1、Spring IoC / DI 概念总结

1）Spring IoC 容器：负责**实例化、配置**和**组装 bean（组件）**核心容器。容器通过读取配置元数据来获取有关要实例化、配置和组装组件的指令。

2）IoC：主要是针对对象的创建和调用控制而言的，也就是说，当应用程序需要使用一个对象时，不再是应用程序直接创建该对象，而是**由 IoC 容器来创建和管理，即控制权由应用程序转移到 IoC 容器中，也就是“反转”了控制权**。这种方式基本上是通过依赖查找的方式来实现的，即 **IoC 容器维护着构成应用程序的对象，并负责创建这些对象**。

3）DI：是指在**组件之间传递依赖关系**的过程中，将依赖关系**在容器内部进行处理**，这样就**不必在应用程序代码中硬编码对象之间的依赖关系，实现了对象之间的解耦合**。在 Spring 中，DI 是通过 XML 配置文件或注解的方式实现的。三种形式的依赖注入：构造函数注入、Setter 方法注入和接口注入。

2、Spring IoC/DI 实现步骤：

1）**配置元数据**：即是编写交给 SpringIoC 容器管理组件的信息，配置方式有三种：XML 配置方式、注解方式和 Java 配置类方式；（也就是定义好 bean 的相关信息和相关依赖关系）

2）**实例化 IoC 容器**：提供 ApplicationContext 构造函数的位置路径是资源字符串地址，允许容器从各种外部资源（如本地文件系统、Java CLASSPATH 等）加载配置元数据。

3）**获取 Bean（组件）**：ApplicationContext 是一个高级工厂的接口，能够维护不同 bean 及其依赖项的注册表。通过使用方法 T getBean(String name, Class<T> requiredType)，可以检索 bean 的实例。

```
//创建ioc容器对象，指定配置文件，ioc也开始实例组件对象
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml",
"daos.xml");
//获取ioc容器的组件对象
PetStoreService service = context.getBean "petStore", PetStoreService.class);
//使用组件对象
List<String> userList = service.getUsernameList();
```

【三种配置方式总结】

1）XML 方式配置总结

1>所有内容写到 xml 格式配置文件中；
2>声明 bean 通过\<bean 标签；
3><bean>标签包含基本信息（id,class）和属性信息 <property name, value / ref>；
4>引入外部的 properties 文件可以通过<context:property-placeholder=“”>；
5>IoC 具体容器实现选择 ClassPathXmlApplicationContext 对象。

2）XML+注解方式配置总结

1>注解负责标记 IoC 的类和进行属性装配；
2>xml 文件依然需要，需要通过<context:component-scan 标签指定注解范围；
3>标记 IoC 注解：@Component,@Service,@Controller,@Repository；
4>标记 DI 注解：@Autowired,@Qualifier,@Resource,@Value；
5>IoC 具体容器实现选择 ClassPathXmlApplicationContext 对象。

3）完全注解方式配置总结

1>完全注解方式指的是去掉 xml 文件，使用**配置类+注解**实现；
2>xml 文件替换成使用**@Configuration 注解**标记的类；
3>标记 IoC 注解：@Component,@Service,@Controller,@Repository；
4>标记 DI 注解：@Autowired,@Qualifier,@Resource,@Value；
5><context:component-scan 标签指定注解范围使用**@ComponentScan**(basePackages = {"com.atguigu.components"})替代
6><context:property-placeholder 引入外部配置文件使用**@PropertySource**({"**classpath:**application.properties","classpath:jdbc.properties"})替代；
7><bean 标签使用@Bean 注解和方法实现；
8>**IoC 具体容器实现**选择 **AnnotationConfigApplicationContext** 对象。

