Spring 框架

01. Spring概述

Spring是一个开源框架，Spring是于2003 年兴起的一个**轻量级的Java 开发框架**，由Rod Johnson 在其著作Expert One-On-One J2EE Development and Design中阐述的部分理念和原型衍生而来。**它是为了解决企业应用开发的复杂性而创建的**。Spring使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益

简单来说，Spring是一个轻量级的**控制反转（IoC）**和**面向切面（AOP）**的容器框架。

1.1 spring的优势

1. 方便解耦，简化开发:

Spring就是一个大工厂，专门负责生成Bean，可以将所有对象创建和依赖关系维护由Spring管理

2. AOP编程的支持:

   Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能

3. 声明式事务的支持:

   只需要通过配置就可以完成对事务的管理，而无需手动编程

4. 方便程序的测试:

   Spring对Junit4支持，可以通过注解方便的测试Spring程序

5. 方便集成各种优秀框架:

   Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz等）的支持

6. 降低JavaEE API的使用难度

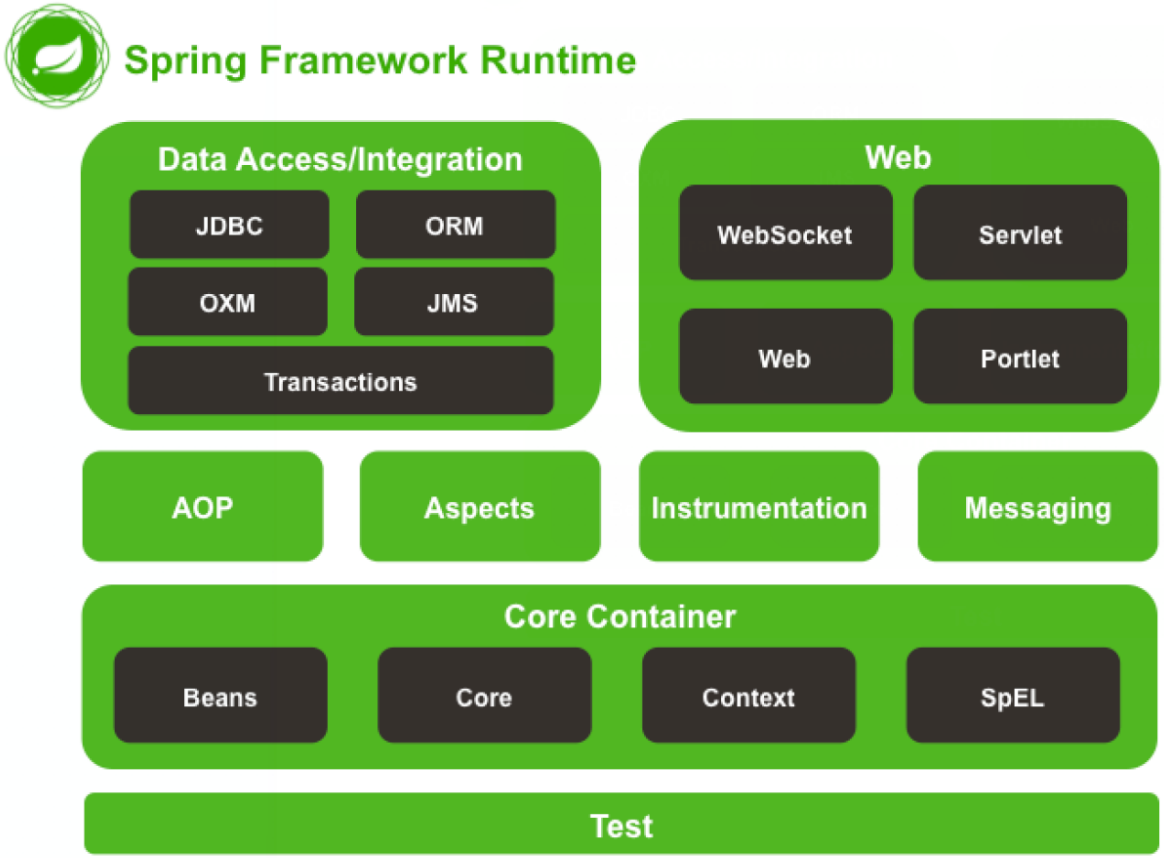
   Spring对JavaEE开发中一些难用的API（JDBC、JavaMail、远程调webservice用等），都提供了封装，使这些API应用难度大大降低

7. 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。它的源代码无意是Java技术的最佳实践的范例

1.2 spring的体系结构

Spring 框架是一个分层架构,,它包含一系列的功能要素并被分为大约**20个模块**。这些模块分为Core Container、Data Access/Integration、Web、AOP（Aspect Oriented Programming)、Instrumentation和测试部分,如下图所示：



- **Core Container(核心容器)**

  Spring 的核心容器是其他模块建立的基础，它主要由Bean模块、Core模块、Context模块、和SpEL（Spring Expression Language,spring 表达式语言）模块组成，具体如下：

  - Bean 模块：提供了BeanFactory，是工厂模式的经典实现，spring将管理对象称为bean。
  - Core 核心模块：提供了Spring 框架的基本组成模块，包括IoC和DI功能。
  - Context 上下文模块：建立在Bean和 core模块的基础上，它是访问定义和配置的任何对象的媒介。其中ApplicationContext 接口是上下文模块的焦点。
  - SpEL 模块：是Spring 3.0 后新增的模块，它提供了Spring Expression Language 支持，是运行时查询和操作对象图的强大的表达式语言。
- Data Access/Integration(数据访问/集成)
  - JDBC 模块：提供了一个JDBC的抽象层，大幅度的减少了开发过程中对数据库操作的编码。
  - ORM 模块：对流行的对象关系映射API，包括JPA、JDO和Hiberate提供了集成支持
  - Transactions 模块：支持对实现特殊接口以及所有的POJO类的编程和声明式的事务管理
- Web Spring的Web层包括WebSocket、Servlet、Web和Portlet模块，具体如下：
  - WebSocket 模块：Spring 4.0以后增加的模块，它提供了WebSocket 和SockJS的实现以及对STOMP的实现
  - Servlet模块：也称为Spring-webMVC模块，它包含了Spring的模型——视图——控制器（MVC）和REST Web Services实现的web应用程序。
  - Web 模块：提供了基本的Web开发集成特性，例如：多文件上传功能、使用Servlet监听器来初始化IoC容器以及Web应用上下文。
  - Protlet模块：提供了再Protlet环境中实现MVC实现，类似Servlet模块的功能。
- 其他模块
  - AOP模块：提供了面向切面编程实现，允许定义方法拦截器和切入点，将代码按照功能进行分离，以降低耦合性。

- Aspects 模块：提供了与AspectJ的集成功能，AspectJ是一个功能强大且成熟的面向切面编程（AOP）框架。
- Instrumentation模块：提供了类工具的支持和类加载器的实现，可以在特定的应用服务器中使用。
- Messaging模块：Spring 4.0以后新增的模块，它提供了消息传递体系结构和协议的支持。
- Test模块：提供了对单元测试和集成的支持。

