应用的会话范围
  - `global-session` ：作用于集群环境的会话范围（全局会话范围），当不是集群环境时，它就是 `session`

生命周期

- 单例对象
  - 出生：当容器创建时，对象出生
  - 活着：只要容器还在，一直活着
  - 死亡：容器销毁，对象死亡

  > - 单例对象的生命周期和容器的生命周期相同
  >
  > > 单例对象当容器销毁的时候是会死亡的，但是如果我们不做手动关闭容器是不会发现的，因为 `java` 的 `main` 函数是所有程序执行的入口，当执行完成之后就进行了内存的释放，也就是说容器还没来得及调用销毁方法就已经被释放了内存，所以我们要进行手动调用关闭方法
  >
  > > 手动调用关闭方法的时候还有一件事，就是：
  > >
  > > 我们现在使用的是多态的形式创建的对象：
  > >
  > > ```
  > > 1   ApplicationContext applicationContext = new
  > >     ClassPathXmlApplicationContext("bean.xml")
  > > ```

> 有个问题就是 `ApplicationContext` 没有关闭的方法，只有子类才有，所以我们不应该使用多态的形式调用

- `AccountServiceImpl.class`

```
package com.bean.service.impl;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.service.IAccountService;

public class AccountServiceImpl implements IAccountService {


    public AccountServiceImpl() {
        System.out.println("对象创建了");
    }

    public void init(){
        System.out.println("对象进行了初始化");
    }

    public void saveAccount() {
        System.out.println("对象进行了保存操作");
    }

    public void destory(){
        System.out.println("对象进行了关闭");
    }
}
```

- `bean.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">


    <bean id="accountService" class="com.bean.service.impl.AccountServiceImpl" init-method="init" destroy-method="destory"></bean>


</beans>
```

- `cline.class`

```
package com.bean.ui;

import com.bean.dao.IAccountDao;
import com.bean.service.IAccountService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

```

```java
/**
 * 这里是表现层，用于调用业务层，实际开发中这里应该是servlet，但是这只是模拟
 */
public class cline {


    public static void main(String[] args) {

        /*不能使用多态*/
        ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("bean.xml");

        IAccountService accountService = (IAccountService) applicationContext.getBean("accountService");

        accountService.saveAccount();

        applicationContext.close();

    }
}
```

- 结果

```
对象创建了
对象进行了初始化
对象进行了保存操作
十一月 25, 2019 10:21:17 上午 org.springframework.context.support.AbstractApplicationContext doClose
信息: Closing org.springframework.context.support.ClassPathXmlApplicationContext@bebdb06: startup date [Mon Nov 25 10:21:16 CST 2019]; root of context hierarchy
对象进行了关闭
```

- 多例对象
  - 出生：当使用对象时，对象创建
  - 活着：在使用的时候，对象活着
  - 死亡：当对象长时间不用且没有别的对象使用时，由垃圾回收机制回收

> 多例对象在使用对象的时候创建，并且一直活着，知道`java`垃圾回收，`spring`是不回收的

`spring`的依赖注入

- 依赖注入
  - `Dependency Injection`
- `IOC`的作用
  - 降低程序间的耦合
- 依赖关系的管理：

- 以后都交给 `Spring` 来维护

> - 在当前类需要用到其他类的对象，以后都交给 `spring` 为我们提供，我们只需要在配置文件中说明
> - 这些依赖关系的维护就叫做依赖注入

- 依赖注入
  - 能注入的数据：有三类
    - 基本类型和 `String`
    - 其他 `bean` 类型（在配置文件中或者注解配置过的 `bean` ）
    - 复杂类型/集合类型
  - 注入的方式：有三种
    - 使用构造函数提供
    - 使用 `set` 方法提供
    - 使用注解提供（明天）

> - 注意，经常变化的数据是不适用于注入的方式的

构造函数注入

- 构造函数标签： `<constructor-arg></constructor-arg>`
  - `type` ：要注入数据的类型
  - `index` ：要注入数据的索引位置
  - `name` ：要注入数据的名称
  - `value` ：要注入数据的值
  - `ref` ：要注入数据的 `bean` 引用

1. 首先写一个构造函数

```java
package com.bean.service.impl;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.service.IAccountService;

import java.util.Date;

public class AccountServiceImpl implements IAccountService {

    private String name;
    private Integer age;
    private Date birthday;

    /**
     * 我们使用构造函数来进行依赖注入
     * @param name  注意，是Integer类型
     * @param age   注意，是String类型
     * @param birthday  注意，是Date类型,也就是属于其他bean类型的一种
     */
    public AccountServiceImpl(String name, Integer age, Date birthday) {
        this.name = name;
        this.age = age;
```

```
        this.birthday = birthday;
    }

    public void saveAccount() {
        System.out.println("对象进行了保存操作："+name+", "+age+", "+birthday);
    }
}
```

2. `bean.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">


    <!--
        constructor-arg：在<bean></bean>的内部使用，用于通过使用构造函数的方法对对象内部数据进行赋值，进行依赖注入
            type：指定要注入的数据类型，比如java.lang.String，但是缺点假如要有多个相同类型的对象就会报错
            index：指定要注入的数据的索引位置，从0开始。这个数据的注入要比type属性更方便一些，但是每次还要去数位置
            name：指定要注入的数据的名称是什么，是三个里面最好用的一个，实际开发中常用
            ==================以上三个是指定要注入的数据的位置==================================
            value：指定基本类型和String类型的数据，只要value输进去就可以自动转换，比如在这个例子中value="18"，会自动转为Integer
            ref：指定其余的bean类型，其余的bean类型需要定义，使用xml或者注解进行其余的bean类型的定义
    -->
    <bean id="accountService" class="com.bean.service.impl.AccountServiceImpl">
        <constructor-arg name="name" value="名字"></constructor-arg>
        <constructor-arg name="age" value="18"></constructor-arg>
        <constructor-arg name="birthday" ref="now"></constructor-arg>
    </bean>
    <!--一个新的bean类型的定义，spring使用反射的方式找到了Date类并初始化了一个新的对象进来-->
    <bean id="now" class="java.util.Date"></bean>

</beans>
```

3. 结果

```
对象进行了保存操作：名字, 18, Tue Nov 26 20:56:03 CST 2019
```

优缺点分析

- 优点
  - 获取`bean`对象时，注入数据是必须的操作，否则对象无法创建成功
- 缺点
  - 改变了`bean`对象的实例化方式，使得我们在创建对象时，如果用不到这些数据也必须提供

Set方法注入 更常用

- 使用`set`方法注入的时候只需要使用`set`方法而不需要使用`get`方法
- 其实虽然说是`set`方法，但是标签是`<property></property>`

- `name`：要注入数据的`set`方法名称
- `value`：要注入数据的值
- `ref`：要注入数据的`bean`引用

1. 还是那个方法，把构造方法去了，改为了`set方法`方法

```java
package com.bean.service.impl;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.service.IAccountService;

import java.util.Date;

public class AccountServiceImpl implements IAccountService {

    private String name;
    private Integer age;
    private Date birthday;

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

    public void setAge(Integer age) {
        this.age = age;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public void saveAccount() {
        System.out.println("对象进行了保存操作："+name+", "+age+", "+birthday);
    }
}
```

2. `set`方法注入

```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">


    <!--
        property：在<bean></bean>的内部使用，用于通过使用构造函数的方法对对象内部数据进行赋值，进行依赖注入
            name：指定要注入的set方法去掉set的名称，也就是setName=>Name=>name
            value：指定基本类型和String类型的数据，只要value输进去就可以自动转换，比如在这个例子中value="18"，会自动转为Integer
            ref：指定其余的bean类型，其余的bean类型需要定义，使用xml或者注解进行其余的bean类型的定义
    -->
```

```
    <bean id="accountService" class="com.bean.service.impl.AccountServiceImpl">
        <property name="name" value="名称"></property>
        <property name="age" value="18"></property>
        <property name="birthday" value="now"></property>
    </bean>
    <!--一个新的bean类型的定义，spring使用反射的方式找到了Date类并初始化了一个新的对象进来-->
    <bean id="now" class="java.util.Date"></bean>

</beans>
```

优缺点分析

- 优点
  - 不必对每一个数据进行注入也可以得到对象
- 缺点
  - 当你想要一个数据的时候可能发现它没有注入进来

复杂类型的注入

- 复杂类型
  - 数组
  - `List`
  - `Map`
  - `Properties`
- 我们使用`set`方式进行注入

> 使用`Set`方式进行注入的时候，使用`value`属性和`ref`属性都不好使，所以应该有子标签了
>
> - 用于给`List`结构集合注入的标签
>   - `list`
>   - `array`
>   - `set`
> - 用于给`Map`结构集合注入的标签
>   - `map`
>   - `properties`

- `bean.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">

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

        <!--
            首先提示一点：
                虽然标签内容看起来多，但是相同结构是可以相互替换的，一共就只有两组
                第一组：array   list
                第二组：map     props
```

```
14          -->
15
16
17          <!--array标签
18              value：值
19          -->
20          <property name="strings">
21              <array>
22                  <value>string1</value>
23                  <value>string2</value>
24              </array>
25          </property>
26
27          <!--list标签
28              value：值
29          -->
30          <property name="list">
31              <list>
32                  <value>list1</value>
33                  <value>list2</value>
34              </list>
35          </property>
36
37          <!--map标签
38              entry
39                  key：只能为entry的属性，为键
40                  value：可以为entry的属性，可以为entry的子标签，但是不能为entry内部的直接值，为值
41          -->
42          <property name="map">
43              <map>
44                  <entry key="mapKey1" value="mapValue1"></entry>
45                  <entry key="mapKey2">
46                      <value>mapValue2</value>
47                  </entry>
48              </map>
49          </property>
50
51          <!--props标签
52                  key：键，只可以为prop标签的属性，为键
53                      值：这个意思是说只可以为props内部的直接值，不可以为属性，也不可以为子标签
54          -->
55          <property name="properties">
56              <props>
57                  <prop key="proKey1">proValue1</prop>
58                  <prop key="proKey1">proValue2</prop>
59              </props>
60          </property>
61      </bean>
62
63  </beans>
```

第二天

第二天内容介绍

1. `spring` 基于注解的 `IOC` 以及 `IOC` 的案例
2. 案例中使用注解方式和 `xml` 方式实现单表的 `CRUD`
    - 持久层选型： `DBUtils`
3. 改造基于注解的 `IOC` 案例，使用春注解的方式实现
    - `spring` 的一些新注解使用
4. `spring` 和 `Junit` 整合

明确

- 明确一件事：使用注解的方式和使用 `xml` 的方式都是一样的

常用 `IOC` 注解按作用分类

- 用于创建对象的
- 用于注入数据的
- 用于改变作用范围的
- 和生命周期相关的

- 用于创建对象的

> 就和在 `xml` 配置中编写一个 `bean` 标签实现的功能是一样的，例如：

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

- 用于注入数据的

> 就和在 `xml` 配置中写一个带有 `<property></property>` 子标签的 `bean` 标签是一样的