4、基于注解方式管理 Bean

4.1 Bean 注解标记和扫描（IoC）

1）组件标记注解和区别

注解	说明
@Component	该注解用于描述 Spring 中的 Bean，它是一个泛化的概念，仅仅表示容器中的一个组件（Bean），并且可以作用在应用的任何层次，例如 Service 层、Dao 层等。使用时只需将该注解标注在相应类上即可。
@Repository	该注解用于将数据访问层（Dao 层）的类标识为 Spring 中的 Bean，其功能与 @Component 相同。
@Service	该注解通常作用在业务层（Service 层），用于将业务层的类标识为 Spring 中的 Bean，其功能与 @Component 相同。
@Controller	该注解通常作用在控制层（如SpringMVC 的 Controller），用于将控制层的类标识为 Spring 中的 Bean，其功能与 @Component 相同。

2）配置文件**确定扫描范围**

```
<!-- 配置自动扫描的包 -->
<!-- 1.包要精准,提高性能!
      2.会扫描指定的包和子包内容
      3.多个包可以使用,分割 例如: com.atguigu.controller,com.atguigu.service等
-->
<context:component-scan base-package="com.atguigu.components"/>
```

```
<!-- 情况三: 指定不扫描的组件 -->
<context:component-scan base-package="com.atguigu.components">

    <!-- context:exclude-filter标签: 指定排除规则 -->
    <!-- type属性: 指定根据什么来进行排除, annotation取值表示根据注解来排除 -->
    <!-- expression属性: 指定排除规则的表达式, 对于注解来说指定全类名即可 -->
    <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

```
<!-- 情况四: 仅扫描指定的组件 -->
<!-- 仅扫描 = 关闭默认规则 + 追加规则 -->
<!-- use-default-filters属性: 取值false表示关闭默认扫描规则 -->
<context:component-scan base-package="com.atguigu.ioc.components" use-default-
filters="false">

    <!-- context:include-filter标签: 指定在原有扫描规则的基础上追加的规则 -->
    <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

3）**组件 BeanName 问题**

类名首字母小写就是 bean 的 id。例如：SoldierController 类对应的 bean 的 id 就是 soldierController。

也可以使用 **value 属性**指定：

```
@Controller(value = "tianDog")
public class SoldierController {
}
```

当注解中只设置一个属性时，value属性的属性名可以省略：

```
@Service("smallDog")
public class SoldierService {
}
```

4.2 组件（Bean）**作用域**和**周期方法注解**

2. 周期方法声明

```
public class BeanOne {

  //周期方法要求: 方法命名随意, 但是要求方法必须是 public void 无形参列表
  @PostConstruct  //注解制指定初始化方法
  public void init() {
    // 初始化逻辑
  }
}

public class BeanTwo {

  @PreDestroy //注解指定销毁方法
  public void cleanup() {
    // 释放资源逻辑
  }
}
```

3. 作用域配置

```
@Scope(scopeName = ConfigurableBeanFactory.SCOPE_SINGLETON) //单例,默认值
@Scope(scopeName = ConfigurableBeanFactory.SCOPE_PROTOTYPE) //多例  二选一
public class BeanOne {

  //周期方法要求: 方法命名随意, 但是要求方法必须是 public void 无形参列表
  @PostConstruct  //注解制指定初始化方法
  public void init() {
    // 初始化逻辑
  }
}
```

4.3 Bean 属性赋值：**引用类型**自动装配（DI）

1）**自动装配**实现：**@Autowired 注解**

在成员变量上直接标记@Autowired注解即可，**无需提供 setXxx()方法**。

2）@Autowired 注解细节
1>标记位置
- 成员变量
(与 xml 进行 bean ref 引用不同，他不需要有 set 方法)

```
    @Autowired
    private SoldierDao soldierDao;
```

- 构造器

```
@Autowired
public SoldierController(SoldierService soldierService) {
    this.soldierService = soldierService;
}
```

- setXxx()方法

```
@Autowired
public void setSoldierService(SoldierService soldierService) {
    this.soldierService = soldierService;
}
```