02. IoC的概念和作用

2.1 程序的耦合和解耦

- 什么是程序的耦合

  耦合性(Coupling)，也叫耦合度，是对模块间关联程度的度量。耦合的强弱取决于模块间接口的复杂性、调用模块的方式以及通过界面传送数据的多少。模块间的耦合度是指模块之间的依赖关系，包括控制关系、调用关系、数据传递关系。模块间联系越多，其耦合性越强，同时表明其独立性越差(降低耦合性，可以提高其独立性)

  > 我们在开发中，有些依赖关系是必须的，有些依赖关系可以通过优化代码来解除的。
  >
  > 请看下面的示例代码：

```
/**
  * 账户的业务层实现类
 */
public class AccountServiceImpl implements IAccountService {
   private IAccountDao accountDao = new AccountDaoImpl();
}
//上面的代码表示：  业务层调用持久层，并且此时业务层在依赖持久层的接口和实现类。如果此时没有持久层实现类，编译将不能通过。这种编译期依赖关系，应该在我们开发中杜绝。我们需要优化代码解决。
```

  早期我们的JDBC操作，注册驱动时，我们为什么不使用DriverManager的register方法，而是采用Class.forName的方式?

```
public static void main(String[] args) {
        try {
            //1.注册驱动
             DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver());
            //Class.forName("com.mysql.jdbc.Driver");
            //2.获取连接
            //3.获取预处理sql语句对象
            //4.获取结果集
            //5.遍历结果集
        }
 }
```

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

2.2 解决程序耦合的思路

```
//当是我们讲解jdbc时，是通过反射来注册驱动的，代码如下：
Class.forName("com.mysql.jdbc.Driver");//此处只是一个字符串
```

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

同时，也产生了一个新的问题，mysql驱动的全限定类名字符串是在java类中写死的，一旦要改还是要修改源码。

解决这个问题也很简单，使用配置文件配置。

2.3 工厂模式解耦

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

那么，这个读取配置文件，创建和获取三层对象的类就是工厂。

下面模拟spring boot的工厂生成bean的方式，**解耦**

- 2.3.1 新建一个bean.properties

```
accountService=com.dgut.Service.impl.AccountServiceImpl
accountDao=com.dgut.dao.impl.AccountDaoImpl
```

- 2.3.2 新建一个工厂类

```
/**
 * 一个创建Bean对象的工厂
 *
 * Bean：在计算机英语中，有可重用组件的含义。
 * JavaBean：用java语言编写的可重用组件。
 *       javabean >  实体类
 *
 *   它就是创建我们的service和dao对象的。
 *
 *   第一个：需要一个配置文件来配置我们的service和dao
 *           配置的内容：唯一标识=全限定类名（key=value)
 *   第二个：通过读取配置文件中配置的内容，反射创建对象
 *
 *   我的配置文件可以是xml也可以是properties
 */
public class BeanFactory {
```

```
    //定义一个Properties对象
    private static Properties props;

    //定义一个Map,用于存放我们要创建的对象。我们把它称之为容器
    private static Map<String,Object> beans;

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

    /**
     * 根据bean的名称获取对象
     * @param beanName
     * @return
     */
    public static Object getBean(String beanName){
        return beans.get(beanName);
    }
}
```

- 2.3.3 测试类

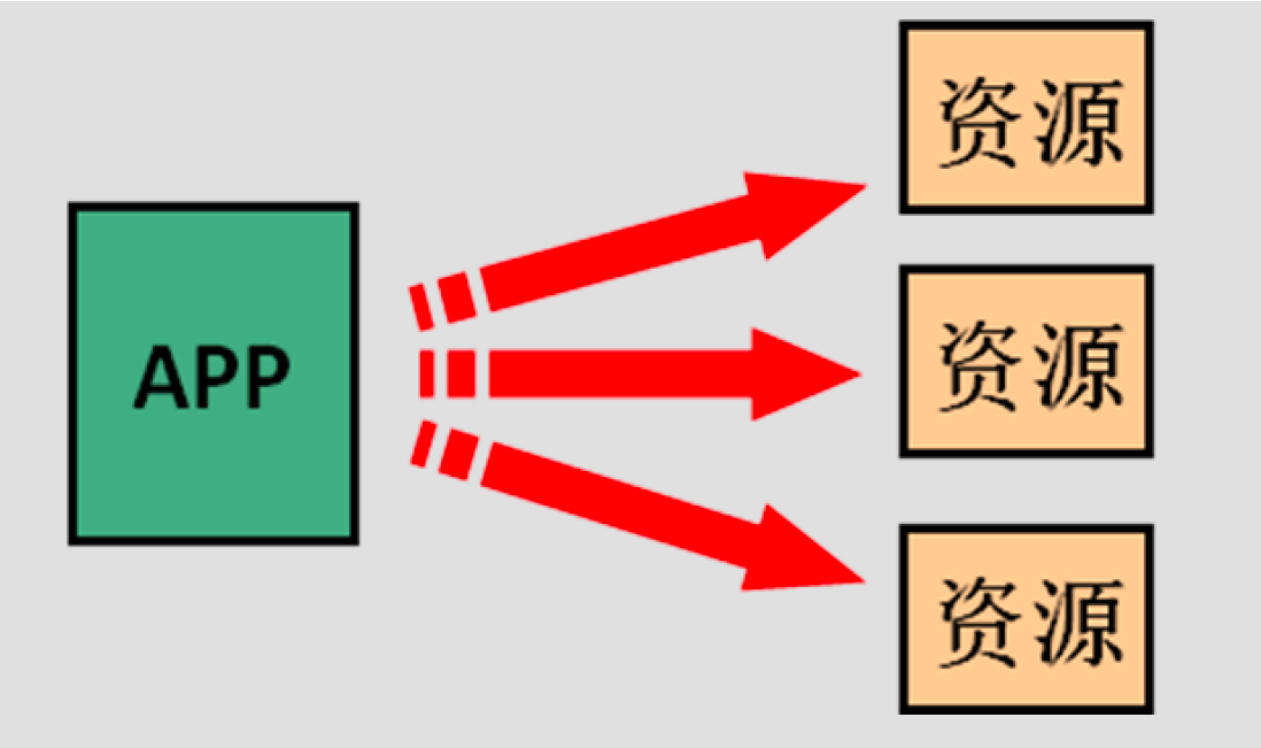
```
public static void main(String[] args) {
    for(int i=0;i<5;i++) {
        IAccountService as = (IAccountService)
BeanFactory.getBean("accountService");
        System.out.println(as);
    }
}
```

