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框架的发展历程和优点
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
10. 使用注解代替相关XML配置
11. 能够编写Spring组件扫描配置
12. 理解Spring相关注解的含义

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

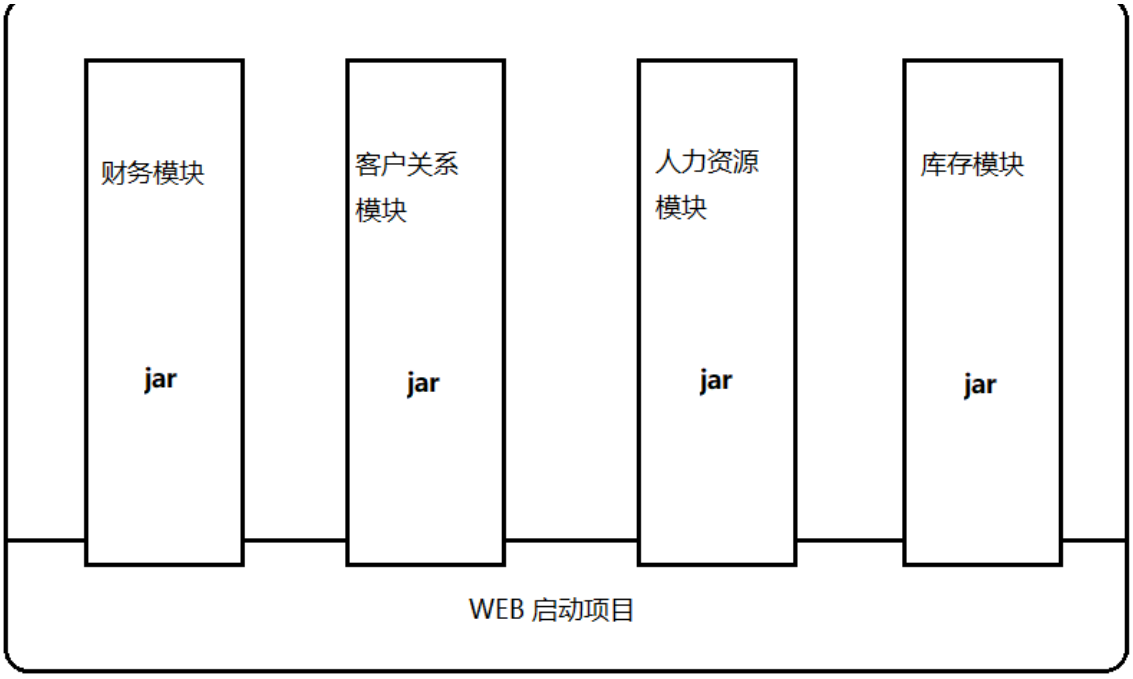
- 门户系统是包含企业 **所有业务** 的统一系统, 如下:

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

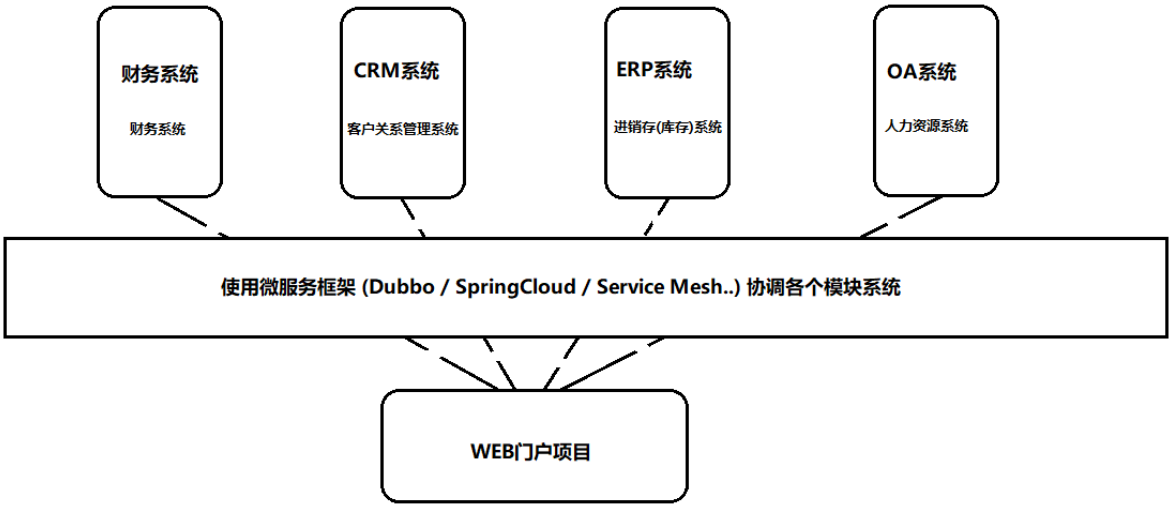
2.1.2 模块开发



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**

```java
package com.itheima.spring.day01.dep.dao.impl;

import com.itheima.spring.day01.dep.dao.UserDao;

/**
 * 持久层实现类.
 *
 * @author : Jason.lee
 * @version : 1.0
 * @date : 2019/5/10 9:58
 * @description : UserDaoImpl
 */
public class UserDaoImpl implements UserDao {
    @Override
    public void save() {
        System.out.println("保存成功");
    }
}
```

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

```java
package com.itheima.spring.day01.dep.service.impl;

import com.itheima.spring.day01.dep.dao.UserDao;
import com.itheima.spring.day01.dep.dao.impl.UserDaoImpl;
import com.itheima.spring.day01.dep.service.UserService;

/**
 * 业务层实现类.
 *
 * @author : Jason.lee
 * @version : 1.0
 * @date : 2019/5/10 10:01
 * @description : UserServiceImpl
 */
```

```
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.
```

```
 * @version : 1.0
 * @date : 2019/5/10 10:22
 * @description : BeanFactory
 */
public class BeanFactory {

    /**
     * 版本1.
     *  使用工厂方法创建对象解决Service强依赖Dao的问题.
     *  当UserDaoImpl再丢失时，不影响Service(编译)
     *  1. 但是问题将转移到工厂方法
     *  2. 并且只能解耦1个Bean
     *
     * @return Bean
     */
    public static UserDao getBean(){
        return new UserDaoImpl();
    }
}
```

使用工厂类

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

配置解耦

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

升级优化

```
// properties工具类
private static Properties properties = new Properties();

static {
    InputStream in = BeanFactory.class.getClassLoader()
            .getResourceAsStream("bean.properties");
    try {
        properties.load(in);
    } catch (IOException e) {
        System.out.println("加载bean.properties文件出错");
        e.printStackTrace();
    }
}

/**
 *版本2.
 *  使用反射在运行过程中加载相关字节码
 *  1. 实现类再丢失只影响运行
 *  2. 切换实现类只需改变配置 (java代码不变)
 *
```

```
 */
public static <T> T getBean(String name){
    try {
        String clazz = properties.getProperty(name);
        Class<?> t = Class.forName(clazz);
        return (T) t.newInstance();
    } catch (Exception e) {
        System.out.println(name+": 实现类不存在");
        e.printStackTrace();
    }
    return null;
}
```