2> 自动装配的工作流程
首先根据**所需要的组件类型**到 IOC 容器中找。
1>> 能够**找到唯一的 bean**：直接执行装配；
如果完全找不到匹配这个类型的 bean：装配失败；
2>> 和**所需类型匹配的 bean 不止一个**：
- 没有**@Qualifier 注解**：根据**@Autowired 标记位置成员变量的变量名**作为 bean 的 id 进行匹配：
  - 能够找到：执行装配
  - 找不到：装配失败
3>> **使用@Qualifier 注解**：根据**@Qualifier 注解**中**指定的名称**作为 bean 的 id 进行匹配：
  - 能够找到：执行装配
  - 找不到：装配失败

```
@Controller(value = "tianDog")
public class SoldierController {

    @Autowired
    @Qualifier(value = "maomiService222")
    // 根据面向接口编程思想，使用接口类型引入Service组件
    private ISoldierService soldierService;
```

3> 佛系装配
给@Autowired 注解**设置 required=false 属性**表示：能装就装，装不上就不装。

```
@Controller(value = "tianDog")
public class SoldierController {

    // 给@Autowired注解设置required = false属性表示：能装就装，装不上就不装
    @Autowired(required = false)
    private ISoldierService soldierService;
```

4> @Resource 注解 VS @Autowired 注解
>>属于谁
@Resource 注解是 **JDK 扩展包中的**，也就是说属于 JDK 的一部分。所以**该注解是标准注解，更加具有通用性**。(JSR-250 标准中制定的注解类型。JSR 是 Java 规范提案。)
@Autowired 注解是 **Spring 框架自己的**。
>>装配方式（Bean 名称/类型）
@Resource 注解**默认根据 Bean 名称装配**，未指定 name 时，使用属性名作为 name。通过 **name 找不到**的话会**自动启动通过类型装配**。
@Autowired 注解**默认根据类型装配**，如果想**根据名称装配，需要配合 @Qualifier 注解**一起用。
>>标记位置
@Resource 注解用在**属性**上、**setter 方法**上。
@Autowired 注解用在**属性**上、**setter 方法**上、**构造方法**上、**构造方法参数**上。
【补充】**@Resource 注解属于 JDK 扩展包**，所以**不在 JDK 当中，**需要额外引入以下依赖：【高于 JDK11 或低于 JDK8 需要引入以下依赖】

```
<dependency>
    <groupId>jakarta.annotation</groupId>
    <artifactId>jakarta.annotation-api</artifactId>
    <version>2.1.1</version>
</dependency>
```

@Resource使用

```
@Controller
public class XxxController {
    /**
     * 1. 如果没有指定name,先根据属性名查找IoC中组件xxxService
     * 2. 如果没有指定name,并且属性名没有对应的组件,会根据属性类型查找
     * 3. 可以指定name名称查找！ @Resource(name='test') == @Autowired + @Qualifier(value='test')
     */
    @Resource
    private XxxService xxxService;

    //@Resource(name = "指定beanName")
    //private XxxService xxxService;

    public void show(){
        System.out.println("XxxController.show");
        xxxService.show();
    }
}
```

4.4Bean 属性赋值：基本类型属性赋值 (DI)
@Value 通常用于注入外部化属性：

声明外部配置

application.properties

```
catalog.name=MovieCatalog
```

xml引入外部配置

```
<!-- 引入外部配置文件-->
<context:property-placeholder location="application.properties" />
```

@Value注解读取配置

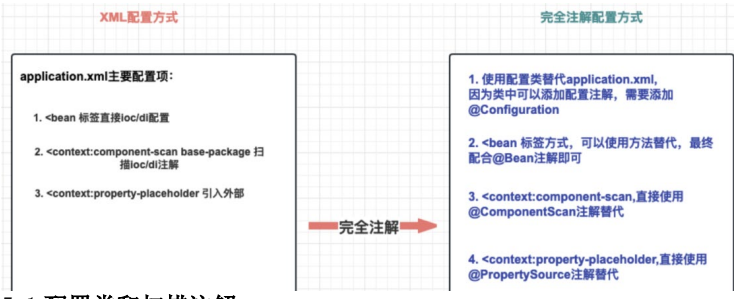
```
package com.atguigu.components;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

/**
 * projectName: com.atguigu.components
 *
 * description: 普通的组件
 */
@Component
public class CommonComponent {

    /**
     * 情况1: ${key} 取外部配置key对应的值!
     * 情况2: ${key:defaultValue} 没有key,可以给与默认值
     */
    @Value("${catalog:hahaha}")
    private String name;

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

Jdbc 的 XML 配置

