

第1天: Spring

一、学习目标

1. 掌握依赖问题的解决办法
2. 了解spring框架的发展历史和优点
3. 能够描述spring框架
4. 能够理解spring的IOC的容器
5. 能够编写spring的IOC的入门案例
6. 能够说出spring的bean标签的配置
7. 能够理解 Bean的实例化方法
8. 能够理解Bean的属性注入方法
9. 能够理解复杂类型的属性注入

二、依赖问题

2.1 模块化开发

- 模块开发也叫组件式开发, 各模块之间、各子系统之间, 需要保持 **相对的独立性**。

2.1.1 传统开发

- 门户系统是包含企业 **所有业务** 的统一系统, 如下:

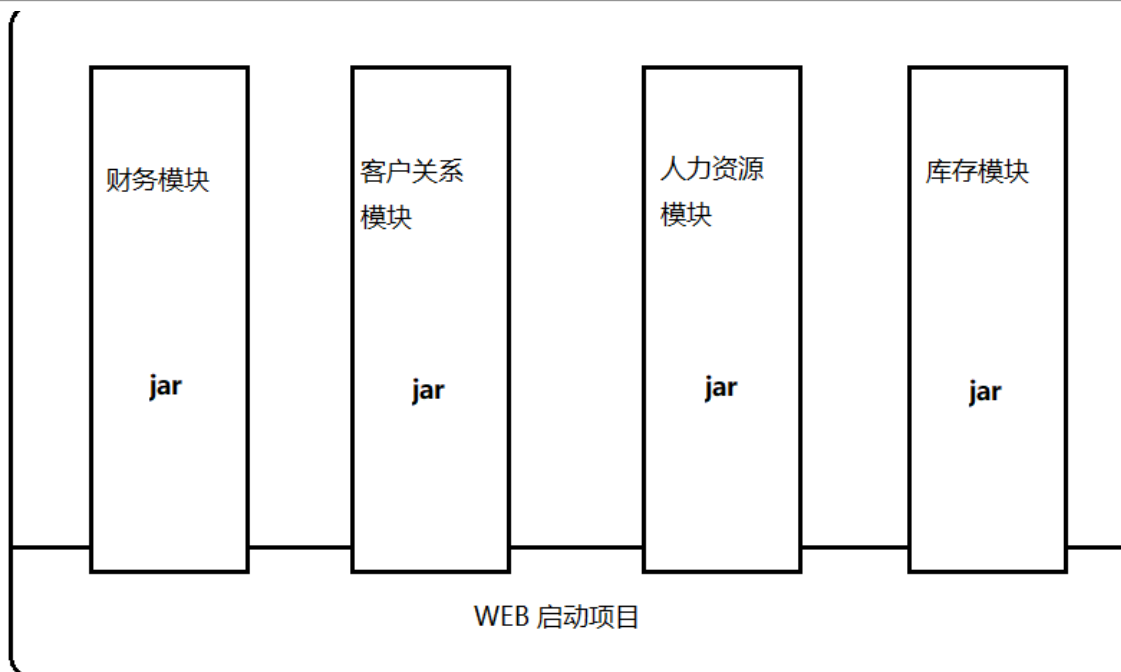
门户系统

WEB 启 动 项 目	com.itheima.caiwu.*	财务相关功能
	com.itheima.kehu.*	客户信息管理功能
	com.itheima.yuangong.*	员工管理功能
	com.itheima.kucun.*	库存管理功能
	com.itheima.x.*	其他功能

ps: 传统开发方式是把这些业务集中在1个web项目中, 缺点是代码量大, 业务插拔不方便。

2.1.2 模块开发

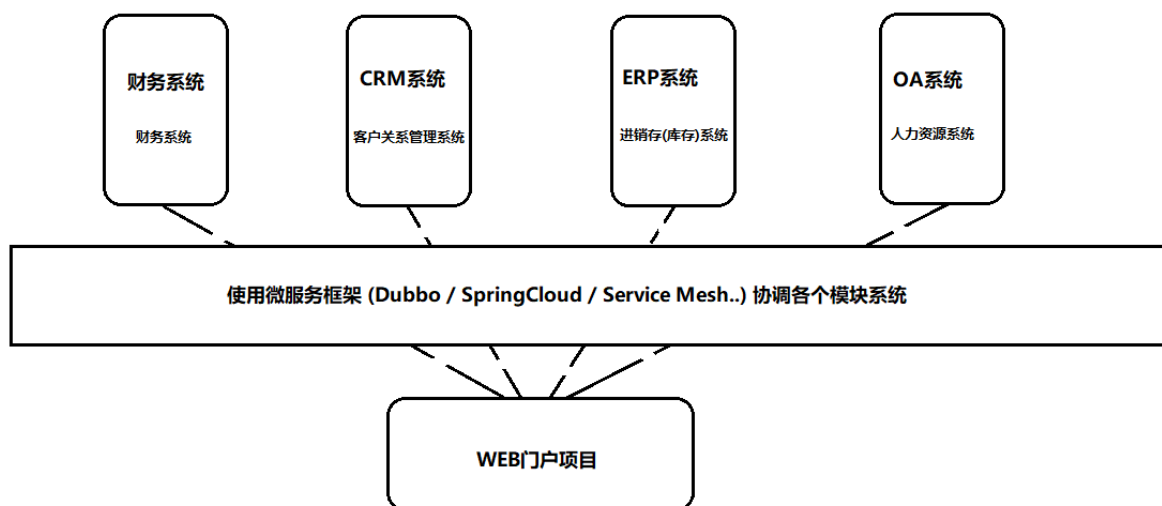
- 将各个业务拆分成多个 **相对独立的模块**, 如下:



ps: 模块化开发业务清晰, 维护方便, 业务插拔较简单, 缺点是各模块间 **仍然存在依赖**。

2.1.3 微服务【扩展】

- 将各个业务拆分成多个 **独立的系统**, 如下:



ps: 本节为扩展内容, 下节将举例讲解模块开发中存在的依赖问题。

2.2 模块依赖问题

- 面向对象的编程原则: 高内聚, **低耦合**。
 - 三层架构[<https://baike.baidu.com/item/三层架构/11031448>] (3-tier architecture) 通常意义上的三层架构就是将整个业务应用划分为: 界面层 (User Interface layer)、业务逻辑层 (Business Logic Layer)、数据访问层 (Data access layer)。

- 微软推荐的分层式结构一般分为三层，从下至上分别为：数据访问层、业务逻辑层（又或称为领域层）、表示层。

2.2.1 举例

- 模拟创建3层架构项目
- 当Dao实现类丢失: 无法编译。

创建项目

- 项目名称: **spring-day01-dep** end

分层开发

- com.itheima.spring.day01.dep.**dao**

```
1 package com.itheima.spring.day01.dep.dao.impl;
2
3 import com.itheima.spring.day01.dep.dao.UserDao;
4
5 /**
6  * 持久层实现类.
7  *
8  * @author : Jason.lee
9  * @version : 1.0
10 * @date : 2019/5/10 9:58
11 * @description : UserDaoImpl
12 */
13 public class UserDaoImpl implements UserDao {
14     @Override
15     public void save() {
16         System.out.println("保存成功");
17     }
18 }
```

- com.itheima.spring.day01.dep.**service**

```
1 package com.itheima.spring.day01.dep.service.impl;
2
3 import com.itheima.spring.day01.dep.dao.UserDao;
4 import com.itheima.spring.day01.dep.dao.impl.UserDaoImpl;
5 import com.itheima.spring.day01.dep.service.UserService;
6
7 /**
8  * 业务层实现类.
9  *
10 * @author : Jason.lee
11 * @version : 1.0
12 * @date : 2019/5/10 10:01
13 * @description : UserServiceImpl
14 */
15 public class UserServiceImpl implements UserService {
16     // 这里应该注入 UserDaoImpl
17     private UserDao userDao;
```



```
18
19     @Override
20     public void save() {
21         userDao.save();
22     }
23 }
```

- com.itheima.spring.day01.dep.controller

```
1 package com.itheima.spring.day01.dep.controller;
2
3 import com.itheima.spring.day01.dep.service.UserService;
4 import com.itheima.spring.day01.dep.service.impl.UserServiceImpl;
5 import org.junit.Test;
6
7 /**
8  * 控制层.
9  *
10  * @author : Jason.lee
11  * @version : 1.0
12  * @date : 2019/5/10 10:06
13  * @description : UserController
14  */
15 public class UserController {
16
17     private UserService userService = new UserServiceImpl();
18
19     @Test
20     public void test (){
21         userService.save();
22     }
23 }
```

持久层丢失

- Service无法编译, 这属于高耦合的依赖问题【可解决】
- 项目无法运行, 这属于代码开发的原则问题【不解决】

2.3 解决依赖问题

- 使用工具类解耦

创建BeanFactory

- com.itheima.spring.day01.dep.factory

```
1 package com.itheima.spring.day01.dep.factory;
2
3 import com.itheima.spring.day01.dep.dao.UserDao;
4 import com.itheima.spring.day01.dep.dao.impl.UserDaoImpl;
5
6 /**
7  * Bean工厂.
```

```
10  * @version : 1.0
11  * @date : 2019/5/10 10:22
12  * @description : BeanFactory
13  */
14  public class BeanFactory {
15
16      /**
17       * 版本1.
18       * 使用工厂方法创建对象解决Service强依赖Dao的问题.
19       * 当UserDaoImpl再丢失时，不影响Service(编译)
20       * 1. 但是问题将转移到工厂方法
21       * 2. 并且只能解耦1个Bean
22       *
23       * @return Bean
24       */
25      public static UserDao getBean(){
26          return new UserDaoImpl();
27      }
28  }
```

使用工厂类

```
1 private UserDao userDao = BeanFactory.getBean();
```

配置解耦

```
1 # 用户持久层实现类
2 UserDao=com.itheima.spring.day01.dep.dao.impl.UserDaoImpl
3 # 用户服务层实现类
4 UserService=com.itheima.spring.day01.dep.service.impl.UserServiceImpl
```

升级优化

```
1 // properties工具类
2 private static Properties properties = new Properties();
3
4 static {
5     InputStream in = BeanFactory.class.getClassLoader()
6         .getResourceAsStream("bean.properties");
7     try {
8         properties.load(in);
9     } catch (IOException e) {
10         System.out.println("加载bean.properties文件出错");
11         e.printStackTrace();
12     }
13 }
14
15 /**
16  *版本2.
17  * 使用反射在运行过程中加载相关字节码
18  * 1. 实现类再丢失只影响运行
19  * 2. 切换实现类只需改变配置 (java代码不变)
20  *
21  * @return Bean
```



```
24  */
25  public static <T> T getBean(String name){
26      try {
27          String clazz = properties.getProperty(name);
28          Class<?> t = Class.forName(clazz);
29          return (T) t.newInstance();
30      } catch (Exception e) {
31          System.out.println(name+": 实现类不存在");
32          e.printStackTrace();
33      }
34      return null;
35  }
```