2.4 控制反转-IOC

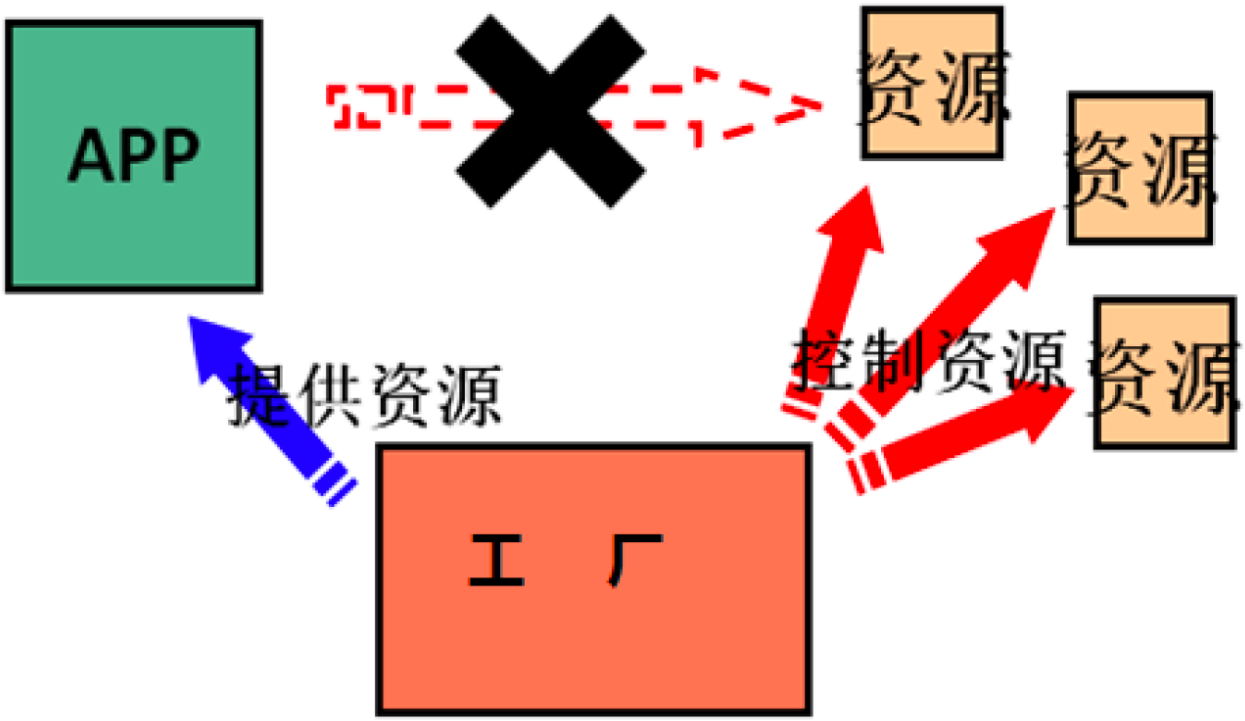
还是没解释什么是工厂?

工厂就是负责给我们从容器中获取指定对象的类。这时候我们获取对象的方式发生了改变。 Spring就是我们的工厂角色

原来： 我们在获取对象时，都是采用new的方式。是**主动**的。



现在： 我们获取对象时，同时跟工厂要，有工厂为我们查找或者创建对象。是**被动**的。



这种被动接收的方式获取对象的思想就是控制反转，它是spring框架的核心之一。

明确**IOC**的作用： 削减计算机程序的耦合(解除我们代码中的依赖关系)。

03.使用spring的IOC解决程序耦合

3.1 环境搭建

3.1.1 pom文件引入依赖

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```

3.1.2 创建业务层接口和实现类

```
public interface IAccountService {
    /**
     * 保存账户（此处只是模拟，并不是真的要保存）
     */
    void saveAccount();
}

public class AccountServiceImpl implements IAccountService {
```

```
    private IAccountDao accountDao = new AccountDaoImpl();//此处的依赖关系有待解决

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

3.1.3 创建持久层接口和实现类

```
public interface IAccountDao {
    /**
     * 保存账户
     */
    void saveAccount();
}
public class AccountDaoImpl implements IAccountDao {
    public void saveAccount() {
        System.out.println("保存了账户");
    }
}
```

3.2 基于XML的配置

3.2.1 在类的根路径下创建一个任意名称的xml文件

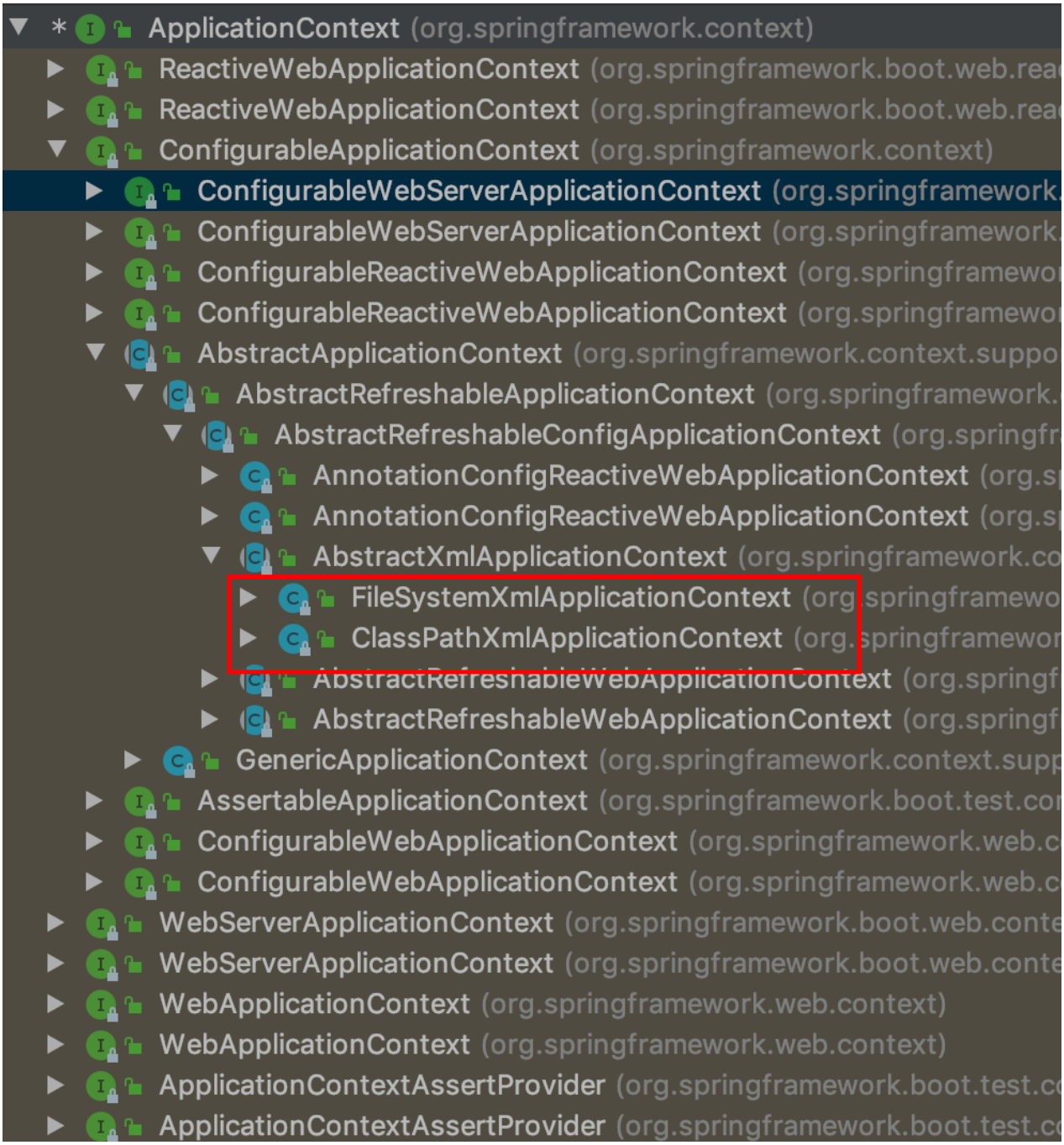
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- bean标签：用于配置让spring创建对象，并且存入ioc容器之中 id属性：对象的唯一标识。
class属性：指定要创建对象的全限定类名 -->
    <!-- 配置service -->
    <bean id="accountService"
class="com.dgut.Service.impl.AccountServiceImpl"></bean>
    <!-- 配置dao -->
    <bean id="accountDao" class="com.dgut.dao.impl.AccountDaoImpl"></bean>
</beans>
```