```
    <!-- 导入外部属性文件 -->
    <context:property-placeholder location="classpath:jdbc.properties" />

    <!-- 配置数据源 -->
    <bean id="druidDataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="url" value="${atguigu.url}"/>
        <property name="driverClassName" value="${atguigu.driver}"/>
        <property name="username" value="${atguigu.username}"/>
        <property name="password" value="${atguigu.password}"/>
    </bean>

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

    <!-- 扫描Ioc/DI注解 -->
    <context:component-scan base-package="com.atguigu.dao,com.atguigu.service,com.atguigu.controller" />
```

5、基于配置类方式管理 Bean

Spring 完全注解配置：是指通过 Java 配置类 代码来配置 Spring 应用程序，使用注解来替代原本在 XML 配置文件中的配置。相对于 XML 配置，完全注解配置具有更强的类型安全性和更好的可读性。



5.1 配置类和扫描注解

```
//标注当前类是配置类，替代application.xml
@Configuration
//使用注解读取外部配置，替代 <context:property-placeholder标签
@PropertySource("classpath:application.properties")
//使用@ComponentScan注解,可以配置扫描包,替代<context:component-scan标签
@ComponentScan(basePackages = {"com.atguigu.components"})
public class MyConfiguration {
```

```
// AnnotationConfigApplicationContext 根据配置类创建 IOC 容器对象
ApplicationContext iocContainerAnnotation =
new AnnotationConfigApplicationContext(MyConfiguration.class);
```

可以使用 no-arg 构造函数实例化 `AnnotationConfigApplicationContext`，然后使用 `register()` 方法对其进行配置。此方法在以编程方式生成 `AnnotationConfigApplicationContext` 时特别有用。以下示例演示如何执行此操作：

```
// AnnotationConfigApplicationContext-IOC容器对象
ApplicationContext iocContainerAnnotation =
new AnnotationConfigApplicationContext();
//外部设置配置类
iocContainerAnnotation.register(MyConfiguration.class);
//刷新后方可生效！！
iocContainerAnnotation.refresh();
```

5.2@Bean 定义组件

需求：将 Druid 连接池对象存储到 IoC 容器；

分析：第三方 jar 包的类，添加到 ioc 容器，无法使用@Component 等相关注解！因为**源码 jar 包内容为只读模式！**

@Bean 注释用于指示方法实例化、配置和初始化要由 Spring IoC 容器管理的新对象。

```
//标注当前类是配置类，替代application.xml
@Configuration
//引入jdbc.properties文件
@PropertySource({"classpath:application.properties","classpath:jdbc.properties"})
@ComponentScan(basePackages = {"com.atguigu.components"})
public class MyConfiguration {

    //如果第三方类进行IoC管理,无法直接使用@Component相关注解
    //解决方案：xml方式可以使用<bean标签
```

```
    //解决方案：配置类方式,可以使用方法返回值+@Bean注解
    @Bean
    public DataSource createDataSource(@Value("${jdbc.user}") String username,
                                       @Value("${jdbc.password}")String password,
                                       @Value("${jdbc.url}")String url,
                                       @Value("${jdbc.driver}")String driverClassName){
        //使用Java代码实例化
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        dataSource.setUrl(url);
        dataSource.setDriverClassName(driverClassName);
        //返回结果即可
        return dataSource;
    }
}
```

5.3 高级特性：@Bean 注解细节

1）@Bean 生成 BeanName 问题

指定@Bean 的名称：@Bean("指定名称")；

缺省情况下，Bean 名称与方法名称相同。

```
@Configuration
public class AppConfig {

  @Bean
  public TransferServiceImpl transferService() {
    return new TransferServiceImpl();
  }
}
```

前面的配置完全等同于下面的Spring XML：