```
<bean id="accountService" class="com.bean.service.impl.AccountServiceImpl">
     <property name="strings">
         <array>
             <value>string1</value>
             <value>string2</value>
         </array>
     </property>
 </bean>
```

- 用于改变生命周期的

> 就和在 `xml` 配置中写一个带有 `init-method` 和 `destory-method` 属性的 `bean` 标签是一样的
>
> ```
> 1    <bean id="accountService" class="com.bean.service.impl.AccountServiceImpl" init-
>      method="saveAccount" destroy-method="saveAccount"></bean>
> ```

- 和生命周期相关的

> 就和在 `xml` 配置中写一个带有 `scope` 属性的 `bean` 标签是一样的
>
> ```
> 1    <bean id="accountService" class="com.bean.service.impl.AccountServiceImpl"
>      scope="singleton"></bean>
> ```

注意事项

> - 我们在使用注解之前，要先告诉 `spring` 我要使用注解，注解的内容从哪里读
> - 所以我们现在使用的约束就不够用了，我们应该添加上注解的约束
> - 而注解的约束可以去官方文档下找： **https://docs.spring.io/spring/docs/5.2.1.RELEASE/spring-framework-reference/core.html#spring-core**
>
> ```
> 1   <beans xmlns="http://www.springframework.org/schema/beans"
> 2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
> 3    xmlns:context="http://www.springframework.org/schema/context"
> 4    xsi:schemaLocation="http://www.springframework.org/schema/beans
> 5        https://www.springframework.org/schema/beans/spring-beans.xsd
> 6        http://www.springframework.org/schema/context
> 7        https://www.springframework.org/schema/context/spring-context.xsd">
> 8   </beans>
> ```
>
> - 使用 `<context:component-scan base-package="com.bean"></context:component-scan>` ，告诉所有的注解内容都去这个包下找

用于创建对象的注解

`@Component`

- `@Component`
  - `value` ：属性为容器的 `id`

> - 当 `value` 自定义时
>   - 当 `value` 名称不写 `value` ，直接写值时： `@Component("accountService")`
>   - 容器就是： `accountService` ： `new AccountServiceImpl()`
>   - 当 `value` 名称也写上时： `@Component(value="accountService")`
>   - 容器就是： `accountService: new AccountServiceImpl()`

    - 当 `value` 值不写的时候
        - 默认就是类的小驼峰式写法

- `bean.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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">
    <!--使用上面的约束告诉spring可以使用xml，也可以使用注解，多导入的是context的名称空间-->

    <!--告诉spring在创建容器时要扫描的包去"com.bean"下找，当扫描包的时候就会扫描到包里的注解-->
    <context:component-scan base-package="com.bean"></context:component-scan>

</beans>
```

- `AccountSeriviceImpl`

```java
package com.bean.service.impl;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.service.IAccountService;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.List;
import java.util.Map;
import java.util.Properties;


/*
* @Component：使用注解，注解的作用是用于创建对象
*       众所周知，在spring中对象的形式是以Map形式储存的，其中值为对象，那么键就要自己定义
*           注解中有一个属性为value，value定义的就是Map的键
*               当value值为一个时可以不写value，也就是@Component("accountService")
*                   - accountService：new AccountServiceImpl()
*               注解的value值也可以不写，当然不写的时候默认为类的小驼峰式写法，也就是acountServiceImpl
*                   - accountServiceImpl：new AccountServiceImpl()
* */
@Component(value="accountService")
public class AccountServiceImpl implements IAccountService {

    private String[] strings;       //数组
    private List list;              //List
    private Map map;                //Map
    private Properties properties;  //properties

    public void setStrings(String[] strings) {
        this.strings = strings;
    }
```

```
34
35      public void setList(List list) {
36          this.list = list;
37      }
38
39      public void setMap(Map map) {
40          this.map = map;
41      }
42
43      public void setProperties(Properties properties) {
44          this.properties = properties;
45      }
46
47      public void saveAccount(){
48          System.out.println("用户保存");
49      }
50  }
```

- `cline.class`

```
1   package com.bean.ui;
2
3   import com.bean.dao.IAccountDao;
4   import com.bean.service.IAccountService;
5   import org.springframework.context.ApplicationContext;
6   import org.springframework.context.support.ClassPathXmlApplicationContext;
7   import org.springframework.context.support.FileSystemXmlApplicationContext;
8
9   /**
10   * 这里是表现层，用于调用业务层，实际开发中这里应该是servlet，但是这只是模拟
11   */
12  public class cline {
13
14
15      public static void main(String[] args) {
16
17          ApplicationContext applicationContext = new
    ClassPathXmlApplicationContext("bean.xml");
18
19          /*注意在这我们要根据容器获取对象，对象的key就是我们在注解中定义的值，注解中我写的是
    accountService*/
20          IAccountService accountService = (IAccountService)
    applicationContext.getBean("accountService");
21
22          accountService.saveAccount();
23
24      }
25  }
```

其他用于创建对象的注解

- `@Controller`：一般用于表现层
- `@Service`：一般用于业务层
- `@Repository`：一般用于持久层

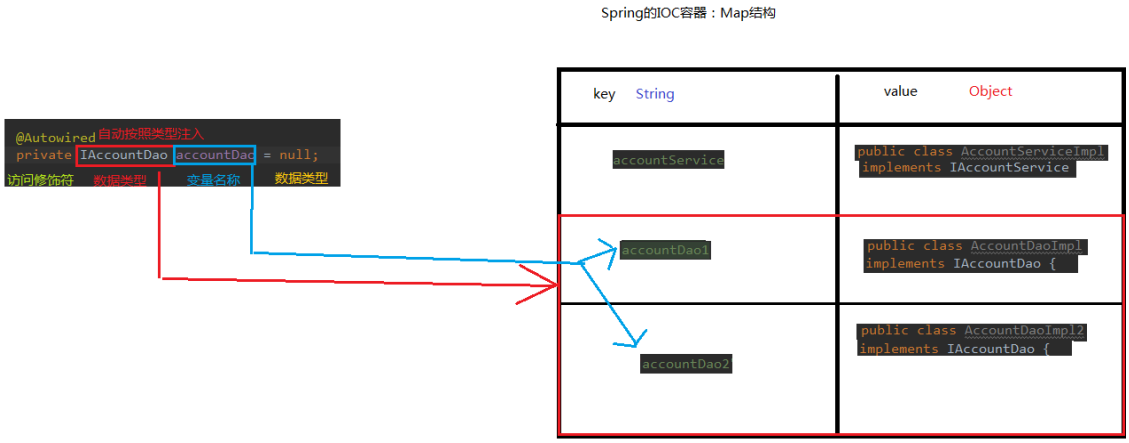
- 以上三个注解和 `@Component` 的作用都是一样的，都是用于创建对象的，那么有什么区别呢?
- 他们三个是 `spring` 框架为我们提供明确三层使用的注解，是我们的三层对象更加清晰

用于注入数据的注解

只能注入 `bean` 类型的数据

`@Autowried`

- 按照类型注入



我们上面有张图，图片的内容是关于自动按照类型注入的，下面讲解一下：

1. 首先给能给变量 `accountDao` 注入是因为有对应的类型 `IAccountDao` ，也就是红框圈的部分
    1. 红框圈出来的部分有两个类： `AccountDaoImpl` 和 `AccountDaoImpl2`
    2. 这两个都继承了 `IAccountDao` ，所以类型对应起来了
    3. 假如没有这个继承，就没有对应的数据类型，就会报错
2. 红框圈起来的两个类都继承了 `IAccountDao` ，也就是说有两个对应的数据，那么应该注入哪一个类型
    1. `spring` 的类型匹配方式是先注意类型，假如类型相同看变量的名称是什么
    2. 图中的变量名称为 `accountDao` ，也就是说类型相同过的时候应该找 `accountDao`
    3. 我们看 `IOC` 容器中，他们的 `key` 没有一个叫做 `accountDao` ，所以这应该找不到一个类型注入
    4. 所以会直接报错

- 下面演示一个不报错的

- `AccountDaoImpl`

```java
package com.bean.dao.impl;

import com.bean.dao.IAccountDao;
import org.springframework.stereotype.Repository;
//注意这里，accountDao1
@Repository("accountDao1")
public class AccountDaoImpl implements IAccountDao {
    public void saveAccount() {
        System.out.println("模拟已经保存了账户");
    }
}
```

- `AccountDaoImpl2`

```java
package com.bean.dao.impl;

import com.bean.dao.IAccountDao;
import org.springframework.stereotype.Service;

@Service("accountDao2")
public class AccountDaoImpl2 implements IAccountDao {
    public void saveAccount() {
        System.out.println("模拟已经保存了账户");
    }
}
```

- `AccountServiceImpl`

```java
package com.bean.service.impl;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.service.IAccountService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

import java.util.Date;
import java.util.List;
import java.util.Map;
import java.util.Properties;

@Service("accountService")
public class AccountServiceImpl implements IAccountService {

    @Autowired
    private IAccountDao accountDao1 = null;

    public void saveAccount(){
        System.out.println("用户保存");
    }
}
```

- `cline.class`

```java
package com.bean.ui;

import com.bean.dao.IAccountDao;
```

```java
import com.bean.service.IAccountService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;


public class cline {


    public static void main(String[] args) {

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

        IAccountService accountService = (IAccountService)
applicationContext.getBean("accountService");

        accountService.saveAccount();

    }
}
```

@Qualifier

- 作用：在按照类中注入的基础上再按照名称注入
  - 它在给类成员注入时不能单独使用，要与 `@Autowired` 配合
  - 它在给方法参数注入时可以使用
- `@Qualifier`：属性
  - value：指定注入 `bean` 的 `id`

```java
package com.bean.service.impl;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.service.IAccountService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

import java.util.Date;
import java.util.List;
import java.util.Map;
import java.util.Properties;

@Service("accountService")
public class AccountServiceImpl implements IAccountService {

    @Autowired
    @Qualifier(value = "accountDao2")  //我就想注入这个accountDao2，没问题
    private IAccountDao accountDao1 = null;

    public void saveAccount(){
        System.out.println("用户保存");
    }
```

```
    }
```

`@Resource`

- 作用：直接根据 `bean` 的 `id` 进行注入，可以独立使用
- `@Resource` 属性：
  - `name` ：指定 `bean` 的 `id`

```
package com.bean.service.impl;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.service.IAccountService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import java.util.Date;
import java.util.List;
import java.util.Map;
import java.util.Properties;

@Service("accountService")
public class AccountServiceImpl implements IAccountService {

    @Resource(name = "accountDao2")
    private IAccountDao accountDao1 = null;

    public void saveAccount(){
        System.out.println("用户保存");
    }
}
```

注意事项

- 以上三种注解只能注入 `bean` 类型的数据
- 集合类型的注入只能通过 `xml` 来实现

基本类型和String类型的注入

`@value`

- `@Value` ：可以注入基本类型和 `String` 类型的数据
- 属性
  - `value` ：用于指定数据的值，可以使用 `spring` 中的 `SpEL` ，也就是 `srping的el` 表达式
    - `SpEL` 的写法： `${表达式}`

用于指定作用范围的注解

`@Scope`

- `@Scope`：指定`bean`的作用范围
  - `value`：指定范围的取值。
    - `singleton`：单例，默认
    - `protoype`：多例

和生命周期相关

- `@PreDestory`
  - 指定销毁方法
- `@PostConstruct`
  - 指定初始化方法

> - 注意，因为父类没有`close`这个方法，所以不能用多态来测试销毁方法
> - 多例状态下销毁方法是由垃圾回收执行的，你`close`也没用
> - 新版本的`spring`这俩注解已经没了

Spring新注解

指定配置类，指定扫描包，注入`bean`对象注解

- `@Configuration`：作用是指定当前类是一个配置类
- `@ComponentScan`：通过注解指定`spring`要创建容器时时要扫描的包
  - `basePackages`：别名是`value`
  - `value`：别名是`basePackages`
- `@Bean`：用于把当前方法的返回值作为`bean`对象存入到`spring`的`ioc`容器中
  - `name`：指定`bean`的`id`，默认值为当前方法的名称，也就是存入容器中的`key`

细节

> 当我们使用注解配置方法的时候，假如方法有参数，`spring`框架会去容器中查找有没有可用的`bean`对象
>
> 查找的方式和`@Autowired`是一样的