使用升级

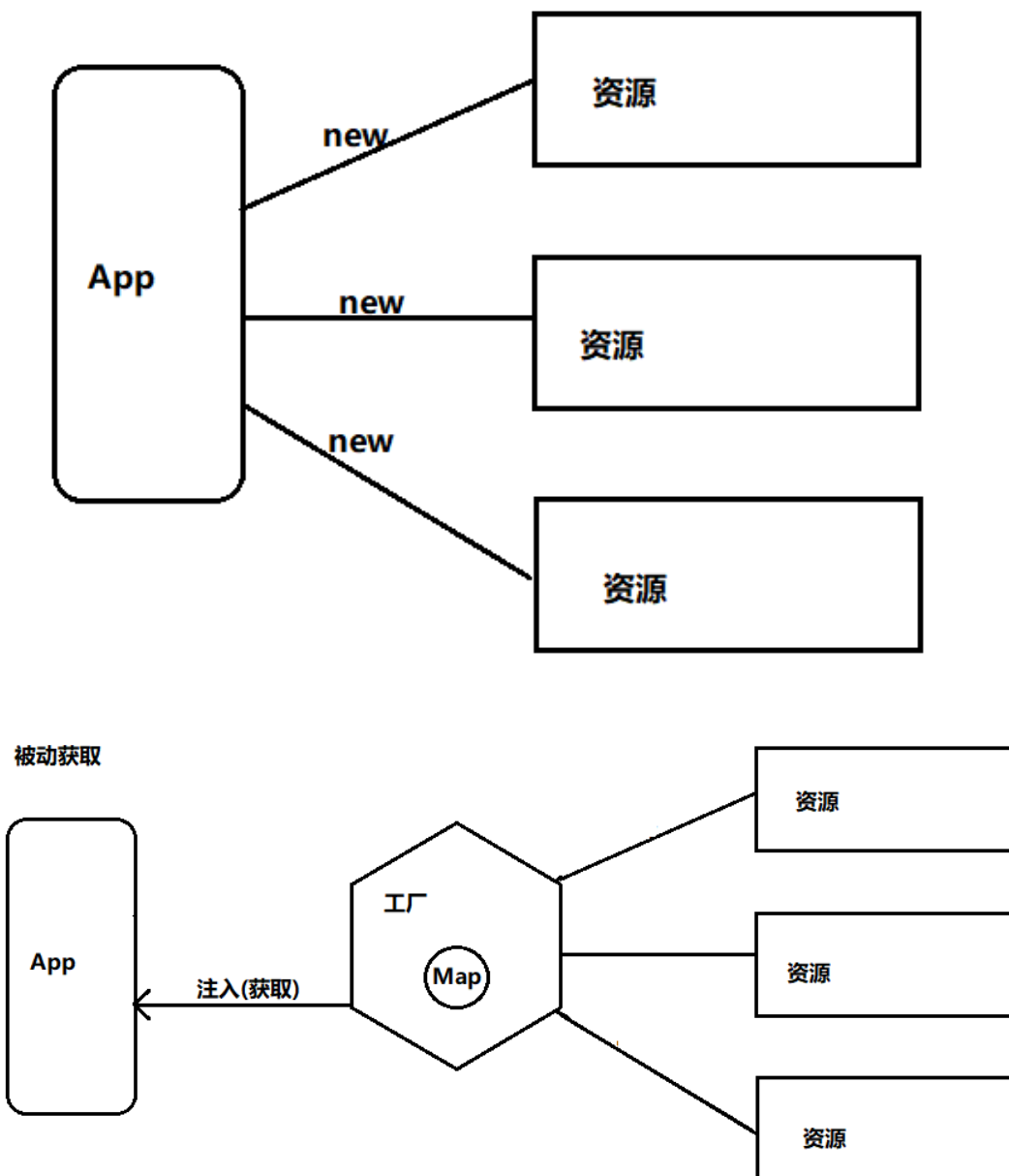
```
1 | private UserDao userDao = BeanFactory.getBean("UserDao");
```

- 这种工厂类的解耦方式被广泛应用, 在 **设计模式** 中称之为 **工厂模式**

2.4 理解工厂模式

- 在实际项目开发中, 我们可以通过配置文件把controller、service、dao对象配置起来
- 当启动服务器加载应用的时候, 读取配置文件, 创建配置文件中的对象并且保存起来。
- 在接下来实际使用的时候, 直接拿过来使用即可。

- 此时我们需要考虑两个问题:
 - 将对象存放在什么地方?
 - 由于项目中对象是大量的, 所以考虑用集合 Map/List 存储
 - 而在使用时需要根据名称查找, 所以通常使用 **Map** 存储
 - 什么是工厂?
 - 工厂是负责创建对象, 并且把对象放到容器中。
 - 并且在使用的时候, 帮助我们从容获取指定的对象。
 - 此时我们获取对象的方式发生了改变 —— 由**主动创建**变成了**被动获取**。