```
<beans>
  <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

2）@Bean 初始化和销毁方法指定

```
public class BeanOne {

  public void init() {
    // initialization logic
  }
}

public class BeanTwo {

  public void cleanup() {
    // destruction logic
  }
}

@Configuration
public class AppConfig {

  @Bean(initMethod = "init")
  public BeanOne beanOne() {
    return new BeanOne();
  }

  @Bean(destroyMethod = "cleanup")
  public BeanTwo beanTwo() {
    return new BeanTwo();
  }
}
```

3）@Bean Scope 作用域

默认作用域为 singleton，但可以使用 **@Scope 注释**覆盖此范围。

```
@Configuration
public class MyConfiguration {

  @Bean
  @Scope("prototype")
  public Encryptor encryptor() {
    // ...
  }
}
```

4）@Bean 方法之间依赖

方案1：

直接调用方法返回 Bean 实例：在一个 @Bean 方法中直接调用其他 @Bean 方法来获取 Bean 实例，虽然是方法调用，也是通过IoC容器获取对应的bean，例如：

```
@Configuration
public class JavaConfig {

    @Bean
    public HappyMachine happyMachine(){
        return new HappyMachine();
    }

    @Bean
    public HappyComponent happyComponent(){
        HappyComponent happyComponent = new HappyComponent();
        //直接调用方法即可!
        happyComponent.setHappyMachine(happyMachine());
        return happyComponent;
    }

}
```

方案2：

参数引用法：通过方法参数传递 Bean 实例的引用来解决 Bean 实例之间的依赖关系，例如：

```
package com.atguigu.config;

import com.atguigu.ioc.HappyComponent;
import com.atguigu.ioc.HappyMachine;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * projectName: com.atguigu.config
 * description: 配置HappyComponent和HappyMachine关系
 */

@Configuration
public class JavaConfig {

    @Bean
    public HappyMachine happyMachine(){
```

```
        return new HappyMachine();
    }

    /**
     * 可以直接在形参列表接收IoC容器中的Bean!
     *     情况1：直接指定类型即可
     *     情况2：如果有多个bean,(HappyMachine 名称 ) 形参名称等于要指定的bean名称!
     *           例如:
     *               @Bean
     *               public Foo foo1(){
     *                   return new Foo();
     *               }
     *               @Bean
     *               public Foo foo2(){
     *                   return new Foo()
     *               }
     *               @Bean
     *               public Component component(Foo foo1 / foo2 通过此处指定引入的bean)
     */
    @Bean
    public HappyComponent happyComponent(HappyMachine happyMachine){
        HappyComponent happyComponent = new HappyComponent();
        //赋值
        happyComponent.setHappyMachine(happyMachine);
        return happyComponent;
    }

}
```

5.4 高级特性：@Import 扩展

@Import 注释允许**从另一个配置类加载@Bean 定义**。

```
@Configuration
public class ConfigA {

  @Bean
  public A a() {
    return new A();
  }
}

@Configuration
@Import(ConfigA.class)
public class ConfigB {

  @Bean
  public B b() {
    return new B();
  }
}
```

作用

现在，在实例化上下文时不需要同时指定 `ConfigA.class` 和 `ConfigB.class`，只需显式提供 `ConfigB`，如以下示例所示：

```
public static void main(String[] args) {
  ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);

  // now both beans A and B will be available...
  A a = ctx.getBean(A.class);
  B b = ctx.getBean(B.class);
}
```

【注意】不同情境下获取 Bean 的方式

1）情景一：

bean 对应的类没有实现任何接口：根据 bean 本身的类型获取 bean；

- 测试：IOC 容器中**同类型的 bean 只有一个**
  正常获取到 IOC 容器中的那个 bean 对象；
- 测试：IOC 容器中**同类型的 bean 有多个**
  会抛出 NoUniqueBeanDefinitionException 异常；

2）情景二：

bean 对应的类实现了接口，这个接口也只有这一个实现类

- 测试：根据**接口类型**获取 bean；
- 测试：根据**类**获取 bean；

结论：上面两种情况其实都能够正常获取到 bean，而且是同一个对象。

3）情景三：

声明一个接口，接口有多个实现类，接口所有实现类都放入 IOC 容器；

- 测试：根据**接口类型**获取 bean
  会抛出 NoUniqueBeanDefinitionException 异常；
- 测试：根据**类**获取 bean
  正常；