使用新方式创建容器

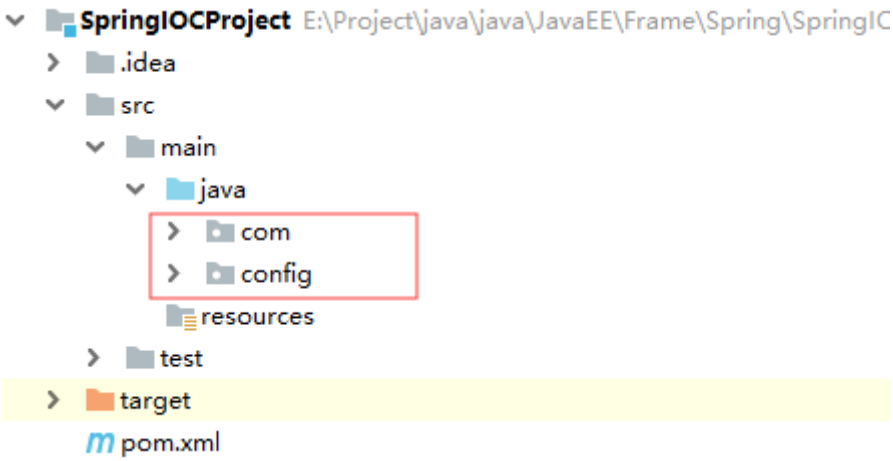
> 在之前，我们创建容器使用的方式是：

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

但是我们现在使用的是注解，也就不能这么配了，所以应该采用新的方式创建容器

`ApplicationContext context = AnnotationConfigApplicationContext(被注解的类)`

- 下面来个例子



- `config.SpringConfiguration`

```java
package config;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import java.util.Date;

/**
 * 这里是配置注解的内容
 * @Configuration：指定这个类是配置类
 *
 * @ComponentScan：指定spring在创建容器的时候要扫描的包，等同于<context:component-scan base-package="com.bean"></context:component-scan>
 *      - value：指定类路径，别名为basePackages，也就是说写basePackages也可以
 *      - basePackages：指定类路径，别名为value，也就是说写value也可以
 *      - value={"com.bean"}：我们学过当数组中有且只有一个值的时候可以省略括号，然后可以省略value，也就是"com.bean"
 *
 * @Bean：指定向容器中存储的bean对象，id默认值为当前方法的名称
 */
@Configuration  //声明了这个类是配置类
@ComponentScan("com.bean")  //在spring要创建容器的时候去读取com.bean下的文件，这样就能找到注解了
public class SpringConfiguration {

    //创建了key=date的bean对象
    @Bean("date")
    public Date getDate(){
        return new Date();
    }

}
```

- `com.bean.dao.impl.AccountDaoImpl`

```java
package com.bean.dao.impl;

import com.bean.dao.IAccountDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import javax.annotation.Resource;
import java.util.Date;

//这里创建了key=accountDao的bean对象
@Repository("accountDao")
public class AccountDaoImpl implements IAccountDao {

    //这里引用key=date的bean对象
    @Resource(name = "date")
    private Date date;

    public void saveAccount() {

        System.out.println("模拟已经保存了账户"+", "+date.toString());
    }
}
```

- `com.bean.service.impl.AccountServiceImpl`

```java
package com.bean.service.impl;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.service.IAccountService;
import config.SpringConfiguration;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;
import java.util.Date;
import java.util.List;
import java.util.Map;
import java.util.Properties;

//这里创建了key=accountService的bean对象
@Service("accountService")
public class AccountServiceImpl implements IAccountService {

    //这里引用了key=accountDao的bean对象
    @Resource(name = "accountDao")
    private IAccountDao accountDao;

    public void saveAccount(){
```

```
29          System.out.println("AccountService调用-->");
30          accountDao.saveAccount();
31      }
32  }
```

- `com.bean.ui.cline`

```
1   package com.bean.ui;
2
3   import com.bean.dao.IAccountDao;
4   import com.bean.service.IAccountService;
5   import config.SpringConfiguration;
6   import org.springframework.context.ApplicationContext;
7   import org.springframework.context.annotation.AnnotationConfigApplicationContext;
8   import org.springframework.context.support.ClassPathXmlApplicationContext;
9   import org.springframework.context.support.FileSystemXmlApplicationContext;
10
11
12  public class cline {
13
14
15      public static void main(String[] args) {
16
17          ApplicationContext applicationContext = new
    AnnotationConfigApplicationContext(SpringConfiguration.class);
18
19          IAccountService accountService = (IAccountService)
    applicationContext.getBean("accountService");
20
21          accountService.saveAccount();
22
23      }
24  }
```

细节

- `@Configuration`：当配置类作为`AnnotationConifgApplicationContext`对象创建的参数时，这个注解可以不写

  > 意思是说在创建容器的时候把这个类的字节码作为参数传进去了，比如：
  >
  > `ApplicationContext context =`
  >
  > `new AnnotationConfigApplicationContext(SpringConfiguration.class);`
  >
  > 这个就是将`SpringConfiguration.class`传进去了

  > 但是只有以上这种情况可以不写，其他情况还是需要写上的，比如我在这个配置类中调用配置类2，那么配置类2就必须写了，因为配置类2首先必须是一个配置类，才会被调用，那这个配置类2要使用就是在配置1中：
  >
  > `@Configuration({"com.bean","配置类2"})`，但是这个不好用，有一个好用的注释是`@Import`

@Import

- @import：用于引入其他的配置类
  - value：字节码
  - 如 @Import(textImport.class)，导入的配置都是子配置

@PropertySource

- @Properties：用于指定 properties 文件的路径
  - value：用于指定文件的名称和路径

> 关键字：classpath
>
> classpath 用于指定文件的路径，比如：
> @Properties("classpath:config/jdbcConfig.properties")
>
> - 因为我们都知道，在项目运行之后 java 下和 reousrce 下的文件都会跑到 classes 下，也就是我们常说的 classpath，所以要读取这个内容，就要使用 classpath 来指定文件的路径

Spring 整合 junit

问题分析

1. 程序的入口函数是 main 方法
2. junit 单元测试中没有 main 方法也会执行
   1. 因为 junit 集成了一个 main 方法
   2. 该方法会判断当前测试类中哪些方法有 @Test 注解
   3. junit 会让有 Test 注解的方法执行
3. junit 不会管我们是否采用 spring 框架
   1. 在测试方法中，junit 根本就不知道我们使用的技术，所以根本不会读取配置创建容器
4. 由以上三十点可以知道
   1. 当测试方法执行时，没有 IOC 容器，即使采用注解配置也不会实现注入

解决

- 我们采用替换掉 junit 中 main 的方式来创建容器

1. 首先导入 spring 为 junit 整合的 jar(坐标)

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

2. 使用注解 `@RunWith(SpringJunit4ClassRunner.class)` 替换掉 `main` 方法
3. 告知 `spring` 的运行器， `spring` 的 `ioc` 是基于 `xml` 还是注解的，并说明位置
    1. `ContextConfiguration`
        1. `locations` ： 指定 `xml` 文件所在位置，加上 `classpath` 关键字，表示在类路径下
        2. `classes` ： 指定注解类所在位置

注意

> 当我们使用 `spring 5.x` 版本的时候，要求 `junit` 的 `jar` 必须为 `4.12` 及以上，否则会报错

第三天

课程内容介绍

1. 完善 `account` 案例
2. 分析案例中的问题
3. 回顾动态代理
4. 动态代理另一种实现方式
5. 解决案例中的问题
6. `AOP` 的概念
7. `spring` 的 `AOP` 相关术语
8. `spring` 中基于 `xml` 和注解的 `AOP` 配置

代理的分析

> 在生活中，我们也会出现代理的现象：
>
> - 电脑，手机，生活等代理商代理这些产品，生产厂家给代理商，我们售后找的也是代理商
> - 比如某猫，某东，等

动态代理

动态代理：

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

基于接口的代理回顾

- 基于接口的动态代理
  - 涉及的类：`Proxy`
  - 提供者：`JDK` 官方
- 如何创建代理对象：
  - 使用 `Proxy` 类中的 `newProxyInstance` 方法
- 创建代理对象的要求：
  - 被代理类最少实现一个接口，如果没有则不能使用
- `newProxyInstance`
  - `ClassLoader`：类加载器

    > 用于加载代理对象字节码的，和被代理对象使用相同的类加载器

  - `Class[]`：字节码数组

    > 它是用于让代理对象和被代理对象有相同的方法，两个都要实现相同的接口，也就是被代理对象的接口

  - `InvocationHandler`：提供增强的代码

    > - 他是让我们如何写代理。
    > - 我们一般都是写一个该接口的实现类，通常情况下都是匿名内部类，但不是必须的
    > - 此接口的实现类都是谁用谁写
    > - 执行被代理对象的任何 `接口方法` 都会经过该方法，有拦截的功能

- `IProducer`

```java
package com.bean.proxy;

/**
 * 生产者的接口，有一个销售方法
 */
public interface IProducer {

    /**
     * 实现销售方法
     */
    public void saleProducer(float money);

}
```

- `Producer`

```java
package com.bean.proxy.impl;

import com.bean.proxy.IProducer;

/**
 * 实现生产者
 */
public class Producer implements IProducer {

    /**
     * 实现销售方法
     * @param money
     */
    public void saleProducer(float money) {
        System.out.println("进行销售"+money);
    }
}
```

- `Clinet`

```java
package com.bean.proxy;

import com.bean.proxy.impl.Producer;

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

/**
 * 这个作为消费者模拟
 */
public class clinet{
    public static void main(String[] args) {

        /**
         * 动态代理
         *      特点：字节码随用随创建，随用随加载
         *      作用：在不修改源码的基础上对方法进行增强
         *      分类
         *          基于子类的动态代理
         *          基于接口的动态代理（回顾）
         *              涉及的类：Proxy
         *              提供者：JDK官方
         *      如何使用动态代理创建对象
         *          使用Proxy类中的newProxyInstance方法
         *      创建代理对象的要求：
         *              被代理对象至少要实现一个接口
         *              （注意这个是重点，也就是说我们的被代理对象poducer必须至少要实现一个接口，在这里就是IProducer.class.getCLassLoader()
         *
         *      newProxyInstance：有了这个，只要producer执行任何一个接口的方法都会经过这个方法
         *              ClassLoader：类加载器，指的是被代理对象的字节码，在这里就是Producer的字节码
         *              Class[]：字节码数组，用于让代理对象和被代理对象有相同的方法，也就是说它也要实现被代理对象的接口
         *                      在这里就是IProducer.class.getInterfaces()
         *              InvocationHandler：提供增强的代码
```

```
35              *                   proxy：当前被代理对象的引用，也就是方法本身，调用这个就等于调用方法，相当于递归
36              *                   method：当前执行的的方法名称是什么
37              *                       method.invoke()
38              *                           Obj：要执行谁的方法，这里是被代理对象的方法，也就是producer
39              *                           Obj...：方法的参数
40              *
41              *                   args：当前方法中执行的参数
42              *
43              */
44
45              //必须是使用final修饰才能被代理
46              final Producer producer = new Producer();
47
48              //下面是动态代理，也就是我们模拟的代理商
49              IProducer iProducer = (IProducer) Proxy.newProxyInstance(producer.getClass().getClassLoader(),
50                      producer.getClass().getInterfaces(),
51                      new InvocationHandler() {
52                      public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
53                          //只要当前执行的方法是saleProducter，那么
54                          if ("saleProducer".equals(method.getName())){
55                              float money = (Float) args[0];//那么获取参数值，因为只有一个参数所以就获取了
56                              money = money*0.8f; //抽2成的利润
57                              return method.invoke(producer,money);   //返回方法，要执行的方法是producer的方法
58                          }
59                          //如果不是该方法，那么就什么也不做
60                          return method.invoke(method,args);
61                      }
62              });
63              iProducer.saleProducer(10000f);
64          }
65      }
```

- 结果

```
1   进行销售8000.0
```

基于子类的动态代理

1. 需要外部依赖

```
1               <dependency>
2                   <groupId>cglib</groupId>
3                   <artifactId>cglib</artifactId>
4                   <version>2.1_3</version>
5               </dependency>
```