- 结论
 - 这种将创建对象的权利，由在程序代码中主动new对象的方式，转变为由工厂类创建提供，我们使用的地方被动接收的方式。称为**控制反转**。
 - 控制反转也叫IOC（Inversion Of Control），即我们接下来要学习的spring框架中的一个重要知识点。
 - 在这里我们首先明确一个事情：spring的IOC解决的问题，即是工厂模式解耦解决的问题。

三、框架概诉

3.1 Spring 介绍

- 它是以IOC（Inversion Of Control）控制反转和AOP（Aspect Oriented Programming）面向切面编程为核心，提供了表现层springmvc和持久层spring JDBC以及业务层的事务管理等企业级应用解决方案。
- 还能实现将开源世界中众多优秀的第三方框架和类库整合，成为越来越受欢迎的Java EE企业级应用框架。

3.2 Spring 发展历程

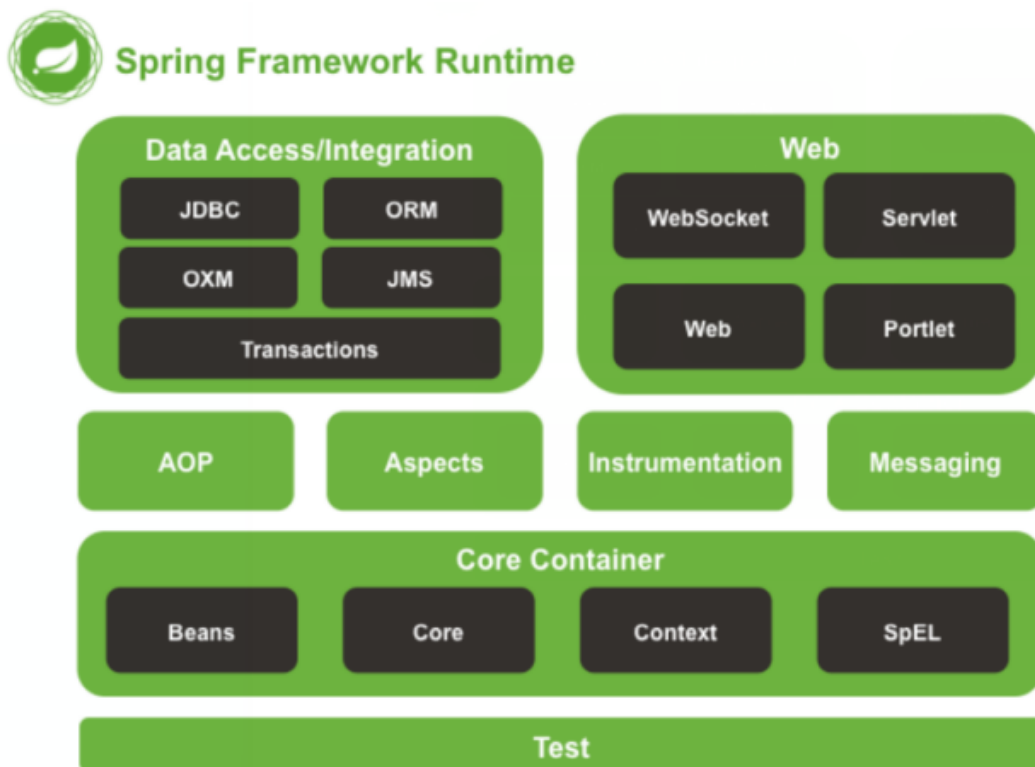
- 1997年IBM提出了EJB的思想
- 1998年，SUN制定开发标准规范EJB1.0
- 1999年，EJB1.1发布
- 2001年，EJB2.0发布
- 2003年，EJB2.1发布
- 2006年，EJB3.0发布
- Rod Johnson（spring之父）
Expert One-to-One J2EE Design and Development(2002)
阐述了J2EE 使用EJB 开发设计的优点及解决方案
Expert One-to-One J2EE Development without EJB(2004)
阐述了J2EE 开发不使用EJB 的解决方式（Spring 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

3.3 Spring 优点

- IOC解耦，简化开发
 - IOC容器，可以将对象间的依赖关系交由spring管理，避免硬编码造成的程序间过度耦合。
 - 用户也不必再为了编写工厂类，属性文件解析等底层实现编写代码，可以更加专注于业务系统需求的实现。
- AOP面向切面编程支持
 - AOP功能，方便实现面向切面编程，很多使用传统OOP编程不容易实现的业务功能，可以通过AOP轻松实现。
 - 比如事务管理，日志功能。
- 声明式事务支持
 - 通过声明式方式灵活实现事务管理，提高开发效率和质量
 - 将开发者从单调烦闷的事务管理代码中解脱出来。
- 方便程序测试
 - 可以使用非容器依赖的方式进行程序测试工作，让测试工作更加轻松，更加方便。
- 集成第三方优秀框架

- 学习java源码的经典案例
 - spring的源码设计精妙、结构清晰、匠心独具，处处体现了大师对java设计模式的灵活应用以及java技术的高深造诣。它的源代码无疑是java技术的最佳实践案例。

3.4 Spring 体系结构



四、IOC入门

4.1 IOC介绍

- IOC (Inversion Of Control) 控制反转。
- 是面向对象编程的一个重要法则，用于削减计算机程序间的耦合问题。
- 控制反转中分为两种类型，一种是DI (Dependency Injection) 依赖注入；
- 另外一种DL (Dependency Lookup) 依赖查找。实际应用中依赖注入使用更多。