4）情景四：

声明一个接口，接口有一个实现类，创建一个切面类，对上面接口的**实现类应用通知**；

- 测试：根据**接口类型**获取 bean
  正常
- 测试：根据**类获取 bean**
  无法获取（原因分析：应用了切面后，**真正放在 IOC 容器中**的是**代理类的对象**目标类并没有被放到 IOC 容器中，所以根据目标类的类型从 IOC 容器中是找不到的）

5）情景五：

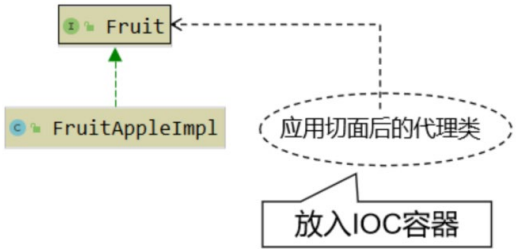
声明一个类，创建一个切面类，对上面的类应用通知；

- 测试：根据**类获取 bean，能获取到**；

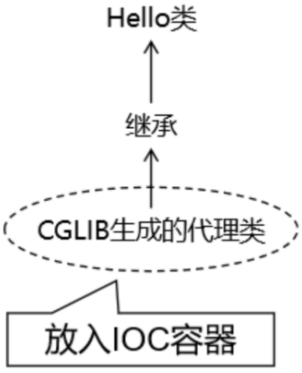
【总结】

如果使用 AOP 技术，目标类有接口，必须使用接口类型接收 IoC 容器中代理组件。

对实现了接口的类应用切面：



对没实现接口的类应用切面：



3、基于 XML 配置方式组件管理

3.1 组件（Bean）信息声明配置（IoC）

1）基于无参数构造函数

```java
<!-- 实验一 [重要]创建bean -->
<bean id="happyComponent" class="com.atguigu.ioc.HappyComponent"/>
```

- bean标签：通过配置bean标签告诉IOC容器需要创建对象的组件信息
- id属性：bean的唯一标识，方便后期获取Bean！
- class属性：组件类的全限定符！
- 注意：要求当前组件类必须包含无参数构造函数！

2）基于静态工厂方法实例化

1. 准备组件类

```java
public class ClientService {
  private static ClientService clientService = new ClientService();
  private ClientService() {}

  public static ClientService createInstance() {
    return clientService;
  }
}
```

```xml
<bean id="clientService"
  class="examples.ClientService"
  factory-method="createInstance"/>
```

- class属性：指定工厂类的全限定符！
- factory-method：指定静态工厂方法，注意，该方法必须是static方法。

3）基于实例工厂方法实例化

1. 准备组建类

```java
public class DefaultServiceLocator {

  private static ClientServiceImpl clientService = new ClientServiceImpl();

  public ClientService createClientServiceInstance() {
    return clientService;
  }
}
```

```xml
<!-- 将工厂类进行ioc配置 -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
</bean>

<!-- 根据工厂对象的实例工厂方法进行实例化组件对象 -->
<bean id="clientService"
  factory-bean="serviceLocator"
  factory-method="createClientServiceInstance"/>
```

- factory-bean属性：指定当前容器中工厂Bean 的名称。
- factory-method：指定实例工厂方法名。注意，实例方法必须是非static的！

3.2 组件（Bean）依赖注入配置（DI）

注入方式有 **setter 方法注入**和**构造函数注入**；注意：引用其他 bean，使用 **ref 属性**。直接注入基本类型值，使用 **value 属性**。

1）基于构造函数的依赖注入（单个构造参数）

2. 准备组件类

```java
public class UserDao {
}
```

```java
public class UserService {

    private UserDao userDao;

    public UserService(UserDao userDao) {
        this.userDao = userDao;
    }
}
```

```xml
<bean id="userService" class="x.y.UserService">
  <!-- 构造函数引用 -->
  <constructor-arg ref="userDao"/>
</bean>
<!-- 被引用类bean声明 -->
<bean id="userDao" class="x.y.UserDao"/>
</beans>
```