- 基于子类的动态代理
  - 涉及的类：`Enhancer`
  - 提供者：第三方`cglib`

- 创建代理对象的方法：`EnHander` 中的 `create` 方法
- 创建代理对象的要求：被代理对象不能是最终类
- `create` 方法的参数
  - `Class`：被代理对象的字节码
  - `Callback`：提供增强方法
    - 在这里写如何进行代理，一般使用一些该接口的实现了类，通常情况下都是匿名内部类，但不是必须的
    - 此接口的实现类都是谁用谁写
    - 我们一般写的都是该接口的实现类：`MethodInterceptor`
    - `MethodInterceptor`
      - `o`：当前代理对象的引用
      - `method`：当前执行的方法
      - `objects`：当前执行的方法的参数
      - `methodProxy`：当前执行的方法的代理

- `Producer`

```java
package com.bean.cglib;

/**
 * 实现生产者
 */
public class Producer {

    /**
     * 实现销售方法
     * @param money
     */
    public void saleProducer(float money) {
        System.out.println("进行销售"+money);
    }
}
```

- `Clinet`

```java
package com.bean.cglib;

import com.bean.proxy.IProducer;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

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

/**
 * 这个作为消费者模拟
 */
public class clinet{
    public static void main(String[] args) {
```

```
17
18
19
20          /**
21           * 动态代理
22           *       特点：字节码随用随创建，随用随加载
23           *       作用：在不修改源码的基础上对方法进行增强
24           *       分类
25           *           基于子类的动态代理
26           *               涉及的类：Enhancer
27           *               提供者：第三方cglib库
28           *           基于接口的动态代理
29           *       如何使用动态代理创建对象
30           *           使用Enhancer中的create方法
31           *       创建代理对象的要求：
32           *               被代理对象不能为最终类
33           *
34           *       create方法的参数
35           *               Class：被代理对象的字节码
36           *               Callback：提供增强的代码
37           *                   在这里写如何代理。一般使用一些该接口的实现类，通常情况下都是匿名内部类，但不是必须的
38           *                   此接口的实现类都是谁用谁写
39           *                   我们一般写的都是该接口的实现类：MethodInterceptor
40           *
41           *                   MethodInterceptor
42           *                       o：当前代理对象的引用
43           *                       method：当前执行的方法
44           *                       objects：当前执行的方法的参数
45           *                       (以上三个和接口代理的参数一样)
46           *                       methodProxy：当前执行的方法的代理
47           */
48
49          //必须是使用final修饰才能被代理
50          final Producer producer = new Producer();
51
52          //下面是动态代理，也就是我们模拟的代理商
53          Producer proxyProducer = (Producer) Enhancer.create(producer.getClass(), new MethodInterceptor() {
54              public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {
55                  if ("saleProducer".equals(method.getName())){
56                      float money = (Float) objects[0];
57                      return method.invoke(producer,money*0.8f);
58                  }
59                  return null;
60              }
61          });
62          proxyProducer.saleProducer(10000f); //结果：进行销售8000.0
63      }
64  }
```

AOP

- `AOP`可译为面向切面编程，通过预编译方式和运行期`动态代理`实现程序功能的统一维护的一种技术
- `AOP`是`OOP（面向对象）`的延伸，是软件开发的一个热点
- `AOP`是`Spring`框架中的一个重要内容，是函数式编程的一种衍生泛型
- 利用`AOP`可以对业务逻辑的各个部分进行隔离，从而使业务逻辑的各个部分之间的耦合度降低，提高程序的可重用性。

- 优势
  - 减少重复代码
  - 提高开发效率
  - 维护方便
- 实现方式
  - 基于动态代理

`AOP`相关术语

- `JoinPoint（连接点）`：所谓连接点指的是那些被拦截到的点，在`spring`中，这些点指的是方法，因为`spring`只支持方法类型的连接点

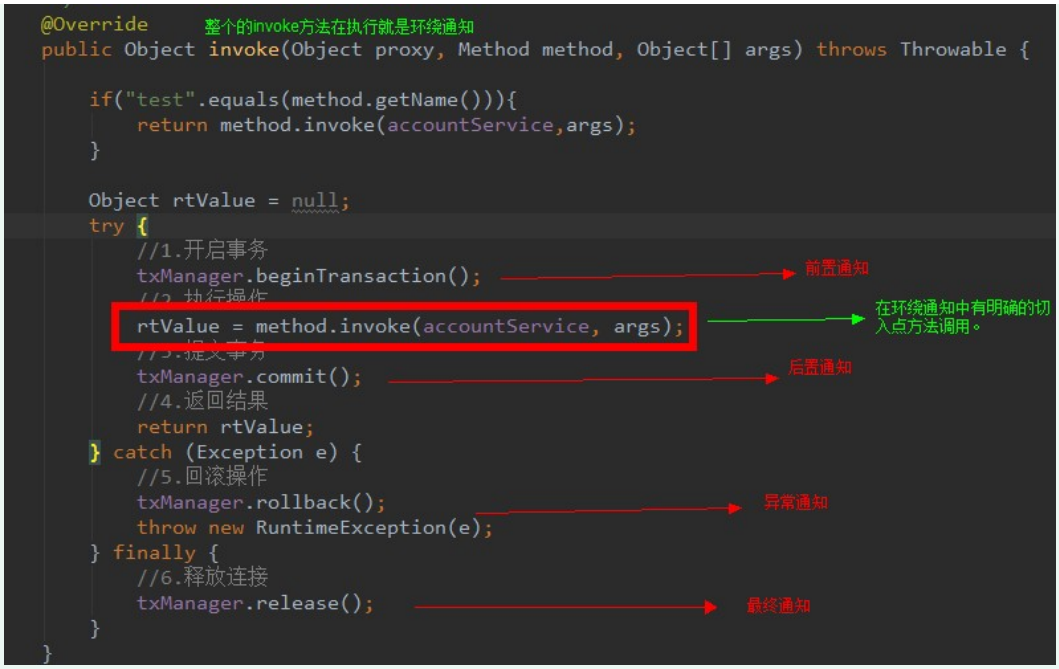
  > 用大白话来讲，就是在业务层中的代码可以和事务进行连接，对方法进行增强，业务层中的方法都是连接点

- `Pointcut（切入点）`：所谓切入点是指我们要对哪些`Joinpoint`进行拦截的定义

  > 在事务中被增强的方法就叫做切入点，因为我们要进行判断，不是所有的方法都是增强的

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

  > 我们知道在增强代码中，`invoke`有拦截的作用，那么拦截到`Joinpoint`之后，我们对方法增强的代码就叫做通知，而通知的类型我们也很好区分
  >
  > - 在`method.invoke(参数，参数)`之前：前置通知
  > - 在`method.invoke(参数，参数)`之后：后置通知
  > - 在`try-catch-finally`的`catch`中：异常通知
  > - 在`try-catch-finally`的`finally`中：最终通知
  > - 整个的`invoke`方法就是环绕通知，环绕通知中有明确的方法调用，就是`method.invoke()`



- `Introduction（引介）`：引介是一种特殊的通知，在不修改类代码的前提下，`Introduction` 可以在运行期为类动态地添加一些方法或者 `Field（成员变量）`
- `Target（目标对象）`：代理的目标对象
- `Weaving（织入）`：指把增强应用到目标对象来创建新的代理对象的过程，`spring` 采用动态代理织入，而 `AspectJ` 采用编译期织入和类装载器织入
- `Proxy（代理）`：一个类被 `AOP` 织入增强后，就产生一个结果代理类
- `Aspect（切面）`：是切入点和通知（引介）的结合

在 AOP 中我们应该做什么

1. 开发阶段（我们做）
    1. 编写核心业务代码
    2. 抽取公共代码，制作成通知
    3. 在配置文件中声明切入点与通知之间的关系，即切面

Spring 基于 XML 的 AOP 配置

跑个流程

- `pom.xml` 配置

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

    <groupId>com.bean</groupId>
    <artifactId>SpringAOP</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
```

```
11
12      <dependencies>
13          <dependency>
14              <groupId>org.springframework</groupId>
15              <artifactId>spring-context</artifactId>
16              <version>5.0.2.RELEASE</version>
17          </dependency>
18
19          <!--用于解析切入点表达式-->
20          <dependency>
21              <groupId>org.aspectj</groupId>
22              <artifactId>aspectjweaver</artifactId>
23              <version>1.8.7</version>
24          </dependency>
25      </dependencies>
26  </project>
```

- `com.bean.service.IAccountService`

```
1   package com.bean.service;
2
3   /**
4    *模拟操作用户，注意这三个的参数和返回值，主要讲解了三类方法
5    */
6   public interface IAccountService {
7
8       /**
9        * 模拟保存操作，无参无返回值
10       */
11      void saveAccount();
12
13      /**
14       * 模拟更新账户，有参无返回值
15       * @param i
16       */
17      void updateAccount(int i);
18
19      /**
20       * 模拟删除账户，无参有返回值
21       * @return
22       */
23      int deleteAccount();
24  }
```

- `com.bean.service.impl.AccountServiceImpl`

```
1   package com.bean.service.impl;
2
3   import com.bean.service.IAccountService;
4
5   public class AccountServiceImpl implements IAccountService {
6       public void saveAccount() {
7           System.out.println("保存方法执行了");
8       }
9
10      public void updateAccount(int i) {
11          System.out.println("更新方法执行了"+i);
12      }
```

```
13
14      public int deleteAccount() {
15          System.out.println("删除方法执行了");
16          return 0;
17      }
18  }
```

- `com.utils.Logger`

```
1   package com.bean.utils;
2
3   /**
4    * 模拟用于记录日志的工具类，里面提供了公共代码
5    */
6   public class Logger {
7
8       /**
9        * 要求：在启动操作用户的三个方法之前首先启动这个方法
10       * 之前我们可以使用动态代理的方式首先执行这个方法然后执行其他方法
11       * 但是假如我们不使用动态代理的方式而是使用spring配置的方式该如何做
12       * 这就是我们需要研究的问题
13       */
14      public void printLogger(){
15          System.out.println("Logger类中的printLogger方法开始记录日志了。。。。。。");
16      }
17
18  }
```

- `resource.bean.xml`

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <beans xmlns="http://www.springframework.org/schema/beans"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xmlns:aop="http://www.springframework.org/schema/aop"
5          xsi:schemaLocation="http://www.springframework.org/schema/beans
6           https://www.springframework.org/schema/beans/spring-beans.xsd
7           http://www.springframework.org/schema/aop
8           https://www.springframework.org/schema/aop/spring-aop.xsd">
9     <!--注意，上面是新的规范-->
10
11      <!--配置spring的IOC，把service对象配置进来
12          我们想要对service方法进行增强，使service中执行任意一个方法前都执行一个日志
13      -->
14      <bean id="accountService" class="com.bean.service.impl.AccountServiceImpl"></bean>
15
16
17      <!--spring中基于xml的AOP配置步骤：
18
19          - 在bean中加入想要的方法
20          - 配置切面，指定通知的方式，方法
21          - 与要增强的方法绑定
22
23          1. 把通知的bean也交给spring来管理
24          2. 使用aop:config配置AOP
25          3. 使用aop:aspect表明配置切面
26                  id属性：给切面配置唯一标识
27                  ref：指定通知类bean的ID
28          4. 在aop:aspect的内部使用对应标签来配置通知类型
```

```
            因为我们示例的是在切入点方法之前执行，所以为前置通知
            aop:before：前置通知
                method属性：用于指定Logger类中哪个方法是前置通知
                pointcut属性：用于指定切入点表达式，该表达式的含义指的是对业务层中哪个方法进行增强

                切入点表达式的写法：
                    关键字：excution（表达式）
                    表达式：
                        访问修饰符 返回值 包名.包名...类名.方法名(参数列表)
                        public void com.bean.service.impl.AccountServiceImpl.save()
    -->

    <!--我们有这个通知类，通知类就是记录日志，我们也交给spring-->
    <bean id="logger" class="com.bean.utils.Logger"></bean>

    <!--配置AOP-->
    <aop:config>
        <!--配置切面，引用通知方法-->
        <aop:aspect id="logAdvice" ref="logger">
            <!--配置通知的方式，指定被增强的方法-->
            <aop:before method="printLogger" pointcut="execution(public void
com.bean.service.impl.AccountServiceImpl.saveAccount())"></aop:before>
        </aop:aspect>
    </aop:config>


</beans>
```