使用升级

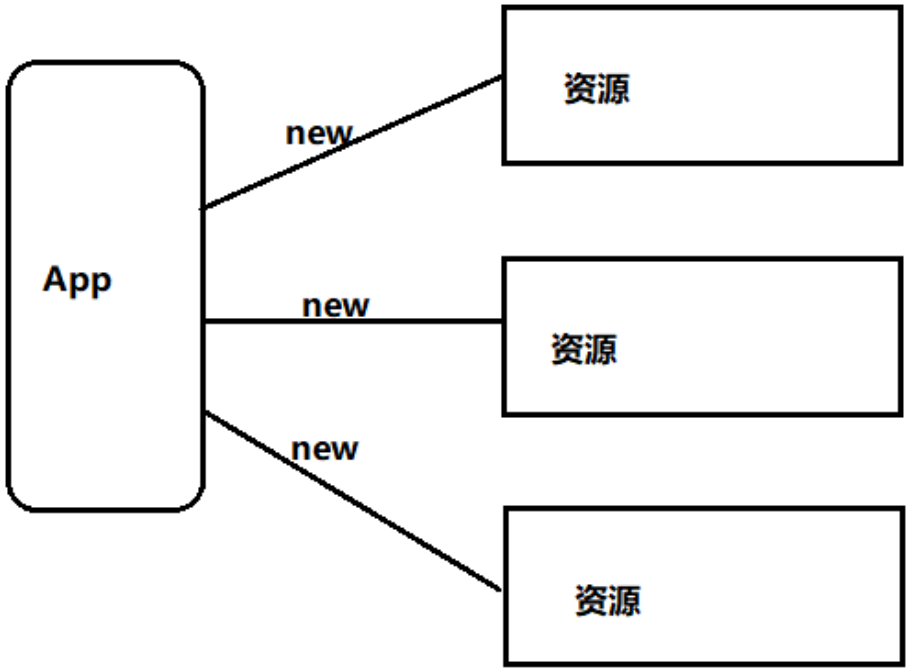
```
private UserDao userDao = BeanFactory.getBean("UserDao");
```

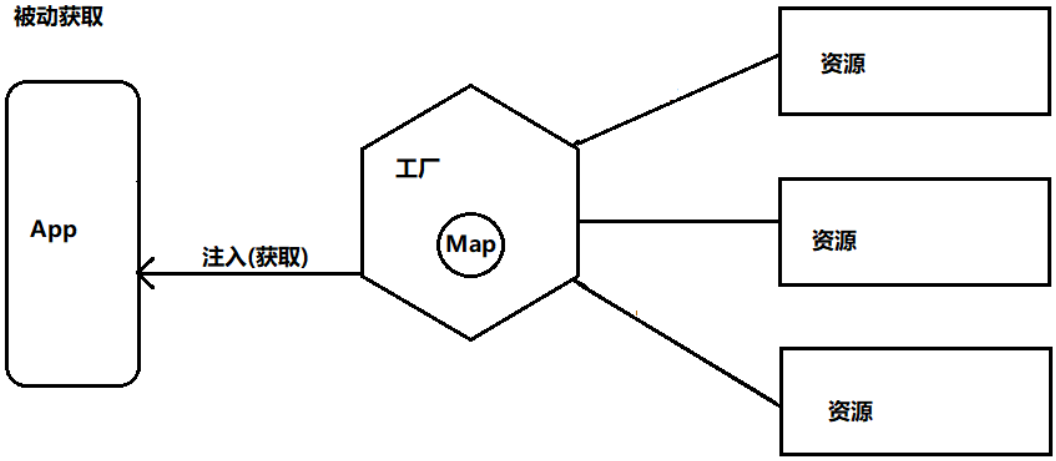
- 这种工厂类的解耦方式被广泛应用, 在 **设计模式** 中称之为 **工厂模式**

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，帮助我们从容器获取指定的对象。
    - 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渡耦合。
  - 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。