2）基于构造函数的依赖注入（多个构造参数）

（多个参数包含**其他 bean** 和**基本数据类型**）

- **constructor-arg 标签**：指定构造参数和对应的值；
- constructor-arg 标签：**name 属性**指定参数名、**index 属性**指定参数角标、**value 属性**指定普通属性值。

```xml
<!-- 场景1：多参数，可以按照相应构造函数的顺序注入数据 -->
<beans>
  <bean id="userService" class="x.y.UserService">
    <!-- value直接注入基本类型值 -->
    <constructor-arg  value="18"/>
    <constructor-arg  value="赵伟风"/>

    <constructor-arg  ref="userDao"/>
  </bean>
  <!-- 被引用类bean声明 -->
  <bean id="userDao" class="x.y.UserDao"/>
</beans>
```

```xml
<!-- 场景2：多参数，可以按照相应构造函数的名称注入数据 -->
<beans>
  <bean id="userService" class="x.y.UserService">
    <!-- value直接注入基本类型值 -->
    <constructor-arg name="name" value="赵伟风"/>
    <constructor-arg name="userDao" ref="userDao"/>
    <constructor-arg name="age"  value="18"/>
  </bean>
  <!-- 被引用类bean声明 -->
  <bean id="userDao" class="x.y.UserDao"/>
</beans>
```

```xml
<!-- 场景2：多参数，可以按照相应构造函数的角标注入数据
        index从0开始 构造函数(0,1,2....)
-->
<beans>
    <bean id="userService" class="x.y.UserService">
    <!-- value直接注入基本类型值 -->
    <constructor-arg index="1" value="赵伟风"/>
    <constructor-arg index="2" ref="userDao"/>
    <constructor-arg index="0"  value="18"/>
  </bean>
  <!-- 被引用类bean声明 -->
  <bean id="userDao" class="x.y.UserDao"/>
</beans>
```

3）基于 Setter 方法依赖注入

- **property 标签**：可以给 setter 方法对应的属性赋值；
- property 标签：**name 属性**代表 set 方法标识、**ref** 代表引用 bean 的标识 id、**value** 属性代表基本属性值。

```xml
<bean id="simpleMovieLister" class="examples.SimpleMovieLister">
  <!-- setter方法，注入movieFinder对象的标识id
       name = 属性名  ref = 引用bean的id值
   -->
  <property name="movieFinder" ref="movieFinder" />

  <!-- setter方法，注入基本数据类型movieName
       name = 属性名 value= 基本类型值
   -->
  <property name="movieName" value="消失的她"/>
</bean>

<bean id="movieFinder" class="examples.MovieFinder"/>
```

3.3 IoC 容器创建和使用

想要配置文件中声明组件类信息真正的进行实例化成 Bean 对象和形成 Bean 之间的引用关系，我们需要**声明 IoC 容器对象，读取配置文件，实例化组件**和**关系维护**的过程都是在 IoC 容器中实现的。

1）容器实例化

```java
//方式1 实例化并且指定配置文件
//参数: String...locations 传入一个或者多个配置文件
ApplicationContext context =
        new ClassPathXmlApplicationContext("services.xml", "daos.xml");

//方式2 先实例化，再指定配置文件，最后刷新容器触发Bean实例化动作 [springmvc源码和ContextLoadListener源码方式]
ApplicationContext context =
        new ClassPathXmlApplicationContext();
//设置配置配置文件,方法参数为可变参数,可以设置一个或者多个配置
iocContainer1.setConfigLocations("services.xml", "daos.xml");
//后配置的文件,需要调用refresh方法,触发刷新配置
iocContainer1.refresh();
```