- `test.com.bean.test.AOPTest`

```
package com.bean.test;

import com.bean.service.IAccountService;
import com.bean.service.impl.AccountServiceImpl;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AOPTest {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");

        IAccountService accountService = (IAccountService)
context.getBean("accountService");

        accountService.saveAccount();
    }
}
```

- 结果

```
Logger类中的printLogger方法开始记录日志了。。。。。。
保存方法执行了
```

> 上面写的只是增强了save()方法，所以很不现实

切入点表达式的全通配写法

`* *..*.*(..)`

- `bean.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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

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

    <bean id="logger" class="com.bean.utils.Logger"></bean>

    <aop:config>
        <aop:aspect id="logAdvice" ref="logger">
            <aop:before method="printLog" pointcut="execution(* *..*.*(..))"></aop:before>
        </aop:aspect>
    </aop:config>


</beans>
```

讲解：由`public void com.bean.service.impl.AccountServiceImpl.save()`变为`* *..*.*(..)`

1. 访问修饰符可以省略

```
void com.bean.service.impl.AccountServiceImpl.save()
```

2. 所有返回值可以使用`*`来代替

```
* com.bean.service.impl.AccountServiceImpl.save()
```

3. 包名可以使用`*.`来代替，但是有几级包就要写几级`*.`

```
* *.*.*.*.AccountServiceImpl.save()
```

4. 包名可以使用`..`来代替当前包及其子包

```
* *..AccountSerciceImpl.save()
```

5. 类名和方法名都可以使用`*`来进行通配

```
* *..*.*()
```

6. 参数问题
    1. 基本类型可以直接写名称：`int`
    2. 引用类型要使用全类名的方式：`java.lang.String`

```
* *..*.*(int)
```

7. 参数可以使用 `*` 来全部表示，但是必须有参数的情况下才能使用

```
* *..*.*(*)
```

8. 参数可以使用 `..` 的方式来表示有参数或者无参数的情况，有参数可以为任意类型

```
* *..*.*(..)
```

> 实际开发中不建议使用全通配符的方式，因为这代表着每一个方法都要进行增强，没有必要

实际开发中建议使用的通配表示方式

- 切到业务层实现类下的所有方法

```
* com.bean.service.impl.*.*(..)
```

- 我们在 `pom.xml` 中配置的 `org.aspectj` 就是配置这个用的

四种常用通知类型

- 前置通知
- 后置通知
- 异常通知
- 最终通知

- `Logger`

```
package com.bean.utils;

/**
 * 模拟用于记录日志的工具类，里面提供了公共代码
 */
public class Logger {

    /**
     * 前置通知
     */
    public void beforePrintLog(){
        System.out.println("Logger类中的前置通知方法开始记录日志了。。。。。。");
    }

    /**
     * 后置通知通知
     */
    public void afterReturningPrintLog(){
        System.out.println("Logger类中的后置通知开始记录日志了。。。。。。");
    }

    /**
     * 异常通知
     */
    public void afterThrowsPrintLog(){
```

```
        System.out.println("Logger类中的异常通知方法开始记录日志了。。。。。。");
    }

    /**
     * 最终通知
     */
    public void afterPrintLog(){
        System.out.println("Logger类中的最终通知方法开始记录日志了。。。。。。");
    }

}
```

- `bean.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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">
  <!--注意，上面是新的规范-->

    <!--配置spring的IOC，把service对象配置进来
        我们想要对service方法进行增强，使service中执行任意一个方法前都执行一个日志
    -->
    <bean id="accountService" class="com.bean.service.impl.AccountServiceImpl"></bean>


    <!--我们有这个通知类，通知类就是记录日志，我们也交给spring-->
    <bean id="logger" class="com.bean.utils.Logger"></bean>

    <!--配置AOP-->
    <aop:config>
        <!--配置切面，引用通知方法-->
        <aop:aspect id="logAdvice" ref="logger">
            <!--配置前置通知-->
            <aop:before method="beforePrintLog" pointcut="execution(*
com.bean.service.impl.*.*(..))"></aop:before>

            <!--配置后置通知-->
            <aop:after-returning method="afterReturningPrintLog" pointcut="execution(*
com.bean.service.impl.*.*(..))"></aop:after-returning>

            <!--&lt;!&ndash;异常通知&ndash;&gt;-->
            <aop:after-throwing method="afterThrowsPrintLog" pointcut="execution(*
com.bean.service.impl.*.*(..))"></aop:after-throwing>

            <!--&lt;!&ndash;最终通知&ndash;&gt;-->
            <aop:after method="afterPrintLog" pointcut="execution(*
com.bean.service.impl.*.*(..))"></aop:after>
        </aop:aspect>
    </aop:config>


</beans>
```

- `AOPTest`

```
package com.bean.test;

import com.bean.service.IAccountService;
import com.bean.service.impl.AccountServiceImpl;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AOPTest {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");

        IAccountService accountService = (IAccountService)
context.getBean("accountService");

        accountService.saveAccount();
    }
}
```

使用标签配置切入点表达式

- `<aop:pointcut></<aop:pointcut>>`
  - `id`
  - `expression`

> 此标签可以写在 `<aop:aspect></aop:aspect>` 里面，那么只能在这里面使用，再来一个切面要重新配，所以我们把它挪到外面
>
> 挪到外面之后可能发现会报错，所以注意这个东西必须在 `aop:aspect` 之前，因为这是约束。
>
> 不然就会报错，一定要注意

- `bean.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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">
  <!--注意，上面是新的规范-->

    <!--配置spring的IOC，把service对象配置进来
        我们想要对service方法进行增强，使service中执行任意一个方法前都执行一个日志
    -->
    <bean id="accountService" class="com.bean.service.impl.AccountServiceImpl"></bean>


    <!--我们有这个通知类，通知类就是记录日志，我们也交给spring-->
    <bean id="logger" class="com.bean.utils.Logger"></bean>

```

```xml
        <!--配置AOP-->
        <aop:config>
            <!--配置切面表达式-->
            <aop:pointcut id="pointCut" expression="execution(* com.bean.service.impl.*.*(..))">
    </aop:pointcut>
            <!--配置切面，引用通知方法-->
            <aop:aspect id="logAdvice" ref="logger">
                <!--配置前置通知，引用切面表达式-->
                <aop:before method="beforePrintLog" pointcut-ref="pointCut" ></aop:before>

                <!--配置后置通知，引用切面表达式-->
                <aop:after-returning method="afterReturningPrintLog" pointcut-ref="pointCut">
    </aop:after-returning>

                <!--异常通知，引用切面表达式;-->
                <aop:after-throwing method="afterThrowsPrintLog" pointcut-ref="pointCut">
    </aop:after-throwing>

                <!--最终通知，引用切面表达式-->
                <aop:after method="afterPrintLog" pointcut-ref="pointCut"></aop:after>
            </aop:aspect>
        </aop:config>


    </beans>
```

Spring 中的环绕通知

- 环绕通知

```java
@Override      整个的invoke方法在执行就是环绕通知
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

    if("test".equals(method.getName())){
        return method.invoke(accountService,args);
    }

    Object rtValue = null;
    try {
        //1.开启事务
        txManager.beginTransaction();                    前置通知
        //2.执行操作
        rtValue = method.invoke(accountService, args);   在环绕通知中有明确的切入点方法调用。
        //3.提交事务
        txManager.commit();                              后置通知
        //4.返回结果
        return rtValue;
    } catch (Exception e) {
        //5.回滚操作
        txManager.rollback();                            异常通知
        throw new RuntimeException(e);
    } finally {
        //6.释放连接
        txManager.release();                             最终通知
    }
}
```

> 这个是我们前面讲的基于动态代理的通知
>
> - 我们发现环绕通知就是这整个方法
> - 里面包含着

    - 前置通知
    - 方法调用
    - 后置通知
    - 异常通知
    - 最终通知

所以环绕通知在`Spring`中的地位非同一般

首先我们需要注意几件事：

1. 既然环绕通知包含了这些东西，那么也就代表着在`spring`中可以在环绕通知中配置其他的通知
2. 既然上面的动态代理图片中有明确的方法调用，所以在环绕通知中也应该进行方法调用，要不然就不会进行方法执行

掌握了以上几件事，我们开始敲代码

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


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


    <bean id="logger" class="com.bean.utils.Logger"></bean>

    <aop:config>

        <aop:pointcut id="pointCut" expression="execution(* com.bean.service.impl.*.*(..))">
</aop:pointcut>

        <aop:aspect id="logAdvice" ref="logger">
            <!--只配置了一个环绕通知，环绕通知的标签就是<aop:around></aop:around>-->
            <aop:around method="aroudnPrintLog" pointcut-ref="pointCut"></aop:around>
        </aop:aspect>
    </aop:config>


</beans>
```

- `Logger`

```java
package com.bean.utils;

import org.aspectj.lang.ProceedingJoinPoint;

```

```java
/**
 * 模拟用于记录日志的工具类，里面提供了公共代码
 */
public class Logger {

    /*
    * 我们刚才在事项里面说到
    *   1. 环绕通知必须要进行方法调用，否则方法不会执行
    *   2. 观看之前我们写的基于动态代理执行的方法，我们也可以进行其他四种通知的调用
    *   3. 我们也要有返回值
    *
    *    既然要进行方法调用，就要有参数
    *       ProceedingJoinPoint就是参数，用于获取方法
    *           - proceed()：参数下面有一个方法proceed()，这个就相当于明确调用切入点方法
    *           - getArgs()：用于获取切入点点方法的参数
    * */
    public Object aroudnPrintLog(ProceedingJoinPoint proceedingJoinPoint){

        Object returnValue = null;
        try {
            System.out.println("这叫前置通知");
            returnValue = proceedingJoinPoint.proceed(proceedingJoinPoint.getArgs());    //这叫切入点方法调用
            System.out.println("这叫后置通知");
        } catch (Throwable throwable) {//注意这里使用的是Throwable，因为Exception拦不住它
            System.out.println("这叫异常通知");
            throwable.printStackTrace();
        }finally {
            System.out.println("这叫最终通知");
        }
        return returnValue;
    }
}
```

AOP 中基于注解的配置

1. 首先同样的，需要更改一下新的约束条件

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

2. 加入 `bean` 对象，配置注解

- `servce`

```java
package com.bean.service.impl;

import com.bean.service.IAccountService;
import org.springframework.stereotype.Service;

@Service
public class AccountServiceImpl implements IAccountService {
    public void saveAccount() {
        System.out.println("保存方法执行了");
    }

    public void updateAccount(int i) {
        System.out.println("更新方法执行了"+i);
    }

    public int deleteAccount() {
        System.out.println("删除方法执行了");
        return 0;
    }
}
```