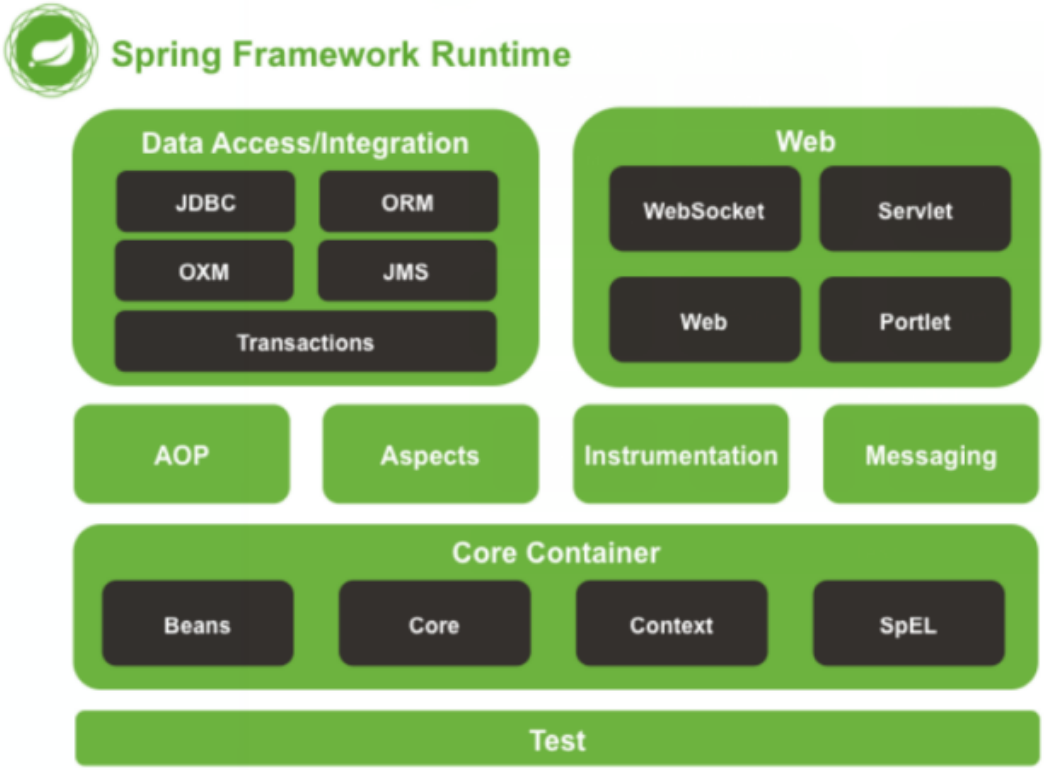
- 声明式事务支持
  - 通过声明式方式灵活实现事务管理，提高开发效率和质量
  - 将开发者从单调烦闷的事务管理代码中解脱出来。

- 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；
- 另外一种是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

```java
public class CustomDao {

    public void save(){
        System.out.println("保存成功");
    }

}
```

- 业务层

```java
public class CustomService {

    public void save(){
        CustomDao customDao = new CustomDao();
        customDao.save();
    }
}
```

- 表现层

```java
public class CustomController {

    public static void main(String[] args) {
        // 创建业务层对象
        CustomService customService = new CustomService();
        customService.save();
    }
}
```

- 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

```xml
<dependencies>
    <!-- Spring IOC依赖 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.7.RELEASE</version>
    </dependency>
</dependencies>
```

- beans.xml

```
        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">

    <!-- 将Service和dao放置到IOC容器中 -->
    <bean id="customService" class="CustomService" />
    <bean id="customDao" class="CustomDao" />

</beans>
```

- 表现层

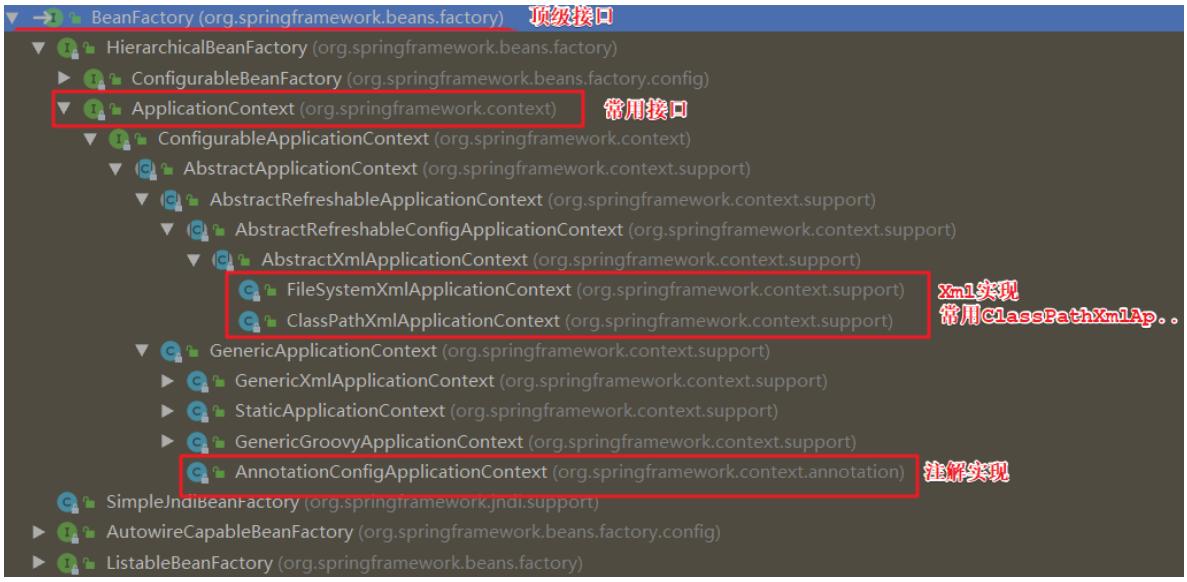
```
public class CustomController {

    public static void main(String[] args) {
        // 启动spring框架(IOC容器)
        ClassPathXmlApplicationContext factory = new
ClassPathXmlApplicationContext("beans.xml");
        // 从IOC容器中获取业务层对象
        CustomService customService = factory.getBean("customService",
CustomService.class);
        customService.save();
    }
}
```