4.2 XML入门案例

- 官网: <https://spring.io/>
- 下载: <http://repo.springsource.org/libs-release-local/org/springframework/spring/>
- 本地: 课前资料 / 框架包
 - doc: API文档
 - libs: jar包和源码包
 - schema: 约束文件

4.2.1 创建3层结构项目

```
1 public class CustomDao {
2
3     public void save(){
4         System.out.println("保存成功");
5     }
6
7 }
```

- 业务层

```
1 public class CustomService {
2
3     public void save(){
4         CustomDao customDao = new CustomDao();
5         customDao.save();
6     }
7 }
```

- 表现层

```
1 public class CustomController {
2
3     public static void main(String[] args) {
4         // 创建业务层对象
5         CustomService customService = new CustomService();
6         customService.save();
7     }
8 }
```

- 演示

Connected to the target VM, address: '127.0.0.1:60200', transport: 'socket'

保存成功

Disconnected from the target VM, address: '127.0.0.1:60200', transport: 'socket'

Process finished with exit code 0

4.2.2 配置spring框架

- pom.xml

```
1 <dependencies>
2     <!-- Spring IOC依赖 -->
3     <dependency>
4         <groupId>org.springframework</groupId>
5         <artifactId>spring-context</artifactId>
6         <version>5.1.7.RELEASE</version>
7     </dependency>
8 </dependencies>
```

- beans.xml

```
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
5
6      <!-- 将Service和dao放置到IOC容器中 -->
7      <bean id="customService" class="CustomService" />
8      <bean id="customDao" class="CustomDao" />
9
10 </beans>
```

- 表现层

```
1 public class CustomController {
2
3     public static void main(String[] args) {
4         // 启动spring框架(IOC容器)
5         ClassPathXmlApplicationContext factory = new
ClassPathXmlApplicationContext("beans.xml");
6         // 从ioc容器中获取业务层对象
7         CustomService customService = factory.getBean("customService",
CustomService.class);
8         customService.save();
9     }
10 }
```

- 演示

Connected to the target VM, address: '127.0.0.1:60200', transport: 'socket'

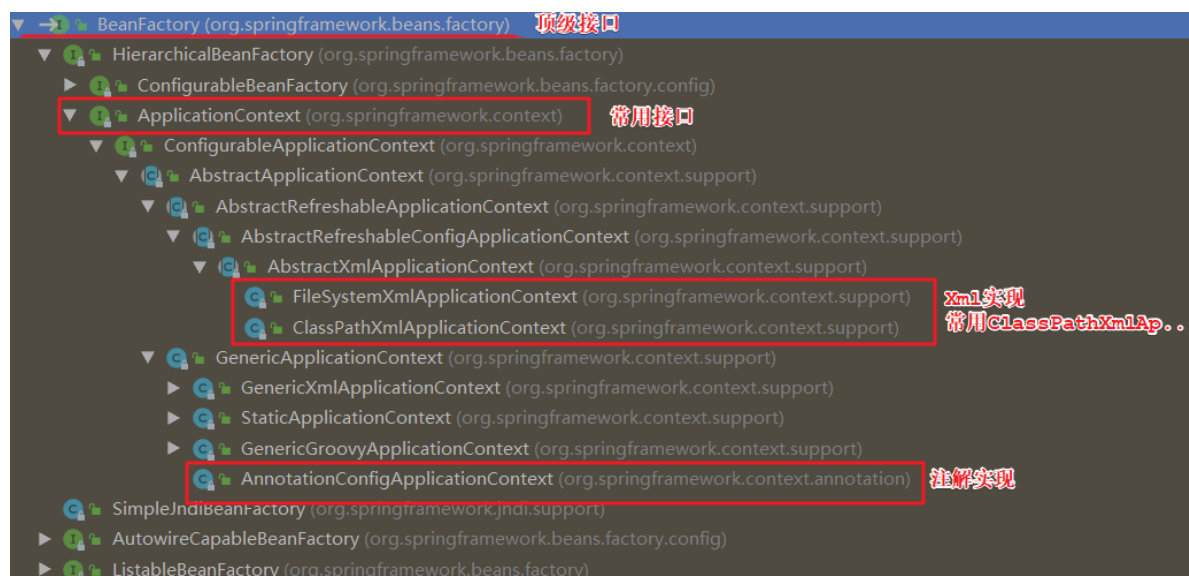
保存成功

Disconnected from the target VM, address: '127.0.0.1:60200', transport: 'socket'

Process finished with exit code 0

4.3 XML配置细节

4.3.1 工厂结构



- BeanFactory: IOC容器 **顶层** 接口
- ApplicationContext: IOC容器 **常用** 接口

实现方式

- ClassPathXmlApplicationContext: **基于项目Xml配置实现的容器**
- AnnotationConfigApplicationContext: **基于注解配置实现的容器**

4.3.2 接口区别

创建对象的时间不同

- BeanFactory是Spring容器的顶层接口, 采用 **延迟创建** 对象的思想
- ApplicationContext是BeanFactory的子接口, 采用 **即时创建** 对象的思想

BeanFactory 案例

- 改造Controller

```
1 public class CustomController {
2
3     public static void main(String[] args) {
4         // 启动spring框架
5         Resource resource = new ClassPathResource("beans.xml");
6         // 使用BeanFactory接口
7         BeanFactory factory = new XmlBeanFactory(resource);
8
9         System.out.println("容器先创建..");
10
11        // 从IOC容器中获取业务层对象
12        CustomService customService = factory.getBean("customService",
13        CustomService.class);
14        customService.save();
15    }
16 }
```

- 改造Service

```
1 public class CustomService {
2
3     public CustomService () {
4         // 为了展示创建顺序，打印如下
5         System.out.println("对象后创建..");
6     }
7
8
9     public void save() {
10        CustomDao customDao = new CustomDao();
11        customDao.save();
12    }
13 }
```

- 测试

对象 后 创建..