3.2.2 编写测试类

```
public class SpringTest {
    public static void main(String[] args) {
        ApplicationContext ac = new
ClassPathXmlApplicationContext("beans.xml");
        //2.根据bean的id获取对象
        IAccountService aService = (IAccountService)
ac.getBean("accountService");
        System.out.println(aService);
        IAccountDao aDao = (IAccountDao) ac.getBean("accountDao");
        System.out.println(aDao);
    }
}
```

3.3 ApplicationContext 类图



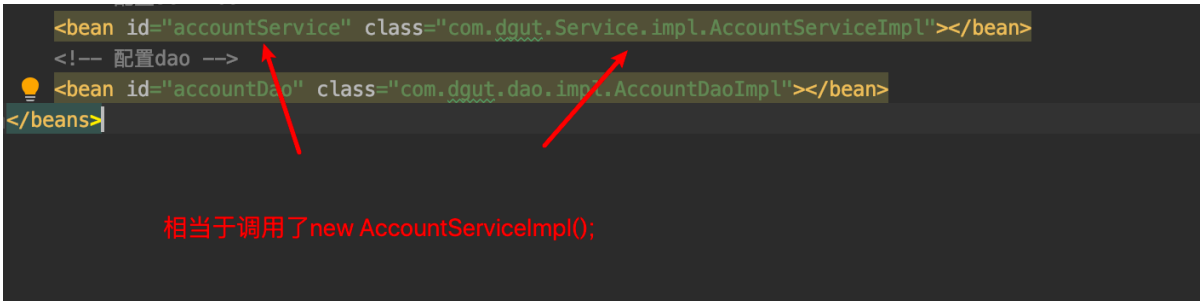
```
ClassPathXmlApplicationContext：
它是从类的根路径下加载配置文件 推荐使用这种
FileSystemXmlApplicationContext：
它是从磁盘路径上加载配置文件，配置文件可以在磁盘的任意位置。
AnnotationConfigApplicationContext：
当我们使用注解配置容器对象时，需要使用此类来创建spring 容器。它用来读取注解。
```

3.4 IOC中bean标签和管理对象

3.4.1 bean标签

1. 作用： 用于配置对象让spring来创建的。

   默认情况下它调用的是类中的无参构造函数。如果没有无参构造函数则不能创建成功。



2. 属性：

   id：给对象在容器中提供一个唯一标识。用于获取对象。

   class：指定类的全限定类名。用于反射创建对象。默认情况下调用无参构造函数。

   scope：指定对象的作用范围。

   - singleton :默认值，单例的.
   - prototype :多例的.
   - request :WEB项目中,Spring创建一个Bean的对象,将对象存入到request域中.
   - session :WEB项目中,Spring创建一个Bean的对象,将对象存入到session域中.
   - global session :WEB项目中,应用在Portlet环境.如果没有Portlet环境那么globalSession相当于session.

   init-method：指定类中的初始化方法名称。

   destroy-method：指定类中销毁方法名称。

3.4.2 bean的生命周期

1. 单例对象：scope="singleton"

一个应用只有一个对象的实例。它的作用范围就是整个引用。

生命周期：

对象出生：当应用加载，创建容器时，对象就被创建了。

对象活着：只要容器在，对象一直活着。

对象死亡：当应用卸载，销毁容器时，对象就被销毁了。

2. 多例对象：scope="prototype"

   每次访问对象时，都会重新创建对象实例。

   生命周期：

   对象出生：当使用对象时，创建新的对象实例。

   对象活着：只要对象在使用中，就一直活着。

   对象死亡：当对象长时间不用时，被java的垃圾回收器回收了。

```
    // AccountServiceImpl.java
    public void init(){
            System.out.println("init方法调用");
       }

       public void destory(){
           System.out.println("destory方法调用");
       }

   //对应的配置
     <bean id="accountService"
class="com.dgut.Service.impl.AccountServiceImpl" init-method="init" destroy-
method="destory"></bean>

       //测试
        public static void main(String[] args) {
           ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("beans.xml");
           //2.根据bean的id获取对象
           IAccountService aService = (IAccountService)
ac.getBean("accountService");
           System.out.println(aService);
           IAccountDao aDao = (IAccountDao) ac.getBean("accountDao");
           System.out.println(aDao);
           ac.close();
       }
```

3.4.3 实例化Bean的三种方式

第一种方式：使用默认无参构造函数

```
<!--在默认情况下：  它会根据默认无参构造函数来创建类对象。如果bean中没有默认无参构造函数，将
会创建失败--> <bean id="accountService"
class="com.dgut.service.impl.AccountServiceImpl"/>
```

第二种方式：**spring**管理静态工厂-使用静态工厂的方法创建对象

```java
/**
 * 模拟一个静态工厂，创建业务层实现类
 */
public class StaticFactory {
    public static IAccountService createAccountService() {
        return new AccountServiceImpl();
    }
}
```

```xml
<!--
此种方式是：使用StaticFactory类中的静态方法createAccountService创建对象，并存入spring容器
   id属性：指定bean的id，用于从容器中获取
  class属性：指定静态工厂的全限定类名
 factory-method属性：指定生产对象的静态方法 -->
<bean id="accountService2" class="com.dgut.factory.StaticFactory" factory-method="createAccountService"/>
```

第三种方式：**spring**管理实例工厂-使用实例工厂的方法创建对象

```java
/**
 * 模拟一个实例工厂，创建业务层实现类
 * 此工厂创建对象，必须现有工厂实例对象，再调用方法
 */
public class InstanceFactory {
    public IAccountService createAccountService() {
        return new AccountServiceImpl();
    }
}
```

```xml
<!-- 此种方式是：
先把工厂的创建交给spring来管理。
然后在使用工厂的bean来调用里面的方法
  factory-bean属性：用于指定实例工厂bean的id。
  factory-method属性：用于指定实例工厂中创建对象的方法。 -->
<bean id="instanceFactory" class="com.dgut.factory.InstanceFactory" />
<bean id="accountService3"  factory-bean="instanceFactory" factory-method="createAccountService"/>
```

3.5 spring的依赖注入

依赖注入的概念 :**Dependency Injection**。它是spring框架核心ioc的具体实现。

我们的程序在编写时，通过控制反转，把对象的创建交给了spring，但是代码中不可能出现没有依赖的情况。ioc解耦只是降低他们的依赖关系，但不会消除。例如：我们的业务层仍会调用持久层的方法。