2）Bean 对象读取

Bean对象读取

3.4 组件（Bean）作用域和周期方法配置

1）周期方法：我们可以在组件类中定义方法，然后当 **IoC 容器实例化**和**销毁组件对象**的时候进行调用。

```xml
<beans>
  <bean id="beanOne" class="examples.BeanOne" init-method="init" />
  <bean id="beanTwo" class="examples.BeanTwo" destroy-method="cleanup" />
</beans>
```

2）组件作用域配置：

<bean>标签声明 Bean，只是将 Bean 的信息配置给 SpringIoC 容器！在 IoC 容器中，这些 <bean>标签对应的信息转成 Spring 内部 BeanDefinition 对象， BeanDefinition 对象内，包含定义的信息（id,class,属性等等）！这意味着， BeanDefinition 与“类”概念一样，SpringIoC 容器可以可以根据 BeanDefinition 对象反射创建多 Bea 对象实例。**具体创建多少个 Bean 的实例对象，**由 Bean 的**作用域 Scope 属性**指定！

```xml
<!--bean的作用域
    准备两个引用关系的组件类即可！！
-->
<!-- scope属性: 取值singleton（默认值），bean在IOC容器中只有一个实例，IOC容器初始化时创建对象 -->
<!-- scope属性: 取值prototype，bean在IOC容器中可以有多个实例，getBean()时创建对象 -->
<bean id="happyMachine8" scope="prototype" class="com.atguigu.ioc.HappyMachine">
    <property name="machineName" value="happyMachine"/>
</bean>

<bean id="happyComponent8" scope="singleton" class="com.atguigu.ioc.HappyComponent">
    <property name="componentName" value="happyComponent"/>
</bean>
```

3.5 FactoryBean 特性和使用

FactoryBean 接口是 Spring IoC 容器实例化逻辑的可插拔性点。用于配置复杂的 Bean 对象，可以将创建过程存储在 FactoryBean 的 getObject 方法！

FactoryBean<T> 接口提供三种方法：

- **T getObject()：**

返回此工厂创建的对象的实例。该返回值会被存储到 IoC 容器！

- **boolean isSingleton()：**

如果此 FactoryBean 返回单例，则返回 true，否则返回 false。此方法的默认实现返回 true（注意，lombok 插件使用，可能影响效果）。

- **Class<?> getObjectType()：**返回 getObject()方法返回的对象类型，如果事先不知道类型，则返回 null。

```java
// 实现FactoryBean接口时需要指定泛型
// 泛型类型就是当前工厂要生产的对象的类型
public class HappyFactoryBean implements FactoryBean<HappyMachine> {

    private String machineName;

    public String getMachineName() {
        return machineName;
    }

    public void setMachineName(String machineName) {
        this.machineName = machineName;
    }

    @Override
    public HappyMachine getObject() throws Exception {

        // 方法内部模拟创建、设置一个对象的复杂过程
        HappyMachine happyMachine = new HappyMachine();

        happyMachine.setMachineName(this.machineName);

        return happyMachine;
    }

    @Override
    public Class<?> getObjectType() {

        // 返回要生产的对象的类型
        return HappyMachine.class;
    }
}
```

```xml
<!-- FactoryBean机制 -->
<!-- 这个bean标签中class属性指定的是HappyFactoryBean，但是将来从这里获取的bean是HappyMachine对象 -->
<bean id="happyMachine7" class="com.atguigu.ioc.HappyFactoryBean">
    <!-- property标签仍然可以用来通过setXxx()方法给属性赋值 -->
    <property name="machineName" value="iceCreamMachine"/>
</bean>
```

```java
@Test
public void testExperiment07()  {

    ApplicationContext iocContainer = new ClassPathXmlApplicationContext("spring-bean-07.xml");

    //注意：直接根据声明FactoryBean的id,获取的是getObject方法返回的对象
    HappyMachine happyMachine = iocContainer.getBean("happyMachine7",HappyMachine.class);
    System.out.println("happyMachine = " + happyMachine);

    //如果想要获取FactoryBean对象，直接在id前添加&符号即可！ &happyMachine7 这是一种固定的约束
    Object bean = iocContainer.getBean("&happyMachine7");
    System.out.println("bean = " + bean);
}
```