- `Logger`

```java
package com.bean.utils;

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

/**
 * 模拟用于记录日志的工具类，里面提供了公共代码
 */
@Component//加入bean
@Aspect//注意，这里制定了这个Logger类是一个切面类
public class Logger {


    @Pointcut("execution(* com.bean.service.impl.*.*(..))")//配置切面表达式,id就是方法名称
    public void AspectPointCut(){};

    /**
     * 前置通知
     */
    @Before("AspectPointCut()")//指定前置通知，注意看好包，配置了切面表达式，注意一定要加括号
    public void beforePrintLog(){
        System.out.println("Logger类中的前置通知方法开始记录日志了。。。。。。");
    }

    /**
     * 后置通知通知
     */
    @AfterReturning("AspectPointCut()")//指定后置通知，配置切面表达式，注意加括号
    public void afterReturningPrintLog(){
        System.out.println("Logger类中的后置通知开始记录日志了。。。。。。");
    }

    /**
```

```
     * 异常通知
     */
    @AfterThrowing("AspectPointCut()")//指定异常通知，配置切面表达式，注意加括号
    public void afterThrowsPrintLog(){
        System.out.println("Logger类中的异常通知方法开始记录日志了。。。。。。");
    }

    /**
     * 最终通知
     */
    @After("AspectPointCut()")//指定最终通知，配置切面表达式，注意加括号
    public void afterPrintLog(){
        System.out.println("Logger类中的最终通知方法开始记录日志了。。。。。。");
    }


    //环绕通知
    @Around("AspectPointCut()")//指定环绕通知，配置切面表达式，注意加括号
    public Object aroudnPrintLog(ProceedingJoinPoint proceedingJoinPoint){
        Object returnValue = null;
        try {
            System.out.println("这叫前置通知");
            returnValue = proceedingJoinPoint.proceed(proceedingJoinPoint.getArgs());   //这叫切入点方法调用
            System.out.println("这叫后置通知");
        } catch (Throwable throwable) {//注意这里使用的是Throwable，因为Exception拦不住它
            System.out.println("这叫异常通知");
            throwable.printStackTrace();
        }finally {
            System.out.println("这叫最终通知");
        }
        return returnValue;
    }

}
```

- AOPTest

```
package com.bean.test;

import com.bean.service.IAccountService;
import com.bean.service.impl.AccountServiceImpl;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AOPTest {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");

        IAccountService accountService  = (IAccountService) context.getBean("accountServiceImpl");

        accountService.saveAccount();
    }
}
```

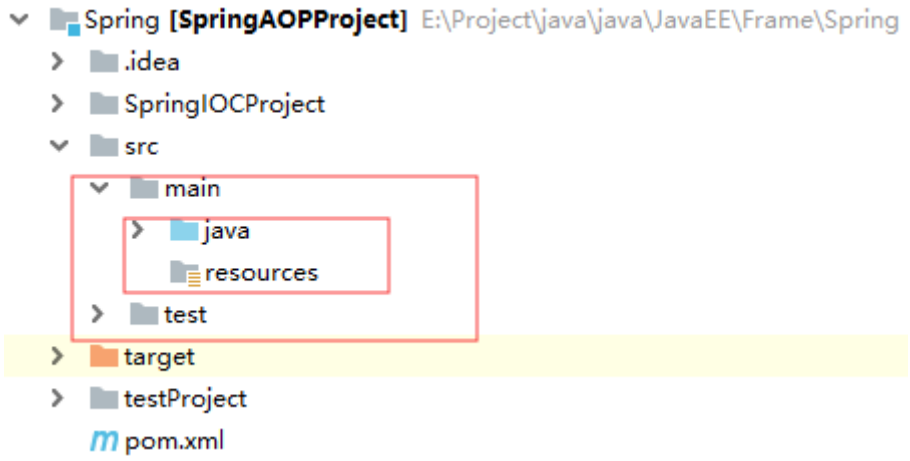
AOP 注解中的问题，实际开发中应该怎么做

> 其实虽然注解好用，但是我不得不告诉你，Spring 在完全使用注解方式执行 AOP 的时候会出现问题，就是顺序调用问题，比如下面的
>
> ```
> Logger类中的前置通知方法开始记录日志了。。。。。。
> 保存方法执行了
> Logger类中的最终通知方法开始记录日志了。。。。。。
> Logger类中的后置通知开始记录日志了。。。。。。
> ```
>
> - 这是不使用环绕通知的时候进行的测试，代码写的没有问题，但是调用顺序出错了，其实是 Spring 有问题
> - 调用顺序为：前置通知-->最终通知-->后置通知
> - 所以假如使用注解，可能会出现问题

> 所以在这个方法下，采用 半注解半代码的形式来配置（注解环绕通知） 或者 使用xml 的形式，便可以避免调用顺序出错的问题

纯注解

1. 首先看一下前面的不使用 xml 的时候使用的配置类
2. 然后在配置类上加上一个 @EnableAspectJAutoProxy 来配置好切面类
3. 下面是一个例子



- java.config.SpringConfiguration

```
package config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@ComponentScan("com.bean")
@EnableAspectJAutoProxy
public class SpringConfiguration {
}
```

- java.com.bean.service.impl.AccountServiceImpl

```java
package com.bean.service.impl;

import com.bean.service.IAccountService;
import org.springframework.stereotype.Service;

@Service
public class AccountServiceImpl implements IAccountService {
    public void saveAccount() {
        System.out.println("保存方法执行了");
    }

    public void updateAccount(int i) {
        System.out.println("更新方法执行了"+i);
    }

    public int deleteAccount() {
        System.out.println("删除方法执行了");
        return 0;
    }
}
```

- `java.com.bean.utils.Logger`

```java
package com.bean.utils;

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

/**
 * 模拟用于记录日志的工具类，里面提供了公共代码
 */
@Component//加入bean
@Aspect//注意，这里制定了这个Logger类是一个切面类
public class Logger {


    @Pointcut("execution(* com.bean.service.impl.*.*(..))")//配置切面表达式,id就是方法名称
    public void AspectPointCut(){};

    /**
     * 前置通知
     */
    @Before("AspectPointCut()")//指定前置通知，注意看好包，配置了切面表达式，注意一定要加括号
    public void beforePrintLog(){
        System.out.println("Logger类中的前置通知方法开始记录日志了。。。。。。");
    }

    /**
     * 后置通知通知
     */
    @AfterReturning("AspectPointCut()")//指定后置通知，配置切面表达式，注意加括号
    public void afterReturningPrintLog(){
        System.out.println("Logger类中的后置通知开始记录日志了。。。。。。");
    }

    /**
```

```
     * 异常通知
     */
    @AfterThrowing("AspectPointCut()")//指定异常通知，配置切面表达式，注意加括号
    public void afterThrowsPrintLog(){
        System.out.println("Logger类中的异常通知方法开始记录日志了。。。。。。");
    }

    /**
     * 最终通知
     */
    @After("AspectPointCut()")//指定最终通知，配置切面表达式，注意加括号
    public void afterPrintLog(){
        System.out.println("Logger类中的最终通知方法开始记录日志了。。。。。。");
    }


    //环绕通知
    @Around("AspectPointCut()")//指定环绕通知，配置切面表达式，注意加括号
    public Object aroudnPrintLog(ProceedingJoinPoint proceedingJoinPoint){
        Object returnValue = null;
        try {
            System.out.println("这叫前置通知");
            returnValue = proceedingJoinPoint.proceed(proceedingJoinPoint.getArgs());   //这叫切入点方法调用
            System.out.println("这叫后置通知");
        } catch (Throwable throwable) {//注意这里使用的是Throwable，因为Exception拦不住它
            System.out.println("这叫异常通知");
            throwable.printStackTrace();
        }finally {
            System.out.println("这叫最终通知");
        }
        return returnValue;
    }

}
```

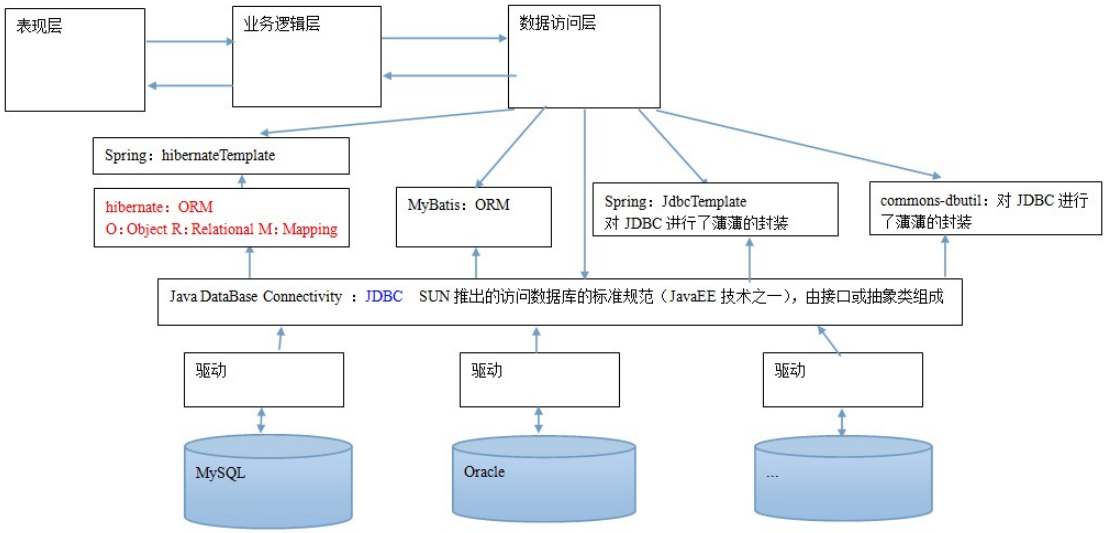
- `test.com.bean.test.AOPTest`

```
package com.bean.test;

import com.bean.service.IAccountService;
import com.bean.service.impl.AccountServiceImpl;
import config.SpringConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AOPTest {

    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfiguration.class);

        IAccountService accountService  = (IAccountService) context.getBean("accountServiceImpl");

        accountService.saveAccount();
    }
```

```
19    }
```

第四天

Spring中的JdbdTemplate



> 上面这张图是持久层总图，我们今天的主角是`JdbcTemplate`，可以看到对`JDBC`进行了薄薄封装

> 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。
>
> - 操作关系型数据的：
>   - `JdbcTemplate`
>   - `HibernateTemplate`
> - 操作 nosql 数据库的：
>   - `RedisTemplate`
> - 操作消息队列的：
>   - `JmsTemplate`
> - 我们今天的主角在`spring-jdbc-5.0.2.RELEASE.jar`中
> - 还需要导入一个`spring-tx-5.0.2.RELEASE.jar`（它是和事务相关的）。
> - 不可避免地要导入数据库驱动

JdbcTemplate的作用

> 与数据库进行交互，实现对表的`CRUD`

如何创建该对象

1. 依赖

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

    <groupId>com.bean</groupId>
    <artifactId>SpringAOP</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

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

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

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

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.6</version>
        </dependency>
    </dependencies>
</project>
```

2. 表

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

insert into account(name,money) values('aaa',1000);
insert into account(name,money) values('bbb',1000);
insert into account(name,money) values('ccc',1000);
```

3. 实体类

```java
package com.bean.domain;

import java.io.Serializable;

public class Account implements Serializable {
```

```
6
7       private Integer id;
8       private String name;
9       private Float money;
10
11      public Account() {
12      }
13
14      public Account(Integer id, String name, Float money) {
15          this.id = id;
16          this.name = name;
17          this.money = money;
18      }
19
20      public Integer getId() {
21          return id;
22      }
23
24      public void setId(Integer id) {
25          this.id = id;
26      }
27
28      public String getName() {
29          return name;
30      }
31
32      public void setName(String name) {
33          this.name = name;
34      }
35
36      public Float getMoney() {
37          return money;
38      }
39
40      public void setMoney(Float money) {
41          this.money = money;
42      }
43
44      @Override
45      public String toString() {
46          return "Account{" +
47                  "id=" + id +
48                  ", name='" + name + '\'' +
49                  ", money=" + money +
50                  '}';
51      }
52  }
```