保存成功

Disconnected from the target VM, address: '127.0.0.1:61442', transport: 'socket'

Process finished with exit code 0

- ApplicationContext测试

Connected to the target VM, address: '127.0.0.1:61490', transport: 'socket'

对象 后 创建..

容器 先 创建..

保存成功

Disconnected from the target VM, address: '127.0.0.1:61490', transport: 'socket'

Process finished with exit code 0

4.3.3 Bean标签

bean标签的作用

- 配置javaBean对象，让spring容器创建管理
- 默认调用类中无参数的构造方法创建对象

bean标签的属性

属性	说明
id	bean的唯一标识名称
class	类的全限定名称
scope	设置bean的作用范围。 singleton : 单例, 默认值 prototype : 多例 request: web项目中 , 将对象存入request域中 session: web项目中 , 将对象存入session域中 globalsession: web项目中 , 应用在集群环境, 没有集群环境, 相当于session
init-method	指定类中初始化方法的名称, 在构造方法执行完毕后立即执行
destroy-method	指定类中销毁方法名称, 在销毁spring容器前执行

scope生命周期

单例对象: scope="singleton"	一个应用中只有一个对象实例	出生: 加载配置文件, 容器创建, 对象出生 活着: 只要容器存在, 对象就一直活着 死亡: 容器销毁, 对象死亡
多例对象: scope="prototype"	在一次使用过程中	出生: 第一次获取对象, 对象出生 活着: 在一次使用过程中, 对象活着 死亡: 当对象不再使用, 也没有被其它对象引用, 交由垃圾回收器回收

scope案例演示

- singleton

```

1 <!-- 测试singleton范围 (默认) -->
2 <bean id="customDao1" scope="singleton"
  class="com.itheima.bean.dao.CustomDao"/>
    
```

- 代码示例

```

1 @Test
2 public void testScope () {
3     // 1. 启动容器
4     BeanFactory factory = new
    ClassPathXmlApplicationContext("bean.xml");
5     CustomDao customDao1 = factory.getBean("customDao1",
    CustomDao.class);
6     CustomDao customDao2 = factory.getBean("customDao1",
    CustomDao.class);
7     // 2. 对象对比
8     System.out.println(customDao1==customDao2);
9     System.out.println(customDao1.hashCode());
10    System.out.println(customDao2.hashCode());
11 }
    
```

- 测试结果

```

Connected to the target VM, address: '127.0.0.1:50428', transport: 'socket'
true
1050065615
1050065615
Disconnected from the target VM, address: '127.0.0.1:50428', transport: 'socket'
Process finished with exit code 0
    
```

- prototype

```

1 <!-- 测试prototype的范围 -->
2 <bean id="customDao2" scope="prototype"
  class="com.itheima.bean.dao.CustomDao"/>
    
```

- 代码示例



```

3 // 1. 启动容器
4 BeanFactory factory = new
ClassPathXmlApplicationContext("bean.xml");
5 CustomDao customDao1 = factory.getBean("customDao2",
CustomDao.class);
6 CustomDao customDao2 = factory.getBean("customDao2",
CustomDao.class);
7 // 2. 对象对比
8 System.out.println(customDao1==customDao2);
9 System.out.println(customDao1.hashCode());
10 System.out.println(customDao2.hashCode());
11 }

```

• 测试结果

```

Connected to the target VM, address: '127.0.0.1:50504', transport: 'socket'
false
1730704097
848363848
Disconnected from the target VM, address: '127.0.0.1:50504', transport: 'socket'
Process finished with exit code 0

```

bean的其他属性

- init-method和destroy-method
- 改造Dao

```

1 public class CustomDao {
2     public CustomDao () {
3         System.out.println("创建对象");
4     }
5
6
7     public void save() {
8         System.out.println("保存成功");
9     }
10
11
12     public void init() {
13         System.out.println("初始化..");
14     }
15
16
17     public void destroy() {
18         System.out.println("销毁..");
19     }
20 }

```

- 改造bean.xml



```
class="com.itheima.bean.dao.CustomDao"/>
```

- 测试代码

```
1 @Test
2 public void testI_D () {
3     // 1. 启动容器
4     ClassPathXmlApplicationContext factory = new
5     ClassPathXmlApplicationContext("bean.xml");
6     CustomDao customDao1 = factory.getBean("customDao3",
7     CustomDao.class);
8     // 2. 关闭容器
9     factory.close();
10 }
```

- 测试结果

Connected to the target VM, address: '127.0.0.1:51546', transport: 'socket'

创建对象

创建对象

初始化..

销毁..