那这种业务层和持久层的依赖关系，在使用spring之后，就让spring来维护了。

简单的说，就是坐等框架把持久层对象传入业务层，而不用我们自己去获取。

3.5.1 构造函数注入

顾名思义，就是使用类中的构造函数，给成员变量赋值。注意，赋值的操作不是我们自己做的，而是通过配置的方式，让spring框架来为我们注入。具体代码如下：

```
public class AccountServiceImpl implements IAccountService {

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

    public AccountServiceImpl() {
    }

    public AccountServiceImpl(String name, Integer age, Date birthday) {
        this.name = name;
        this.age = age;
        this.birthday = birthday;
    }
```

```
 <bean id="now" class="java.util.Date"/>
<!-- 使用构造函数的方式，给service中的属性传值
    要求： 类中需要提供一个对应参数列表的构造函数。
涉及的标签： constructor-arg
属性： index:指定参数在构造函数参数列表的索引位置
      type:指定参数在构造函数中的数据类型
      name:指定参数在构造函数中的名称
      value:它能赋的值是基本数据类型和String类型
      ref:它能赋的值是其他bean类型，也就是说，必须得是在配置文件中配置过的bean -->
<bean id="accountService4" class="com.dgut.Service.impl.AccountServiceImpl" >
        <constructor-arg name="age" value="12"/>
        <constructor-arg name="name" value="xieman"/>
        <constructor-arg name="birthday"  ref="now"/>
    </bean>
```

3.5.2 set方法注入

顾名思义，就是在类中提供需要注入成员的set方法。具体代码如下：

```
public class AccountServiceImpl implements IAccountService {

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

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

```
    }

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

    public Integer getAge() {
        return age;
    }
```

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

<!-- 通过配置文件给bean中的属性传值：使用set方法的方式
涉及的标签： property
属性：
  name：找的是类中set方法后面的部分
  ref：给属性赋值是其他bean类型的
 value：给属性赋值是基本数据类型和string类型的
 实际开发过程中用得比较多
-->
    <bean id="accountService4"
class="com.dgut.Service.impl.AccountServiceImpl" >
        <property name="age" value="12"/>
        <property name="name" value="xieman"/>
        <property name="birthday" ref="now"/>
    </bean>
```

3.5.2 集合属性注入

顾名思义，就是给类中的集合成员传值，它用的也是set方法注入的方式，只不过变量的数据类型都是集合。我们这里介绍注入数组，List,Set,Map,Properties。具体代码如下：

```
public class AccountServiceImpl implements IAccountService {

    private String[] myStrs;
    private List<String> myList;
    private Set<String> mySet;
    private Map<String, String> myMap;
    private Properties myProps;

    public String[] getMyStrs() {
        return myStrs;
    }

    public void setMyStrs(String[] myStrs) {
        this.myStrs = myStrs;
    }
```

```
    public List<String> getMyList() {
        return myList;
    }

    public void setMyList(List<String> myList) {
        this.myList = myList;
    }

    public Set<String> getMySet() {
        return mySet;
    }

    public void setMySet(Set<String> mySet) {
        this.mySet = mySet;
    }

    public Map<String, String> getMyMap() {
        return myMap;
    }

    public void setMyMap(Map<String, String> myMap) {
        this.myMap = myMap;
    }

    public Properties getMyProps() {
        return myProps;
    }

    public void setMyProps(Properties myProps) {
        this.myProps = myProps;
    }
```

```
<bean id="accountService4" class="com.dgut.Service.impl.AccountServiceImpl">
        <property name="myStrs">
            <set>
                <value>AAA</value>
                <value>BBB</value>
                <value>CCC</value>
            </set>
        </property>
        <property name="myList">
            <list >
                <value>ccc</value>
                <value>ddd</value>
            </list>
        </property>
        <property name="myMap">
            <map>
```

```
                <entry key="aKey" value="a"></entry>
                <entry key="bKey" value="b"></entry>
            </map>
        </property>
        <property name="myProps">
            <props>
                <prop key="propa">A</prop>
                <prop key="propb">b</prop>
            </props>
        </property>
    </bean>
```

04. 案例：使用spring的IoC的实现账户的CRUD

实现账户的CRUD操作

- pom文件

```
<dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.17</version>
        </dependency>

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

        <dependency>
            <groupId>com.mchange</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.5.4</version>
        </dependency>
    </dependencies>
```

- 创建数据库以及编写实体类

```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);
```

```java
public class Account {
    private Integer id;
    private String name;
    private Float 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;
    }

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

- 编写数据库工具类（演示用）

```
public class DBUtils {

   //这里演示通过C3P0获取连接对象
    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public Connection getConnection() throws SQLException {
        return dataSource.getConnection();
    }
}
```

- 编写持久层代码

```
/**
 * 账户的持久层接口
 */
public interface IAccountDao {

        /**
         * 查询所有
         * @return
         */
        List<Account> findAll();
    }
//实现类
public class AccountDaoImpl implements IAccountDao {

    private DBUtils dbUtils;

    public void setDbUtils(DBUtils dbUtils) {
        this.dbUtils = dbUtils;
    }

    public List<Account> findAll()  {
        Connection conn = null;
        PreparedStatement statement = null;
        ResultSet rs = null;
        List<Account> accounts = new ArrayList<Account>();
        try {
            conn = dbUtils.getConnection();
            statement = conn.prepareStatement("select * from account");
            rs = statement.executeQuery();
            while (rs.next()) {
```

```
                int id = rs.getInt("id");
                String name = rs.getString("name");
                float money = rs.getFloat("money");
                System.out.println(id + " " + name + " " + money);
                accounts.add(new Account(id, name, money));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }finally {
            try {
                rs.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
            try {
                statement.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return accounts;
    }
}
```

- 编写业务层代码

```
public interface IAccountService {

    /**
     * 查询所有
     * @return
     */
    List<Account> findAll();
}

public class AccountService implements IAccountService {

    IAccountDao accountDao;

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

```
    }

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

- 在类目录下创建beans.xml并编写配置文件

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

    <bean id="accountDao" class="com.dgut.dao.impl.AccountDaoImpl">
        <property name="dbUtils" ref="dbUtils"/>
    </bean>
    <bean id="accountService"
class="com.dgut.service.impl.AccountService">
        <property name="accountDao" ref="accountDao"/>
    </bean>
    <bean id="dbUtils" class="com.dgut.utils.DBUtils">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--采用c3p0池-->
    <bean id="dataSource"
class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="com.mysql.cj.jdbc.Driver"/>
        <property name="jdbcUrl" value="jdbc:mysql:///spring"/>
        <property name="user" value="root"/>
        <property name="password" value="xieman123"/>
    </bean>
</beans>
```

- 测试

```
public class AccountTest {

    @Test
    public void testAccountFindAll(){
        ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext("beans.xml");
        IAccountService accountService = (IAccountService)
applicationContext.getBean("accountService");
        accountService.findAll();
    }
}
```