- 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

```
public class CustomController {

    public static void main(String[] args) {
        // 启动spring框架
        Resource resource = new ClassPathResource("beans.xml");
        // 使用BeanFactory接口
        BeanFactory factory = new XmlBeanFactory(resource);

        System.out.println("容器先创建..");

        // 从IOC容器中获取业务层对象
        CustomService customService = factory.getBean("customService",
CustomService.class);
        customService.save();
    }
}
```

- 改造Service

```
public class CustomService {

    public CustomService (){
        // 为了展示创建顺序，打印如下
        System.out.println("对象后创建..");
    }


    public void save(){
        CustomDao customDao = new CustomDao();
        customDao.save();
    }
}
```

- 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

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

- 代码示例

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

- 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

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

- 代码示例

```
    // 1. 启动容器
    BeanFactory factory = new ClassPathXmlApplicationContext("bean.xml");
    CustomDao customDao1 = factory.getBean("customDao2", CustomDao.class);
    CustomDao customDao2 = factory.getBean("customDao2", CustomDao.class);
    // 2. 对象对比
    System.out.println(customDao1==customDao2);
    System.out.println(customDao1.hashCode());
    System.out.println(customDao2.hashCode());
}
```

- 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

```
public class CustomDao {
    public CustomDao (){
        System.out.println("创建对象");
    }


    public void save(){
        System.out.println("保存成功");
    }


    public void init(){
        System.out.println("初始化..");
    }


    public void destroy(){
        System.out.println("销毁..");
    }
}
```

- 改造bean.xml

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

- 测试代码

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

- 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

```
<!-- 无参构造方法实例化 -->
<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

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

- 静态工厂

```
    /**
     * 使用静态方法创建对象
     * @return
     */
    public static CustomDao createCustomDao() {
        System.out.println("静态工厂方法准备创建对象。");
        return new CustomDao();
    }
}
```

- 测试代码

```
@Test
public void testStaticFactory (){
    BeanFactory factory = new
ClassPathXmlApplicationContext("bean.xml");
    CustomDao customDao1 = factory.getBean("customDao4",
CustomDao.class);
    customDao1.save();
}
```

- 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

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

- 实例工厂

```
public class InstanceFactory {

    /**
     * 使用实例工厂创建对象
     * @return
     */
    public CustomDao createCustomDao(){
        System.out.println("实例工厂方法准备创建对象。");
        return new CustomDao();
    }
}
```

- 测试代码

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

- 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

```
public class CustomDao {

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

    public CustomDao(int id, String name, Integer age, Date birthday) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.birthday = birthday;
    }

    public void save(){
        System.out.println(id+","+name+","+age+","+birthday);
    }
```

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

- 测试代码

```
@Test
public void testConstructor (){
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("bean.xml");
    CustomDao customDao1 = context.getBean("customDao1", CustomDao.class);
    customDao1.save();
}
```

- 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

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

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

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

```

```
15 }
```

- bean.xml

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

- 测试代码

```
@Test
public void testSet (){
    ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("bean.xml");
    CustomDao customDao1 = context.getBean("customDao2",
CustomDao.class);
    customDao1.save();
}
```