4. `JdbcTemplate`的使用

```
1   package com.bean.jdbcTemplate;
2
3   import org.springframework.jdbc.core.JdbcTemplate;
4   import org.springframework.jdbc.datasource.DriverManagerDataSource;
5
6   public class JdbcTemplateDemo {
7       public static void main(String[] args) {
8
```

```
        //准备数据源：JdbcTemplate内置数据源
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/spring");
        dataSource.setUsername("root");
        dataSource.setPassword("root");

        //JdbcTemplate
        JdbcTemplate jdbcTemplate = new JdbcTemplate();

        //设置数据源
        jdbcTemplate.setDataSource(dataSource);

        jdbcTemplate.execute("select * from account");

    }
}
```

使用 Spring 创建

XML 配置

- bean.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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--配置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="url" value="jdbc:mysql://localhost:3306/spring"></property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
    </bean>

</beans>
```

- JdbcTemplateDemo

```
package com.bean.jdbcTemplate;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

public class JdbcTemplateDemo {
    public static void main(String[] args) {

```

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

        JdbcTemplate jdbcTemplate = (JdbcTemplate)context.getBean("jdbcTemplate");

        jdbcTemplate.execute("insert into account (name,money) values ('dd',1000)");

    }
}
```

注解配置

- `config.SpringConfiguration`

```
package config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.annotation.Resource;
import javax.sql.DataSource;

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

    @Bean
    public JdbcTemplate jdbcTemplateBean(){
        return new JdbcTemplate(dataSourceBean());
    }

    @Bean
    public DriverManagerDataSource dataSourceBean(){
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setUrl("jdbc:mysql://localhost:3306/spring");
        dataSource.setUsername("root");
        dataSource.setPassword("root");
        return dataSource;
    }


}
```

- `JdbcTemplateDemo`

```
package com.bean.jdbcTemplate;

import config.SpringConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

public class JdbcTemplateDemo {
```

```
    public static void main(String[] args) {

        ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfiguration.class);

        JdbcTemplate jdbcTemplate = (JdbcTemplate)context.getBean("jdbcTemplateBean");

        jdbcTemplate.execute("insert into account (name,money) values ('dd',1000)");

    }
}
```

> 注解不能和 `xml` 一起，否则报错，前面已经讲过了

JdbcTemplae 的 CRUD

- JdbcTemplateDemo

```
package com.bean.jdbcTemplate;

import com.bean.domain.Account;
import config.SpringConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import java.util.List;

public class JdbcTemplateDemo {
    public static void main(String[] args) {

        ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfiguration.class);

        JdbcTemplate jdbcTemplate = (JdbcTemplate)context.getBean("jdbcTemplateBean");

//        增加：使用update，没啥好说的的，注意类型是float
        jdbcTemplate.update("insert into account (name,money) values(?,?)","new Account",1000f);
//        删除：使用update，没啥好说的的，注意类型是float
        jdbcTemplate.update("delete from account where id=?",5);
//        更新：没啥好说的的，注意类型是float
        jdbcTemplate.update("update account set name=?,money=? where id=?","update test",100f,6);

//        查询
//        查询所有：查询所有，使用query方法，spring使用了这个BeanPropertyRowMapper封装进去了类型，然后直接封装进去
        List<Account> accounts = jdbcTemplate.query("select * from account", new BeanPropertyRowMapper<Account>(Account.class));
        for (Account account : accounts) {
            System.out.println(account);
```

```
        }

//          查询一个：这里也可以使用查询出List的，不过输出第0位就可以了，注意这里使用的构造函数中的最后一个参数为可变参数，只有在jdk1.5之后才能使用
        List<Account> account = jdbcTemplate.query("select * from account where id=?", new BeanPropertyRowMapper<Account>(Account.class), 7);
        System.out.println(account.isEmpty()?"无参数":account.get(0));

//          查询一行一列（聚合函数）：这里需要的就是一个类型，当查到的时候会自动转变为这个类型，不过一般推荐Long，因为假如int范围不够了呢
        //注意这里使用的是queryForObject
        jdbcTemplate.queryForObject("select count(id) from account",Long.class);
        //加参数：jdbcTemplate.queryForObject("select count(id) from account where money>?",Long.class,1500);

    }
}
```

> 下面是实际开发中的写法，结合 `Spring` 使用

基于 `XML` 的配置

- `bean.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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--首先配置dao-->
    <bean id="accountDaoImpl" class="com.bean.dao.impl.AccountDaoImpl">
        <property name="jdbcTemplate" ref="jdbcTemplateBean"></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://localhost:3306/spring"></property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
    </bean>

    <!--配置JdbcTemplate-->
    <bean id="jdbcTemplateBean" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"></property>
    </bean>


</beans>
```

- `Account`

```
package com.bean.domain;
```

```
2
3   import java.io.Serializable;
4
5   public class Account implements Serializable {
6
7       private Integer id;
8       private String name;
9       private Float money;
10
11      public Account() {
12      }
13
14      public Account(Integer id, String name, Float money) {
15          this.id = id;
16          this.name = name;
17          this.money = money;
18      }
19
20      public Integer getId() {
21          return id;
22      }
23
24      public void setId(Integer id) {
25          this.id = id;
26      }
27
28      public String getName() {
29          return name;
30      }
31
32      public void setName(String name) {
33          this.name = name;
34      }
35
36      public Float getMoney() {
37          return money;
38      }
39
40      public void setMoney(Float money) {
41          this.money = money;
42      }
43
44      @Override
45      public String toString() {
46          return "Account{" +
47                  "id=" + id +
48                  ", name='" + name + '\'' +
49                  ", money=" + money +
50                  '}';
51      }
52  }
```

- `AccountDaoImpl`

```
1   package com.bean.dao.impl;
2
3   import com.bean.dao.IAccountDao;
4   import com.bean.domain.Account;
```

```
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import java.util.List;

public class AccountDaoImpl implements IAccountDao {

    private JdbcTemplate jdbcTemplate;

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

    public Account findById(Integer id) {

        List<Account> accounts = jdbcTemplate.query("select * from account where id=?", new BeanPropertyRowMapper<Account>(Account.class), id);

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

    public Account findByName(String name) {

        List<Account> accounts = jdbcTemplate.query("select * form account where name=?", new BeanPropertyRowMapper<Account>(Account.class), name);

        //假如没有查到，返回null
        if (accounts.isEmpty()){
            return null;
        }
        //假如有多个，返回异常
        if (accounts.size()>1)
        {
            throw new RuntimeException("结果不唯一");
        }
        //否则返回唯一一个
        return accounts.get(0);
    }

    public void updateAccount(Account account) {
        jdbcTemplate.update("update account set name=?,money=?",account.getName(),account.getMoney());
    }
}
```

- `JdbcTemplcateDemo`

```
package com.bean.jdbcTemplate;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.domain.Account;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class JdbcTemplateDemo {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
```

```
        IAccountDao accountDao = (IAccountDao) context.getBean("accountDaoImpl");
        System.out.println(accountDao.findById(7));
    }
}
```

基于注解的配置

- `SpringConfiguration`

```
package config;

import com.bean.dao.impl.AccountDaoImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

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

//    声明dao
    @Bean
    public AccountDaoImpl accountDao(){
        return new AccountDaoImpl();
    }
//    声明JdbcTemplate
    @Bean
    public JdbcTemplate jdbcTemplate(){
        return new JdbcTemplate(dataSource());
    }
//    声明数据源
    @Bean
    public DriverManagerDataSource dataSource(){
        DriverManagerDataSource driverManagerDataSource = new DriverManagerDataSource();
        driverManagerDataSource.setDriverClassName("com.mysql.jdbc.Driver");
        driverManagerDataSource.setUrl("jdbc:mysql://localhost:3306/spring");
        driverManagerDataSource.setUsername("root");
        driverManagerDataSource.setPassword("root");
        return driverManagerDataSource;
    }

}
```

- `Account`

```
package com.bean.domain;

import java.io.Serializable;

public class Account implements Serializable {

    private Integer id;
    private String name;
    private Float money;

```

```
    public Account() {
    }

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

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

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

- `AccountDaoImpl`

```
package com.bean.dao.impl;

import com.bean.dao.IAccountDao;
import com.bean.domain.Account;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class AccountDaoImpl implements IAccountDao {
```

```
    @Autowired
    private JdbcTemplate jdbcTemplate;

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

    public Account findById(Integer id) {

        List<Account> accounts = jdbcTemplate.query("select * from account where id=?", new BeanPropertyRowMapper<Account>(Account.class), id);

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

    public Account findByName(String name) {

        List<Account> accounts = jdbcTemplate.query("select * form account where name=?", new BeanPropertyRowMapper<Account>(Account.class), name);

        //假如没有查到，返回null
        if (accounts.isEmpty()){
            return null;
        }
        //假如有多个，返回异常
        if (accounts.size()>1)
        {
            throw new RuntimeException("结果不唯一");
        }
        //否则返回唯一一个
        return accounts.get(0);
    }

    public void updateAccount(Account account) {
        jdbcTemplate.update("update account set name=?,money=?",account.getName(),account.getMoney());
    }
}
```

- JdbcTemplateDemo

```
package com.bean.jdbcTemplate;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.domain.Account;
import config.SpringConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class JdbcTemplateDemo {
    public static void main(String[] args) {
//        ApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
        ApplicationContext context = new AnnotationConfigApplicationContext(SpringConfiguration.class);
        IAccountDao accountDao = (IAccountDao) context.getBean("accountDaoImpl");
        System.out.println(accountDao.findById(7));
```

```
        }
    }
```

Spring中的事务控制

Spring控制我们需要明确的

1. javaEE体系分层开发，事务处理位于业务层，Spring提供了分层设计业务层的事务处理解决方案
2. Spring框架为我们提供了一组事务控制的接口，具体后面介绍，这组接口是在spring-tx-5.0.2.RELEASSE.jar中
3. Spring中的事务控制都是基于AOP的，它既可以使用编程的方式进行实现，也可以使用配置的方式进行实现

Spring中事务控制的API介绍

用之前需要先导jar包

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

事务管理器

- PlatformTransacionManager管理器，这是个事务接口，包含三个具体操作
  - 获取事务状态信息：TransactionStatus getTransaction(TransactionDefinition definition)
  - 提交事务：void commit(TransactionStatus status)
  - 回滚事务：void rollback(TransactionStatus status)

- 实现类：
  - DataSourceTransactionManager
    - 包：org.springframework.jdbc.datasource.DataSourceTransactionManager
    - 使用Spring JDBC或者iBatis进行持久化数据时使用
  - HibernateTransactionManager
    - 包：org.springframework.orm.hibernate5.HibernateTransactionManager
    - 使用Hibernate版本进行持久化数据时使用

事务的定义信息

- TransactionDefinition：事物的定义信息对象，里面有如下方法
  - String getName()：获取事务对象名称

- `int getIsolationLevel()`：获取事务隔离级别
  - `ISOLATION_DEFAULT`：默认级别，归属下列某一种
  - `ISOLATION_READ_UNCOMMITTED`：可以读取未提交数据
  - `ISOLATION_READ_COMMITTED`：只能读取已提交数据，解决脏读问题（Oracle默认级别）
  - `ISOLATION_REPEATABLE_READ`：是否读取其他事务提交修改后的数据，解决不可重复度的问题（MYSQL默认）
  - `ISOLATION_SERIALIZABLE`：是否读取其他事务提交添加后的数据，解决幻影读问题
- `int getPropagetionBehavior()`：获取事务传播行为
  - `REQUIRED`：如果当前没有事务，就新建一个事务，如果已经存在一个事务，加入到这个事务（默认），增删改的事务
  - `SUPPORTS`：支持当前事务，加入当前没有事务，就以非事务方式执行（没有事务），只有查询才能用的事务
  - `MANDATORY`：使用当前事务，如果当前没有事务，抛出异常
  - `REQUERS_NEW`：新建事务，如果当前在事务中，就把当前事务挂起
  - `NOT_SUPPORTED`：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起
  - `NEVER`：以非事务方式运行，加入当前存在事务，抛出异常
  - `NESTED`：如果当前存在事务，则在嵌套事务内执行；如果当前没有事务，则执行`REQUIRED`类似的操作
- `int getTimeout()`：获取事务超时时间：默认值为-1（没有超时时间）。假如有，则以秒为单位进行设置
- `boolean isReadOnly()`：获取事务是否只读：建议查询时设置为只读