Disconnected from the target VM, address: '127.0.0.1:51546', transport: 'socket'

4.3.4 实例化

无参构造方法

- bean.xml

```
1 <!-- 无参构造方法实例化 -->
2 <bean id="customDao1" class="com.itheima.bean.dao.CustomDao"/>
```

- 代码与测试

- 与scope案例一致

静态工厂方法

- bean.xml

```
1 <!-- 测试静态工厂创建对象 -->
2 <bean id="customDao4" class="com.itheima.bean.StaticFactory" factory-
3 method="createCustomDao"/>
```

- 静态工厂



```

3      /**
4       * 使用静态方法创建对象
5       * @return
6       */
7      public static CustomDao createCustomDao() {
8          System.out.println("静态工厂方法准备创建对象。");
9          return new CustomDao();
10     }
11 }

```

- 测试代码

```

1  @Test
2  public void testStaticFactory () {
3      BeanFactory factory = new
4      ClassPathXmlApplicationContext("bean.xml");
5      CustomDao customDao1 = factory.getBean("customDao4",
6      CustomDao.class);
7      customDao1.save();
8  }

```

- 测试结果

Connected to the target VM, address: '127.0.0.1:52091', transport: 'socket'

静态工厂方法准备创建对象。

创建对象

保存成功

Disconnected from the target VM, address: '127.0.0.1:52091', transport: 'socket'

实例工厂方法

- bean.xml

```

1  <!-- 测试实例工厂创建对象-->
2  <bean id="instanceFactory" class="com.itheima.bean.InstanceFactory"/>
3  <bean id="customDao5" factory-bean="instanceFactory" factory-
4  method="createCustomDao"/>

```

- 实例工厂

```

1  public class InstanceFactory {
2
3      /**
4       * 使用实例工厂创建对象
5       * @return
6       */
7      public CustomDao createCustomDao(){
8          System.out.println("实例工厂方法准备创建对象。");
9          return new CustomDao();
10     }
11 }

```

- 测试代码

```
3     BeanFactory factory = new  
    ClassPathXmlApplicationContext("bean.xml");  
4     CustomDao customDao1 = factory.getBean("customDao5",  
    CustomDao.class);  
5     customDao1.save();  
6 }
```

- 测试结果

Connected to the target VM, address: '127.0.0.1:52206', transport: 'socket'

实例工厂方法准备创建对象。

创建对象

保存成功

Disconnected from the target VM, address: '127.0.0.1:52206', transport: 'socket'

五、依赖注入

5.1 依赖注入介绍

- 依赖注入（Dependency Injection），可以将Bean中依赖的其他Bean注入进去，从而达到维护Bean的依赖关系功能。
- 简单理解：依赖注入就是给成员变量赋值。

5.2 注入类型

- Bean成员变量的赋值方式
- 创建工程: maven-day01-di

5.2.1 构造方法注入

- 通过构造方法赋值
- CustomDao.java

```
1 public class CustomDao {  
2  
3     private int id;  
4     private String name;  
5     private Integer age;  
6     private Date birthday;  
7  
8     public CustomDao(int id, String name, Integer age, Date birthday) {  
9         this.id = id;  
10        this.name = name;  
11        this.age = age;  
12        this.birthday = birthday;  
13    }  
14  
15    public void save(){  
16        System.out.println(id+","+name+","+age+","+birthday);  
17    }
```

```
1  <!-- 讲解构造方法注入，说明：
2      constructor-arg: 指定通过构造方法，给成员变量赋值
3      属性：
4          index: 指定成员变量在构造方法参数列表中的索引
5          name: 指定成员变量在构造方法参数列表中的名称（index和name二者使用一个即可）
6          type: 指定成员变量的类型（一般不需要指定，默认即可）
7          value: 给java简单类型成员变量赋值（八种基本类型+字符串）
8          ref: 给其它bean类型成员变量赋值
9  -->
10 <bean id="customDao1" class="com.itheima.di.dao.CustomDao">
11     <constructor-arg index="0" name="id" type="int" value="1"/>
12     <constructor-arg name="name" value="明世隐"/>
13     <constructor-arg name="age" value="18"/>
14     <constructor-arg name="birthday" ref="now"/>
15 </bean>
```

- 测试代码

```
1  @Test
2  public void testConstructor () {
3      ClassPathXmlApplicationContext context = new
4      ClassPathXmlApplicationContext("bean.xml");
5      CustomDao customDao1 = context.getBean("customDao1",
6      CustomDao.class);
7      customDao1.save();
8  }
```

- 测试结果

Connected to the target VM, address: '127.0.0.1:52296', transport: 'socket'

1,明世隐,18,Mon May 20 13:47:04 CST 2019

Disconnected from the target VM, address: '127.0.0.1:52296', transport: 'socket'

5.2.2 set方法注入

- 通过set方法赋值
- CustomDao.java

```
1  public void setId(int id) {
2      this.id = id;
3  }
4
5  public void setName(String name) {
6      this.name = name;
7  }
8
9  public void setAge(Integer age) {
10     this.age = age;
11 }
12
```

```
15 | }
```

- bean.xml

```
1  <!--讲解set方法注入，说明：
2      property: 指定通过构造方法，给成员变量赋值
3      属性：
4          name: 指定成员变量在构造方法参数列表中的名称
5          value: 给java简单类型成员变量赋值（八种基本类型+字符串）
6          ref: 给其它bean类型成员变量赋值
7  -->
8  <bean id="customDao2" class="com.itheima.di.dao.CustomDao">
9      <property name="id" value="2"/>
10     <property name="name" value="盾山"/>
11     <property name="age" value="18"/>
12     <property name="birthday" ref="now"/>
13 </bean>
```