- 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

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

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

    <bean id="customDao3" class="com.itheima.di.dao.CustomDao"
```

```
15 </beans>
```

- 测试代码

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

- 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
   ClassPathXmlApplicationContext("bean.xml");
4      CustomDao customDao1 = context.getBean("customDao4",
   CustomDao.class);
5      customDao1.save();
6  }
```

> 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

```
public class CollectionDao {
    private String[] array;
    private List<String> list;
    private Set<String> set;
    private Map<String,String> map;
    private Properties prop;

    public void setArray(String[] array) {
        this.array = array;
    }

    public void setList(List<String> list) {
        this.list = list;
    }

    public void setSet(Set<String> set) {
        this.set = set;
    }

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

    public void setProp(Properties prop) {
        this.prop = prop;
    }

    public void save() {
        System.out.println(array !=null? Arrays.asList(array):"");
        System.out.println(list);
        System.out.println(set);
        System.out.println(map);
        System.out.println(prop);
    }


}
```

- bean.xml

```
            array/list/set
    2.Map结构：
            map/prop
    3.数据结构一致，标签可以互换
-->
<bean id="collectionDao" class="com.itheima.di.dao.CollectionDao">
    <!--array-->
    <property name="array">
        <array>
            <value>白起</value>
            <value>钟馗</value>
        </array>
    </property>
    <!--list-->
    <property name="list">
        <list>
            <value>大乔</value>
            <value>小乔</value>
        </list>
    </property>
    <!--set-->
    <property name="set">
        <set>
            <value>诸葛亮</value>
            <value>司马懿</value>
        </set>
    </property>
    <!--map-->
    <property name="map">
        <map>
            <entry key="张良" value="法师"></entry>
            <entry key="东皇太一" value="坦克"></entry>
        </map>
    </property>
    <!--prop-->
    <property name="prop">
        <props>
            <prop key="盾山">辅助</prop>
            <prop key="大乔">辅助</prop>
        </props>
    </property>
</bean>
```

- 测试代码

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

```
[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'
```

六、基于注解的IOC配置

- 前言: 注解只是代替xml配置的一种方式

6.1 环境搭建

- 构建工程: spring-day02-anno
- pom.xml

```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">
    <parent>
        <artifactId>spring2</artifactId>
        <groupId>com.itheima</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>spring-day02-anno</artifactId>

    <dependencies>
        <!-- Spring IOC容器依赖 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.7.RELEASE</version>
        </dependency>
        <!-- 单元测试 依赖 -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

- Account.java

```java
package com.itheima.anno;
```

```java
/**
 * @Component:  用于修饰以下之外的类
 *  @Controller 用于修饰视图层的控制器
 *  @Service    用于修饰业务层的业务类
 *  @Repository 用于修饰持久层的操作类
 *  位置: 类
 *  作用: 创建对象并加入到IOC容器中
 *  替代: bean标签
 *      value: 对象名称, 相当于bean标签属性id, 默认首字母小写的类名
 *
 *  【扩展】: 同时使用只有1个生效, 并且以指定名称的为准
 */
@Component
public class Account {
    private Integer id;
    private Integer uid;
    private Double money;

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

- applicationContext.xml

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

    <!--
        使用注解需要开启注解扫描 (用于发现注解类)
            base-package: 扫描指定包路径以及其包下所有子包
     -->
    <context:component-scan base-package="com.itheima.anno"/>


    <!--
        使用注解后不需要再定义对象
     -->
    <!--<bean id="account" class="com.itheima.anno.Account"/>-->
</beans>
```

- 单元测试

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

/**
 * IOC注解单元测试类.
 *
 * @author : Jason.lee
 * @version : 1.0
 */
public class AnnoTests {

    // 创建IOC容器
    ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

    @Test
    public void testCreate (){
        // 根据名称获取对象
        Account account = (Account) context.getBean("account");
        System.out.println(account);
    }
}

```

6.2 常用注解

3.2.1 创建对象

@Component

- 位置: 类
- 作用: 创建对象并加入到IOC容器中
- 意义: **代替** bean标签配置
- value: 对象名称, 相当于bean标签属性id, 默认首字母小写的类名

@Controller

- @Component在三层架构中衍生的控制器注解
- 用于识别为不同层的对象 (不需要识别时可用@Component代替)