事务的状态信息

- `TransactionStatus`接口描述了某个时间点上事务的状态信息，包含有六个基本操作
  - 刷新事务：`void flush()`
  - 获取是否是存在存储点：`boolean hasSavepoint()`

    > 事务是以节点来提交的，每一个节点都是一个储存点，当事务回滚的时候，只会回滚到上个节点，而不是从头来
  - 获取事务是否完成：`boolean isCompleted()`
  - 获取事务是否为新的事务：`boolean isNewTransaction()`
  - 获取事务是否回滚：`boolean isRollbackOnly()`
  - 设置事务回滚：`void setRollbackOnly()`

事务控制的代码准备

- `pom.xml`

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

```

```
    <groupId>com.bean</groupId>
    <artifactId>SpringAOP</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

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

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

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

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.6</version>
        </dependency>

        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.8.7</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>

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

基于 XML 的事务控制

> Spring 中基于 XML 的声明式事务控制配置步骤

1. 配置事务管理器
    1. 注入 `DataSource`
2. 配置事务通知
    1. 导入事务的约束： `tx` 和 `aop` 的名称空间和约束

    ```
    <?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:tx="http://www.springframework.org/schema/tx"
     xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         https://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/tx
         https://www.springframework.org/schema/tx/spring-tx.xsd
         http://www.springframework.org/schema/aop
         https://www.springframework.org/schema/aop/spring-aop.xsd">
    ```

    2. 配置事务管理器的 `bean` 对象

    > 包为：
    > `org.springframework.jdbc.datasource.DataSourceTransactionManager`

    3. 使用标签 `<tx:advice></tx:advice>` 标签配置事务通知

    > - `id` ：配置唯一标识
    > - `transaction-manager` ：事务管理器

    4. 在 `<tx:advice></tx:advice>` 内部配置事务的属性
        - `<tx:attributes></tx:attributes>`
            - `<tx:method>`
                - `name` 指定 `service` 中的方法
                - `isolation`

                > 用于指定事物的隔离级别，默认值为 `DEFAULT` ，表示使用数据库的默认隔离级别

                - `propagetion`

                > - 用于指定事物的传播行为，默认值为 `REQUIRED` ，表示一定会有事务，这是增删改的选择
                > - 查询可以选择为 `SUPPORTS`

                - `read-only`

                > 指定事物是否只读，只有查询方法才能设置为 `true` ，默认值为 `false` ，表示读写

- `timeout`

  > 用于指定事物的超时时间，默认值为-1，表示永不超时，如果指定数值则以秒为单位

- `rollback-for`

  > 用于指定一个异常，碰到这个异常则回滚，其他异常不会滚。
  >
  > 没有默认值，表示任何异常都回滚

- `no-rollback-for`

  > 用于指定一个异常，碰到这个异常不回滚，其他异常都回滚。
  >
  > 没有默认值，表示任何异常都回滚

3. 配置`AOP`的通用切入点表达式
4. 建立事务通知和切入点表达式的对应关系

> 使用`<aop:advisor></aop:advicor>`

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

    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="jdbc:mysql://localhost:3306/spring"></property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
    </bean>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <bean id="accountService" class="com.bean.service.impl.AccountServiceImpl">
        <property name="accountDao" ref="accountDao"></property>
    </bean>

    <bean id="accountDao" class="com.bean.dao.impl.AccountDaoImpl">
        <property name="jdbcTemplate" ref="jdbcTemplate"></property>
    </bean>
```

```
31
32      <!--上面是配的bean对象，下面开始编写事务-->
33
34      <!--首先要有一个事务管理器，包为：
    org.springframework.jdbc.datasource.DataSourceTransactionManager
35          事务管理器内要配置dataSource
36      -->
37      <bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
38          <property name="dataSource" ref="dataSource"></property>
39      </bean>
40
41      <!--现在配置事务的状态
42          id 为唯一标识
43          transaction-manager为配置事务管理器
44      -->
45      <tx:advice id="txAdvice" transaction-manager="transactionManager">
46          <!--配置事务属性-->
47          <tx:attributes>
48              <!--
49                  <tx:method></tx:method>
50                      name：要配置的方法名称，比如我们这里用AOP配置的全都是service.impl包下的，所以
51                          * ：匹配全部的方法（虽然也就那么一个transer方法）
52                          find*：将来假如有方法为find开头的时候，匹配折现方法
53                          优先级：find* 高于 *
54                      isolation：指定事物的隔离级别，默认值为DEFAULT，表示数据库的默认隔离级别
55                      read-only：是否只读
56                      propagation：指定事物的传播行为，默认为REQUIRED，为增删改的，查询可以为SUPPORTS
57                      timeout：指定超时时间
58                      rollback-for：用于指定一个异常，发生了此异常则回滚，其他异常不回滚。默认值无，表示任
    何异常都回滚。
59                      no-rollback-for：指定一个异常，发生此异常不回滚，其余异常全都回滚。默认值无，表示任
    何异常都回滚。
60              -->
61              <tx:method name="*" read-only="false" propagation="REQUIRED"/>
62              <tx:method name="find*" read-only="true" propagation="SUPPORTS"></tx:method>
63          </tx:attributes>
64      </tx:advice>
65
66
67      <aop:config>
68          <aop:pointcut id="pointCut" expression="execution(* com.bean.service.impl.*.*(..))">
    </aop:pointcut>
69          <aop:advisor advice-ref="txAdvice" pointcut-ref="pointCut"></aop:advisor>
70      </aop:config>
71
72  </beans>
```

- `AccountDaoImpl`

```
1   package com.bean.dao.impl;
2
3   import com.bean.dao.IAccountDao;
4   import com.bean.domain.Account;
5   import org.springframework.jdbc.core.BeanPropertyRowMapper;
6   import org.springframework.jdbc.core.JdbcTemplate;
7
8   import java.util.List;
```

```
9
10  public class AccountDaoImpl implements IAccountDao {
11
12      JdbcTemplate jdbcTemplate;
13
14      public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
15          this.jdbcTemplate = jdbcTemplate;
16      }
17
18      public Account findById(Integer id) {
19          List<Account> accounts = jdbcTemplate.query("select * from account where id=?",new
    BeanPropertyRowMapper<Account>(Account.class),id);
20          return accounts.isEmpty()?null:accounts.get(0);
21      }
22
23      public void updateAccount(Account account) {
24          jdbcTemplate.update("update account set name=?,money=? where
    id=?",account.getName(),account.getMoney(),account.getId());
25      }
26  }
```

- `AccountServiceImpl`

```
1   package com.bean.service.impl;
2
3   import com.bean.dao.IAccountDao;
4   import com.bean.dao.impl.AccountDaoImpl;
5   import com.bean.domain.Account;
6   import com.bean.service.IAccountService;
7   import org.springframework.context.ApplicationContext;
8   import org.springframework.context.support.ClassPathXmlApplicationContext;
9
10  public class AccountServiceImpl implements IAccountService {
11
12      private IAccountDao accountDao;
13
14      public void setAccountDao(IAccountDao accountDao) {
15          this.accountDao = accountDao;
16      }
17
18      public void transfer(Integer Transferer, Integer Payee, Float money) {
19          System.out.println("transer");
20
21          Account accountMinus = accountDao.findById(1);
22
23          Account accountAdd = accountDao.findById(2);
24
25          accountMinus.setMoney(accountMinus.getMoney()-money);
26
27          accountAdd.setMoney(accountAdd.getMoney()+money);
28
29          accountDao.updateAccount(accountMinus);
30  //        int i = 1/0;
31          accountDao.updateAccount(accountAdd);
32      }
33  }
```

- 测试

```java
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.service.IAccountService;
import com.bean.service.impl.AccountServiceImpl;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.test.context.junit.jupiter.SpringJUnitConfig;

@SpringJUnitConfig
public class TransactionTest {
    @Test
    public void Test(){

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

        IAccountService accountService = (IAccountService)
context.getBean("accountService");

        accountService.transfer(1,2,100F);
    }
}
```

- 效果

> 效果就是：
>
> - 当没有报错时：转账正常执行
> - 当出现错误时：回滚

基于注解的事务控制

1. 导入关于 `context` 的名称空间

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

2. 配置事务管理器
3. 开启 `spring` 对注解事务的支持，使用 `<tx:annotation-drivern/>`
4. 在需要事务支持的时候使用 `@Transaction` 注解

- `bean.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"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         https://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/tx
         https://www.springframework.org/schema/tx/spring-tx.xsd
         http://www.springframework.org/schema/aop
         https://www.springframework.org/schema/aop/spring-aop.xsd
         http://www.springframework.org/schema/context
         https://www.springframework.org/schema/context/spring-context.xsd">

    <!--首先配置一下spring创建容器时要扫描的包-->
    <context:component-scan base-package="com.bean"></context:component-scan>


    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="jdbc:mysql://localhost:3306/spring"></property>
        <property name="username" value="root"></property>
        <property name="password" value="root"></property>
    </bean>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!--
        使用注解进行事务控制的配置
        1. 配置事务管理器
        2. 开启spring对注解事务的支持：使用<tx:annotation-driven/>,里面配置事务管理器
        3. 在需要事务支持的时候使用@Transaction注解
    -->


    <!--事务管理器还得留下-->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!--开启spring对注解事务的支持-->
    <tx:annotation-driven transaction-manager="transactionManager"></tx:annotation-driven>

</beans>
```

- `AccountDaoImpl`

```java
package com.bean.dao.impl;

import com.bean.dao.IAccountDao;
import com.bean.domain.Account;
import org.springframework.beans.factory.annotation.Autowired;
```

```java
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;

import java.util.List;

@Component
public class AccountDaoImpl implements IAccountDao {

    @Autowired
    JdbcTemplate jdbcTemplate;

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

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

- `AccountServiceImpl`

```java
package com.bean.service.impl;

import com.bean.dao.IAccountDao;
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.domain.Account;
import com.bean.service.IAccountService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@Service
@Transactional(propagation = Propagation.SUPPORTS,readOnly = true)  //这里是配置事务控制的注解的，配置的全局为只读型的配置
public class AccountServiceImpl implements IAccountService {

    @Autowired
    private IAccountDao accountDao;

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


    //这里单独配置，配置可写型的配置
    @Transactional(propagation = Propagation.REQUIRED,readOnly = false)
    public void transfer(Integer Transferer, Integer Payee, Float money) {
        System.out.println("transer");

```

```
        Account accountMinus = accountDao.findById(1);

        Account accountAdd = accountDao.findById(2);

        accountMinus.setMoney(accountMinus.getMoney()-money);

        accountAdd.setMoney(accountAdd.getMoney()+money);

        accountDao.updateAccount(accountMinus);
//        int i = 1/0;
        accountDao.updateAccount(accountAdd);
    }
}
```

- 测试

```
import com.bean.dao.impl.AccountDaoImpl;
import com.bean.service.IAccountService;
import com.bean.service.impl.AccountServiceImpl;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringJUnitConfig;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:bean.xml")
public class TransactionTest {

    @Autowired
    private  IAccountService accountService;

    @Test
    public void Test(){
        accountService.transfer(1,2,100F);
    }
}
```

Spring5 新特性介绍

> spring5.0 在 2017-9 发布了 GA(通用) 版本，该版本基于 jdk8 编写的，所以 jdk8 之前的版本无法使用
>
> 同时 tomcat 要求 8.5 以上
>
> 我们使用 jdk8 构建工程，可以降低版本编译，但是不能使用 jdk8 以下版本构建工程
>
> IDE 同时需要更新， eclipse 4.7.2 之前的就甭想了

> https://www.bilibili.com/video/av47952931?p=82