05.基于注解的IOC配置

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

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

我们在讲解注解配置时，采用上一章节的案例，把spring的xml配置内容改为使用注解逐步实现。

5.1@Component注解

相当于：**<bean id="" class="">**

作用： 把资源让spring来管理。相当于在xml中配置一个bean。

属性： value：指定bean的id。如果不指定value属性，默认bean的id是当前类的类名。首字母小写。

开启注解后需要告诉spring容器我是采用注解方式，请帮我扫包

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd"> <!--
<!-- 告知spring创建容器时要扫描的包 -->
  <context:component-scan base-package="com.dgut"></context:component-scan>
```

5.2 @Controller @Service @Repository

他们三个注解都是针对一个的衍生注解，他们的作用及属性都是一模一样的。

他们只不过是提供了更加明确的语义化。

@Controller：一般用于表现层的注解。

@Service：一般用于业务层的注解。

@Repository：一般用于持久层的注解。

5.3 @Autowired、@Qualifier 、@Resource

相当于：

5.3.1 @ Autowired

作用：

自动按照类型注入。当使用注解注入属性时，set方法可以省略。它只能注入其他bean类型。当有多个类型匹配时，使用要注入的**对象变量**名称作为bean的id，在spring容器查找，找到了也可以注入成功。找不到就报错。

5.3.2 @Qualifier

作用：

在自动按照类型注入的基础之上，再按照Bean的id注入。它在给字段注入时不能独立使用，必须和@Autowire一起使用；但是给方法参数注入时，可以独立使用。

属性：

value：指定bean的id。

5.3.3 @Resource

作用：

直接按照Bean的id注入。它也只能注入其他bean类型。

属性：

name：指定bean的id。

5.3.4 @Value

作用：

注入基本数据类型和String类型数据的

属性：

value：用于指定值

5.4 @Scope

作用：

指定bean的作用范围。

属性：

value：指定范围的值。

取值：singleton prototype request session globalsession

5.5 生命周期注解

相当于：**<bean id="" class="" init-method="" destroy-method="" />**

5.5.1 @PostConstruct

作用：

用于指定初始化方法。

5.5.2 @PreDestroy

作用：

用于指定销毁方法。

5.6 关于Spring注解和XML的选择问题

注解的优势：

配置简单，维护方便（我们找到类，就相当于找到了对应的配置）。 只能在用户自己开发的类上注解。

XML的优势：

修改时，不用改源码。不涉及重新编译和部署。

5.7 基于注解改造上一个案例

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

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

我们发现，之所以我们现在离不开xml配置文件，是因为我们有一句很关键的配置：

```
<!-- 告知spring框架在，读取配置文件，创建容器时，扫描注解，依据注解创建对象，并存入容器中 -
->
 <context:component-scan base-package="com.dgut"/>
```

另外，数据源和dbUtils的配置也需要靠注解来实现。

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

<!--    <bean id="accountDao" class="com.dgut.dao.impl.AccountDaoImpl">-->
<!--        <property name="dbUtils" ref="dbUtils"/>-->
```

```
<!--    </bean>-->
<!--    <bean id="accountService"
class="com.dgut.service.impl.AccountService">-->
<!--        <property name="accountDao" ref="accountDao"/>-->
<!--    </bean>-->

    <context:component-scan base-package="com.dgut"/>
    <bean id="dbUtils" class="com.dgut.utils.DBUtils">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--采用c3p0池-->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="com.mysql.cj.jdbc.Driver"/>
        <property name="jdbcUrl" value="jdbc:mysql:///spring"/>
        <property name="user" value="root"/>
        <property name="password" value="xieman123"/>
    </bean>
</beans>
```

5.8 新注解

5.8.1 @Configuration

作用：

用于指定当前类是一个spring配置类，当创建容器时会从该类上加载注解。获取容器时需要使用AnnotationApplicationContext(有@Configuration注解的类.class)。

属性：

value:用于指定配置类的字节码

```
@Configuration
public class SpringConfiguration {
}
//注意：  我们已经把配置文件用类来代替了，但是如何配置创建容器时要扫描的包呢？  请看下一个注解。
```

5.8.2 @ComponentScan

作用：

用于指定spring在初始化容器时要扫描的包。作用和在spring的xml配置文件中的：

<context:component-scan base-package="com.dgut"/>是一样的。

属性：

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

```
@Configuration
@ComponentScan("com.dgut")
public class SpringConfiguration {
}
//注意： 我们已经配置好了要扫描的包，但是数据源和dbUtils对象如何从配置文件中移除呢？ 请看下一个注解。
```

5.8.3 @Bean

作用：

该注解只能写在方法上，表明使用此方法创建一个对象，并且放入spring容器。

属性：

name：给当前@Bean注解方法创建的对象指定一个名称(即bean的id）。 如果没写，ID为方法名，如下面的ID为configDataSource

```
    @Bean
    public DataSource configDataSource() throws PropertyVetoException {
        ComboPooledDataSource ds = new ComboPooledDataSource();
        ds.setJdbcUrl(jdbcUrl);
        ds.setUser(user);
        ds.setPassword(password);
        ds.setDriverClass(driverClass);
        return ds;
    }
```

我们已经把数据源和DBUtils从配置文件中移除了，此时可以删除bean.xml了。

但是由于没有了配置文件，创建数据源的配置又都写死在类中了。如何把它们配置出来呢?

请看下一个注解。

5.8.4 @PropertySource

作用：

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

属性：

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

```
@Configuration
@ComponentScan("com.dgut")
@PropertySource("classpath:jdbc.properties")
public class SpringMyConfig {

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

```
    private String jdbcUrl;
    @Value("com.mysql.cj.jdbc.Driver")
    private String driverClass;
    @Value("root")
    private String user;
    @Value("${jdbc.password}")
    private String password;


    @Bean("dataSource")
    public DataSource configDataSource() throws PropertyVetoException {
        ComboPooledDataSource ds = new ComboPooledDataSource();
        ds.setJdbcUrl(jdbcUrl);
        ds.setUser(user);
        ds.setPassword(password);
        ds.setDriverClass(driverClass);
        return ds;
    }

}
```

```
jdbc.properties
jdbc.url = jdbc:mysql:///spring
jdbc.user = root
jdbc.password = xieman123
```

我们已经把要配置的都配置好了，但是新的问题产生了，由于没有配置文件了，如何获取容器呢?

可以通过注解获取容器

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

0.6 Spring整合Junit

产生的问题

```
在测试类中，每个测试方法都有以下两行代码：
ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");
IAccountService as = ac.getBean("accountService",IAccountService.class);
//这两行代码的作用是获取容器，如果不写的话，直接会提示空指针异常。所以又不能轻易删掉。
```

通过上面的测试类，我们可以看出，每个测试方法都重新获取了一次spring的核心容器，造成了不必要的重复代码，增加了我们开发的工作量。这种情况，在开发中应该避免发生。 如何做?