@Service

- @Component在三层架构中衍生的业务类注解
- 用于识别为不同层的对象 (不需要识别时可用@Component代替)

@Repository

- @Component在三层架构中衍生的业务类注解
- 用于识别为不同层的对象 (不需要识别时可用@Component代替)

3.2.2 依赖注入

- User.java

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.Date;

/**
 * @Autowired:
 * @Qualifier: 配合@Autowired使用，只注入指定名称的对象（类型失效）
 * @Value: 注入基本数据类型属性值 或 配置文件中的参数值
 *  位置：属性，方法，参数
 *  作用：根据(类型+名称)注入属性值
 *  意义：代替bean标签中的子标签prototype
 *      required：注入的对象是否必须（true：非空）
 *
 *  【扩展】：多处使用以参数位置为准
 *
 * @Resource: 根据类型+名称 或 单独指定名称注入（相当于@Autowired+@Qualifier）
 *  位置：属性，方法，类
 *  【扩展】：与@Autowired一起使用以@Resource为准
 *
 * 【扩展】：@Autowired @Value @Resource 一起使用
 *  优先级：@Autowired < @Value < @Resource（以优先级最大的为准）
 */
@Component
public class User {
    private Integer id;

    @Autowired(required = false)
    private String username;
    @Value("2019/10/11")
    private Date birthday;
    @Value("1")
    private String sex;
    @Resource(name = "name4")
    private String address;

    @Autowired
    public void setUsername(@Autowired String username) {
        this.username = username;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", birthday=" + birthday +
                ", sex='" + sex + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

- applicationContext.xml

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">

    <!--
        使用注解需要开启注解扫描 (用于发现注解类)
            base-package: 扫描指定包路径以及其包下所有子包
     -->
    <context:component-scan base-package="com.itheima.anno"/>


    <!--
        使用注解后不需要再定义对象
     -->
    <!--<bean id="account" class="com.itheima.anno.Account"/>-->

    <!-- 添加OK字符串对象到IOC容器 -->
    <bean id="username" name="name name2" class="java.lang.String">
        <constructor-arg value="OK"/>
    </bean>
    <bean id="username2" name="name3,name4" class="java.lang.String">
        <constructor-arg value="KO"/>
    </bean>
</beans>
```

- 单元测试

```
@Test
public void testDi (){
    // 根据类型获取对象
    User user = context.getBean(User.class);
    System.out.println(user);
}
```

@Autowired

- 位置: 属性, 方法, 参数
- 作用: 根据对象(类型+名称)注入属性值
- 意义: 代替bean标签中的子标签prototype
- required: 注入的对象是否必须 (true: 非空)
- 【扩展】: 多处使用以参数位置为准

@Qualifier

- 通常**配合@Autowired使用**
- 位置: 与@Autowired一致
- 作用: 只根据对象名称注入
- value: 注入指定名称的对象

- 位置: 属性, 方法, 类
- 作用: 根据类型+名称 或 单独指定名称注入
- 意义: 相当于**@Autowired+@Qualifier (JDK1.8以后不支持)**
- name: 注入指定名称的对象

@Value

- 位置: 属性, 方法, 参数
- 作用: **注入配置文件的参数** 或 基本数据类型数据
- 意义: 配合@PropertySource代替<context:property-placeholder标签
- value: 注入指定内容 (可以使用占位符获取参数值注入)

扩展提醒: 【同时使用】@Autowired < @Value < @Resource (以优先级最大的为准)

6.3 作用范围注解

- Account.java

```
package com.itheima.anno;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Controller;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;

/**
 * @Component:  用于修饰以下之外的类
 *  @Controller 用于修饰视图层的控制器
 *  @Service    用于修饰业务层的业务类
 *  @Repository 用于修饰持久层的操作类
 *  位置: 类
 *  作用: 创建对象并加入到IOC容器中
 *  替代: bean标签
 *      value: 对象名称, 相当于bean标签属性id, 默认首字母小写的类名
 *
 *  【扩展】: 同时使用只有1个生效, 并且以指定名称的为准
 *
 * @Scope: 配合创建对象的注解使用
 *  位置: 类, 方法
 *  作用: 修饰对象的作用范围
 *  替代: bean标签scope属性
 *      singleton: 不配置或不指定value值, 默认singleton
 *      prototype: 多例, 每次获取将创建新的对象
 *
 */
@Component
@Controller
@Service
@Repository
@Scope("prototype")
```