- 测试代码

```
1  @Test
2  public void testSet (){
3      ClassPathXmlApplicationContext context = new
4      ClassPathXmlApplicationContext("bean.xml");
5      CustomDao customDao1 = context.getBean("customDao2",
6      CustomDao.class);
7      customDao1.save();
8  }
```

- 测试结果

Connected to the target VM, address: '127.0.0.1:52406', transport: 'socket'

2,盾山,18,Mon May 20 13:53:56 CST 2019

Disconnected from the target VM, address: '127.0.0.1:52406', transport: 'socket'

5.2.3 P名称空间注入

- 使用名称空间的p标签赋值
- 改造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      <!-- 命名空间: p标签 -->
5      xmlns:p="http://www.springframework.org/schema/p"
6      xsi:schemaLocation="http://www.springframework.org/schema/beans
7      http://www.springframework.org/schema/beans/spring-beans.xsd">
8
9      <bean id="now" class="java.util.Date"/>
10
11     <bean id="customDao3" class="com.itheima.di.dao.CustomDao"
12     <!-- p标签，本质上还是调用set方法赋值 -->
```

15 | </beans>

- 测试代码

```
1  @Test
2  public void testP () {
3      ClassPathXmlApplicationContext context = new
4      ClassPathXmlApplicationContext("bean.xml");
5      CustomDao customDao1 = context.getBean("customDao3",
6      CustomDao.class);
7      customDao1.save();
8  }
```

- 测试结果

Connected to the target VM, address: '127.0.0.1:52483', transport: 'socket'

1,张良,18,Mon May 20 14:01:17 CST 2019

Disconnected from the target VM, address: '127.0.0.1:52483', transport: 'socket'

5.2.4 C名称空间注入

- 使用名称空间的c标签赋值
- 改造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:p="http://www.springframework.org/schema/p"
5      xmlns:c="http://www.springframework.org/schema/c"
6      xsi:schemaLocation="http://www.springframework.org/schema/beans
7      http://www.springframework.org/schema/beans/spring-beans.xsd">
8
9      <bean id="now" class="java.util.Date"/>
10
11     <bean id="customDao4" class="com.itheima.di.dao.CustomDao"
12         <!-- c标签：本质上还是调用构造方法赋值 -->
13         c:id="1" c:name="东皇太一" c:age="18" c:birthday-ref="now"
14     />
15
16 </beans>
```

- 测试代码

```
1  @Test
2  public void testC () {
3      ClassPathXmlApplicationContext context = new
4      ClassPathXmlApplicationContext("bean.xml");
5      CustomDao customDao1 = context.getBean("customDao4",
6      CustomDao.class);
7      customDao1.save();
8  }
```

Connected to the target VM, address: '127.0.0.1:52520', transport: 'socket'

1,东皇太一,18,Mon May 20 14:05:31 CST 2019

Disconnected from the target VM, address: '127.0.0.1:52520', transport: 'socket'

5.3 集合属性注入

- 该部分是【扩展】内容

5.3.1 set方法注入

- CollectionDao.java

```
1 public class CollectionDao {
2     private String[] array;
3     private List<String> list;
4     private Set<String> set;
5     private Map<String,String> map;
6     private Properties prop;
7
8     public void setArray(String[] array) {
9         this.array = array;
10    }
11
12    public void setList(List<String> list) {
13        this.list = list;
14    }
15
16    public void setSet(Set<String> set) {
17        this.set = set;
18    }
19
20    public void setMap(Map<String, String> map) {
21        this.map = map;
22    }
23
24    public void setProp(Properties prop) {
25        this.prop = prop;
26    }
27
28    public void save() {
29        System.out.println(array !=null? Arrays.asList(array): "");
30        System.out.println(list);
31        System.out.println(set);
32        System.out.println(map);
33        System.out.println(prop);
34    }
35
36
37 }
```

- bean.xml



```
3      array/list/set
4      2.Map结构:
5      map/prop
6      3. 数据结构一致，标签可以互换
7      -->
8      <bean id="collectionDao" class="com.itheima.di.dao.CollectionDao">
9          <!--array-->
10         <property name="array">
11             <array>
12                 <value>白起</value>
13                 <value>钟馗</value>
14             </array>
15         </property>
16         <!--list-->
17         <property name="list">
18             <list>
19                 <value>大乔</value>
20                 <value>小乔</value>
21             </list>
22         </property>
23         <!--set-->
24         <property name="set">
25             <set>
26                 <value>诸葛亮</value>
27                 <value>司马懿</value>
28             </set>
29         </property>
30         <!--map-->
31         <property name="map">
32             <map>
33                 <entry key="张良" value="法师"></entry>
34                 <entry key="东皇太一" value="坦克"></entry>
35             </map>
36         </property>
37         <!--prop-->
38         <property name="prop">
39             <props>
40                 <prop key="盾山">辅助</prop>
41                 <prop key="大乔">辅助</prop>
42             </props>
43         </property>
44     </bean>
```

- 测试代码

```
1  @Test
2  public void testCollection () {
3      ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("bean.xml");
4      CollectionDao collectionDao = context.getBean("collectionDao",
CollectionDao.class);
5      collectionDao.save();
6  }
```

[大乔, 小乔]

[诸葛亮, 司马懿]

{张良=法师, 东皇太一=坦克}

{盾山=辅助, 大乔=辅助}

Disconnected from the target VM, address: '127.0.0.1:52728', transport: 'socket'