```
我们的理想
 IAccountService accountService;
    @Test
    public void testAccountFindAll(){
//        ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext("beans.xml");
//        IAccountService accountService = (IAccountService)
applicationContext.getBean("accountService");
        accountService.findAll();
    }
```

- 第一步：pom文件引入包

```
 <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.1.6.RELEASE</version>
            <scope>test</scope>
        </dependency>
```

- 第二步：使用**@RunWith**注解替换原有运行器

```
@RunWith(SpringJUnit4ClassRunner.class)
public class UserTest {
}
```

- 第三步：使用**@ContextConfiguration**指定**spring**配置文件的位置

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {SpringMyConfig.class})
//@ContextConfiguration(locations= {"classpath:bean.xml"})
public class UserTest {
}
```

@ContextConfiguration注解：

locations属性：用于指定配置文件的位置。如果是类路径下，需要用**classpath:**表明

classes属性：用于指定注解的类。当不使用xml配置时，需要用此属性指定注解类的位置。

- 第四步：使用**@Autowired**给测试类中的变量注入数据

```
 @Autowired
IAccountService accountService;
```

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {SpringMyConfig.class})
public class UserTest {
    @Autowired
    IAccountService accountService;
    @Test
    public void test(){
        accountService.findAll();
    }
}
```

07 AOP概念

7.1 什么是AOP

1) 在软件业，**AOP为Aspect Oriented Programming**的缩写，意为：**面向切面编程**，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。

2) AOP是OOP（面向对象编程）的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。

3) 利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

4) AOP采取横向抽取机制，取代了传统纵向继承体系重复性代码

5) 经典应用：事务管理、性能监视、安全检查、缓存、日志等

6) Spring AOP使用纯Java实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类织入增强代码

7) AspectJ是一个基于Java语言的AOP框架，Spring2.0开始，Spring AOP引入对Aspect的支持，AspectJ扩展了Java语言，提供了一个专门的编译器，在编译时提供横向代码的织入

7.2 AOP的作用及优势

作用： 在程序运行期间，不修改源码对已有方法进行增强。

优势：

减少重复代码

提高开发效率

维护方便

7.3 AOP原理

7.3.1 案例引入-日志输出

假设我们现在已开发了一套用户管理系统，现在产品经理提出需求：在用户管理前后添加日志记录，输出执行前后的时间

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

/**
 * 用户管理接口
 */
public interface IUserService {

    /**
     * 添加用户
     */
    public void addUser();
    /**
     * 删除用户
     */
    public void deleteUser();
    /**
     * 编辑用户
     */
    public void editUser();
}

public class UserServiceImpl implements IUserService {

    public void addUser() {
        System.out.println("添加了一个用户");
    }

    public void deleteUser() {
        System.out.println("删除了一用户");

    }

    public void editUser() {
        System.out.println("编辑了用户");
    }
}

public class UserTest {

    @Test
    public void test(){
        IUserService service = new UserServiceImpl();
```

```
        service.addUser();
        service.deleteUser();
        service.editUser();
    }

}
```

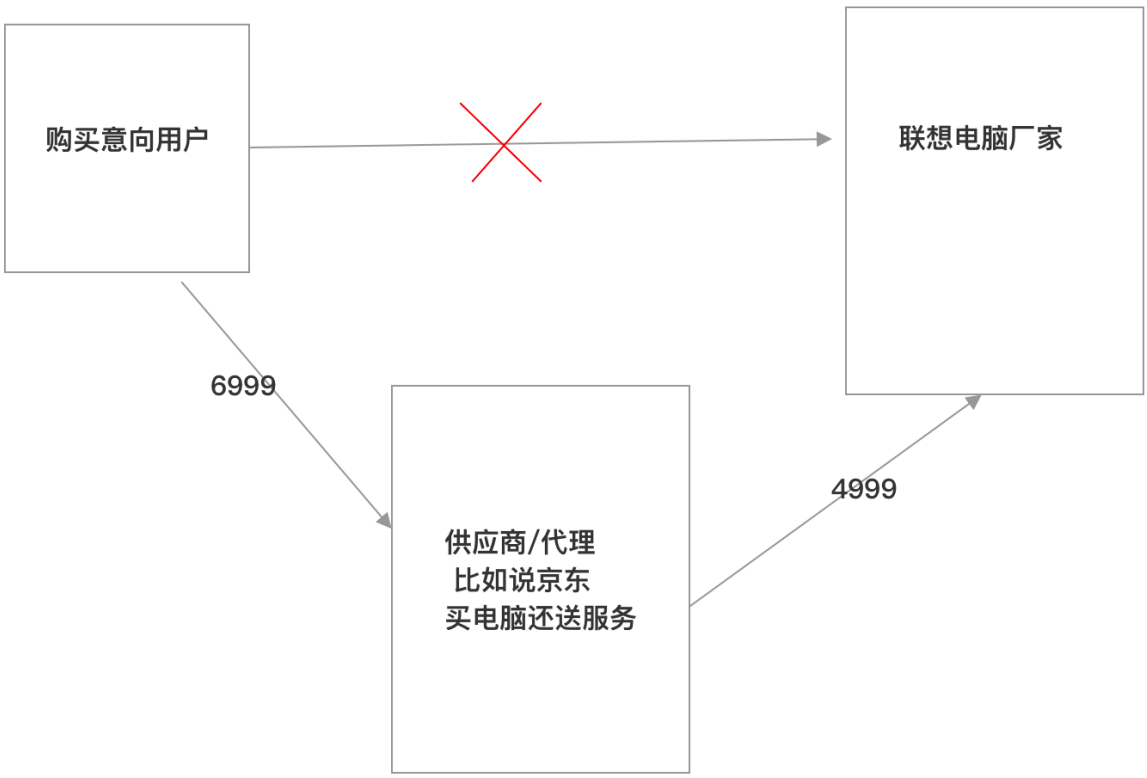
了解这个需求后，同学们按以前的思路可能会在每个方法前后

```
System.out.println("执行前时间"+new Date().getTime());
```

虽然可以实现功能，但我们的代码就是垃圾一堆，大量重复代码，而且毫无健壮性可言。

7.3.2 动态代理技术的两种方式

1. 什么是代理?



2. 基于接口的动态代理

   提供者：JDK官方的Proxy类。

   要求：被代理类最少实现一个接口。

```
 @Test
    public void test() {
        final IUserService service = new UserServiceImpl();

        IUserService serviceProxy = (IUserService)
Proxy.newProxyInstance(UserTest.class.getClassLoader(),
service.getClass().getInterfaces(), new InvocationHandler() {
```

```
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                /**
                 * 执行被代理对象的任何方法，都会经过该方法。
                 * 此方法有拦截的功能。
                 * 参数：
                 * proxy：代理对象的引用。不一定每次都用得到
                 * method：当前执行的方法对象
                 * args：执行方法所需的参数
                 * 返回值：当前执行方法的返回值
                 */
                System.out.println("before");
                Object a = method.invoke(service, args);
                System.out.println("after");
                return a;
            }
        });

        serviceProxy.addUser();
        serviceProxy.deleteUser();
        serviceProxy.editUser();
    }
```

3. 基于子类的动态代理

   提供者：第三方的CGLib

   要求：被代理类不能用final修饰的类（最终类）。

```
 <dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>3.2.12</version>
</dependency>

@Test
    public void test2() {
        final IUserService service = new UserServiceImpl();
        UserServiceImpl userService = (UserServiceImpl)
Enhancer.create(UserServiceImpl.class, new MethodInterceptor() {
            /**
             *  执行被代理对象的任何方法，都会经过该方法。
             *  在此方法内部就可以对被代理对象的任何方法进行增强。
             *  参数：
             *  前三个和基于接口的动态代理是一样的。
             *  MethodProxy：当前执行方法的代理对象。
             *  返回值：当前执行方法的返回值
             */
```