```
    private Integer uid;
    private Double money;

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

- 单元测试

```
@Test
public void testScope (){
    // 根据类型获取对象
    Account account1 = context.getBean(Account.class);
    Account account2 = context.getBean(Account.class);
    System.out.println(account1==account2);
    System.out.println(account1.hashCode());
    System.out.println(account2.hashCode());
}
```

@Scope

- @Scope: 配合创建对象的注解使用

- 位置: 类, 方法
- 作用: 修饰对象的作用范围
- 替代: bean标签scope属性
- singleton: 默认singleton; 单例: 只创建一次对象
- prototype: 多例, 每次获取将创建新的对象

6.4 生命周期注解

- Account.java

```
package com.itheima.anno;

import org.springframework.context.annotation.Lazy;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Controller;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

```

```
 *  @Controller 用于修饰视图层的控制器
 *  @Service    用于修饰业务层的业务类
 *  @Repository 用于修饰持久层的操作类
 *  位置: 类
 *  作用: 创建对象并加入到IOC容器中
 *  意义: 替代bean标签
 *      value: 对象名称, 相当于bean标签属性id, 默认首字母小写的类名
 *
 *  【扩展】: 同时使用只有1个生效, 并且以指定名称的为准
 *
 * @Scope: 用于修饰对象的作用范围
 *  位置: 类, 方法
 *  作用: 配置对象的作用范围
 *  替代: bean标签scope属性
 *      singleton: 不配置或不指定value值, 默认singleton
 *      prototype: 多例, 每次获取将创建新的对象
 *
 * @Lazy:
 *  位置: 类
 *  作用: 单例模式下使用延迟创建对象的策略
 *  意义: 替代bean标签属性 lazy-init
 *      value: true; 默认false
 *
 *  【扩展】: 与Scope("prototype")同时使用, @Lazy不生效
 */
@Component
@Controller
@Service
@Repository
@Scope("prototype")
@Lazy
public class Account {
    private Integer id;
    private Integer uid;
    private Double money;

    public Account() {
        System.out.println("构造方法执行..");
    }

    /**
     * @PostConstruct:
     *  位置: 方法
     *  作用: 构造方法执行后执行
     *  意义: 代替bean标签 init-method 属性
     */
    @PostConstruct
    public void init(){
        System.out.println("初始化方法执行..");
    }


    /**
     * @PostConstruct:
     *  位置: 方法
```

```
    @PreDestroy
    public void destroy(){
        System.out.println("销毁方法执行..");
    }

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

- 单元测试

```
@Test
public void testLc (){
    System.out.println("容器已创建..");
    Account account = context.getBean(Account.class);
    System.out.println(account);
    // 销毁容器
    context.close();
}
```

@PostConstruct

- 位置: 方法
  - 作用: **构造方法**执行**后执行**
  - 意义: 代替bean标签 init-method 属性

@PreDestroy

- 位置: 方法
- 作用: **容器销毁前**执行
- 意义: 代替bean标签 destroy-method 属性

@Lazy

- 位置: 类
- 作用: 单例模式下使用延迟创建对象的策略
- 意义: 替代bean标签属性 lazy-init
- 【扩展】: @Lazy只在单例模式下有效

6.5 注解配置选择

- 注解的优势: 简单, 可读性高
- 配置的优势: 解耦, 非侵入性

Bean定义		@Component
Bean名称	bean标签的id或name属性	@Component的value值
Bean注入	bean标签的子标签<property	@Autowired(1) + @Qualifier (?)
Bean生命周期	bean标签的init- | destroy- method属性	@PostConstruct | @PreDestroy
Bean延迟创建	bean标签的lazy-init | scope 属性	@Lazy | @Scope
适用场景	第三方源码类的对象创建 (必用)	自定义类的简单应用对象创建 (可用)