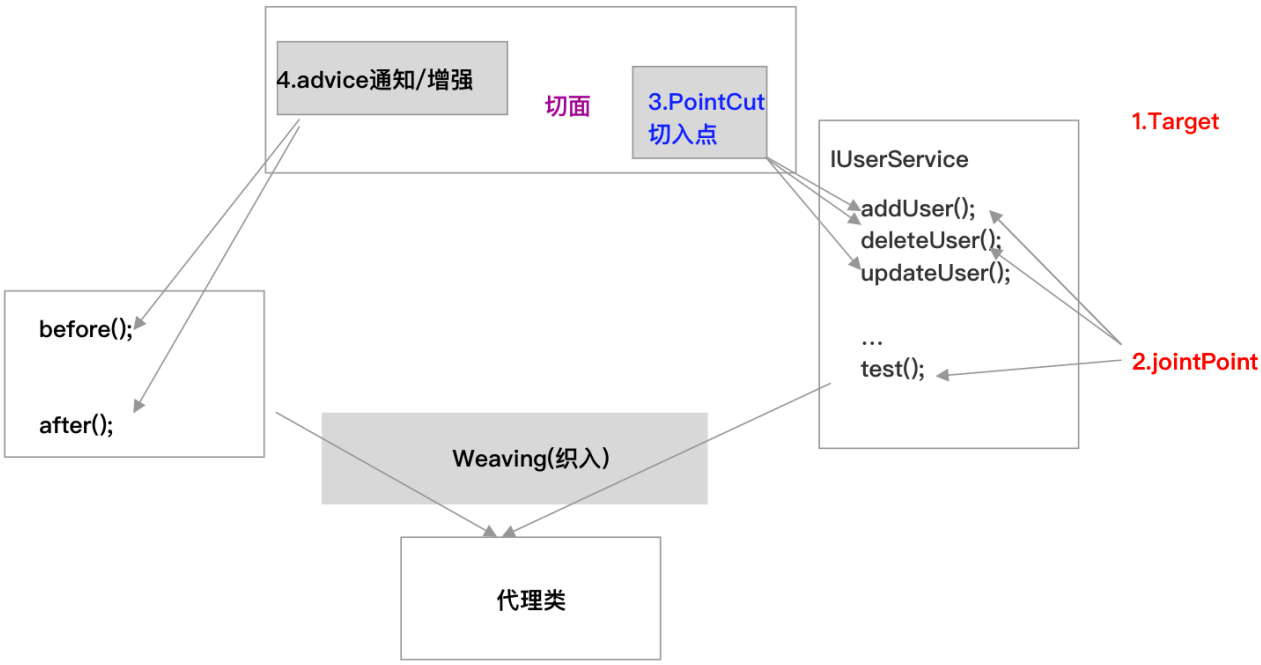
```
            public Object intercept(Object proxy, Method method, Object[]
args, MethodProxy methodProxy) throws Throwable {
                System.out.println("before");
                Object result =  method.invoke(service,args);
                System.out.println("after");
                return result;
            }
        });
        userService.addUser();
    }
```

08 Spring中的AOP

8.1 AOP相关术语

1. target：目标类，需要被代理的类。例如：UserService
2. Joinpoint(连接点):所谓连接点是指那些可能被拦截到的方法。例如：所有的方法
3. PointCut 切入点：已经被增强的连接点。例如：addUser()
4. advice 通知/增强，增强代码。例如：after、before
5. Weaving(织入):是指把增强advice应用到目标对象target来创建新的代理对象proxy的过程.
6. proxy 代理类
7. Aspect(切面): 是切入点pointcut和通知advice的结合

一个切入点和一个通知，组成成一个特殊的面。



8.2 基于XML的aop配置

1. 导包

```
 <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.6.RELEASE</version>
        </dependency>
        <!-- 切入点表达式表达式用-->
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.9.4</version>
        </dependency>
```

2.新建通知类

```
public class MyLog {

    public void log_before(){
        System.out.println("我是log——before");
    }
    public void log_after(){
        System.out.println("我是log——after");
    }
}
```

3. 配置spring

```
<?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="userService" class="com.dgut.service.impl.UserServiceImpl"/>

    <bean id="log" class="com.dgut.log.MyLog"/>

<!--    aop:config: 作用：用于声明开始aop的配置-->
    <aop:config>
        <!--
        aop:aspect: 作用：  用于配置切面。
         属性：  id：给切面提供一个唯一标识。
         ref：引用配置好的通知类bean的id。
        -->
        <!--使用aop:pointcut配置切入点表达式
              aop:pointcut:
```

```
                作用：  用于配置切入点表达式。就是指定对哪些类的哪些方法进行增强。
                属性：  expression：用于定义切入点表达式。
                        id：用于给切入点表达式提供一个唯一标识-->
        <aop:pointcut id="pc1" expression="execution(*
com.dgut.service.impl.*.*(..))"/>
        <aop:aspect id="aspect" ref="log">
            <!--aop:before 作用:
             用于配置前置通知。指定增强的方法在切入点方法之前执行
             属性：  method:用于指定通知类中的增强方法名称
              ponitcut-ref：用于指定切入点的表达式的引用
              poinitcut：用于指定切入点表达式  执行时间点：  切入点方法执行之前执行-->
            <aop:before method="log_before" pointcut-ref="pc1"/>
            <aop:after method="log_after" pointcut-ref="pc1"/>
        </aop:aspect>
    </aop:config>
</beans>
```

4. 编写测试类

```
 @Test
    public void test3(){
        ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext("beans.xml");
        IUserService service = (IUserService)
applicationContext.getBean("userService");
        service.addUser();
    }
```

说明：**切入点表达式**

execution(表达式)

表达式语法：execution([修饰符] 返回值类型 包名.类名.方法名(参数))

写法说明：

```
1.全匹配方式：
public void
com.dgut.service.impl.UserServiceImpl.addUser(com.dgut.domain.User)
访问修饰符可以省略

2.返回值可以使用*号，表示任意返回值
* com.dgut.service.impl.UserServiceImpl.addUser(com.dgut.domain.User)

3.包名可以使用*号，表示任意包，但是有几级包，需要写几个*
* *.*.*.*.UserServiceImpl.addUser(com.dgut.domain.User)
```

```
4.使用..来表示当前包，及其子包
* com..UserServiceImpl.addUser(com.dgut.domain.User)

5.类名可以使用*号，表示任意类
* com..*.addUser(com.dgut.domain.User)

6.方法名可以使用*号，表示任意方法
 * com..*.*(com.dgut.domain.User)

7.参数列表可以使用*，表示参数可以是任意数据类型，但是必须有参数
 * com..*.*(*)

8.参数列表可以使用..表示有无参数均可，有参数可以是任意类型
* com..*.*(..)

9.全通配方式：  * *..*.*(..)

 注：  通常情况下，我们都是对业务层的方法进行增强，所以切入点表达式都是切到业务层实现类。
 execution(* com.dgut.service.impl.*.*(..))
```

8.3 基于注解的aop配置

1.开启注解

```
<?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
http://www.springframework.org/schema/context/spring-context.xsd">
    <context:component-scan base-package="com.dgut"/>
<!--    开启注解AOP-->
    <aop:aspectj-autoproxy/>
</beans>
```

2.使用@Aspect定义一个切面，@Before等定义通知

```java
@Component
@Aspect
public class MyLog {
    @Before("execution(* com.dgut.service.impl.*.*(..))")
    public void log_before(){
        System.out.println("我是log——before");
    }

    @AfterReturning("execution(* com.dgut.service.impl.*.*(..))")
    public void log_after(){
        System.out.println("我是log——after");
    }
}
```