> 作者：千锋-索尔
>
> 版本：QF1.0
>
> 版权：千锋Java教研院

一、获取Bean的方式

1.通过类获取bean

```
Product product = context.getBean(Product.class);
```

2.通过bean的id来获取bean

```
Object product = context.getBean("product");
```

3.通过bean的名字来获取bean

- 在xml中添加bean的name

```
<bean class="com.qf.entity.Product" id="product" name="product1"/>
```

- 通过name获取bean

```
Object product1 = context.getBean("product1");
```

4.通过名字+类型来获取bean

- 在xml中添加bean

```
    <bean class="com.qf.entity.Product" id="product" name="product1"/>
    <bean class="com.qf.entity.Product" id="product2" name="product2">
        <property name="name" value="p2"/>
    </bean>
```

- 获取bean

```
Product product2 = context.getBean("product2", Product.class);
```

二、XML文件配置详解

spring的xml配置文件中默认标签有四种：

- beans
- bean

- import
- alias
  其中bean标签为最基础的标签，其他三个标签都是围绕bean标签来做封装或者修饰的

1.beans标签

包含多个bean标签

2.bean标签

表示一个bean

3.import标签

表示导入另外一个xml，xml又包含这些标签

4.alias标签

修饰某个bean的别名

```
<alias name="product1" alias="qfpro"/>
```

- 使用别名获得bean

```
 Object qfpro = context.getBean("qfpro");
```

三、依赖注入

通过依赖注入，在spring创建对象的同时，为其属性赋值。依赖注入的方式有三种：

- Set注入
- 构造函数注入
- 自动注入

1.Set注入

> 创建对象时，Spring工厂会通过Set方法为对象的属性赋值。

1.1 定义目标Bean类型

```
public class User {
    private Integer id;
    private String password;
    private String sex;
    private Integer age;
    private Date bornDate;
    private String[] hobbys;
    private Set<String> phones;
    private List<String> names;
    private Map<String,String> countries;
    private Properties files;
    //Getters And Setters
}
```

1.2 基本类型 + 字符串类型 + 日期类型

```
<bean id="u1" class="com.qf.entity.User">
    <!--base field-->
    <property name="id" value="1001" />
    <property name="password" value="123456" />
    <property name="sex" value="male" />
    <property name="age" value="20" />
    <property name="bornDate" value="1990/1/1" /><!--注意格式"/"-->
</bean>
```

1.3 容器类型

```
<bean id="u1" class="com.qf.entity.User">
  <!--Array-->
    <property name="hobbys">
        <array>
            <value>Run</value>
            <value>Swim</value>
            <value>Climb</value>
        </array>
    </property>

    <!--Set-->
    <property name="phones">
        <set>
            <value>13777777777</value>
            <value>13888888888</value>
            <value>13999999999</value>
        </set>
    </property>

    <!--List-->
    <property name="names">
```

```
        <list>
            <value>tom</value>
            <value>jack</value>
            <value>marry</value>
        </list>
    </property>

    <!--Map-->
    <property name="countries">
        <map>
            <entry key="CN" value="China" />
            <entry key="US" value="America" />
            <entry key="KR" value="Korea" />
        </map>
    </property>

    <!--Properties-->
    <property name="files">
        <props>
            <prop key="first">One</prop>
            <prop key="second">Two</prop>
            <prop key="third">Three</prop>
        </props>
    </property>
</bean>
```

1.4 自建类型

```
<!--次要bean，被作为属性-->
<bean id="addr" class="com.qf.spring.part1.injection.Address">
    <property name="position" value="北京市海淀区" />
    <property name="zipCode" value="100001" />
</bean>

<!--主要bean，操作的主体-->
<bean id="u2" class="com.qf.spring.part1.injection.User">
    <property name="address" ref="addr" /><!--address属性引用addr对象-->
</bean>
```

2. 构造函数注入

> 创建对象时，Spring工厂会通过构造方法为对象的属性赋值。

2.1 定义目标Bean类型

```
public class Student {
    private Integer id;
    private String name;
    private String sex;
    private Integer age;

    //Constructors
    public Student(Integer id , String name , String sex , Integer age){
        this.id = id;
      this.name = name;
        this.sex = sex;
      this.age = age;
    }
}
```

2.2 注入

```
 <!--构造注入-->
<bean id="u3" class="com.qf.entity.Student">
    <constructor-arg name="id" value="1234" /> <!-- 除标签名称有变化，其他均和Set注入一致 -->
    <constructor-arg name="name" value="tom" />
    <constructor-arg name="age" value="20" />
    <constructor-arg name="sex" value="male" />
</bean>
```

3. 自动注入

> 不用在配置中 指定为哪个属性赋值，及赋什么值.
>
> 由spring自动根据某个 "原则" ，在工厂中查找一个bean，为属性注入属性值

```
public class UserServiceImpl implements UserService {
    private UserDAO userDAO;
    //Getters And Setters
    ....
}
```

```
<bean id="userDao" class="com.qf.spring.part1.injection.UserDaoImpl" />
<!-- 为UserServiceImpl中的属性基于类型自动注入值 -->
<bean id="userService" class="com.qf.spring.part1.injection.UserServiceImpl"
autowire="byType"></bean>
```

```
<bean id="userDao" class="com.qf.spring.part1.injection.UserDaoImpl" />
<!-- 为UserServiceImpl中的属性基于类型自动注入值 -->
<bean id="userService" class="com.qf.spring.part1.injection.UserServiceImpl"
autowire="byName"></bean>
```

四、XML高级配置

1.depends-on

bean的加载顺序能够指定吗？可以通过depends-on指明当前bean的加载依赖哪个bean先加载。

如果一个 bean 是另一个 bean 的依赖,通常这个 bean 也就是另一个 bean 的属性之一. 多数情况下,开发者可以在配置 XML 元数据的时候使用元素。然而,有时 bean 之间的依赖不是直接关联的. 例如: 需要调用类的静态实例化器来触发依赖,类似数据库驱动注册. `depends-on` 属性可以显式强制初始化一个或多个 bean. 以下示例使用 `depends-on` 属性表示对单个bean的依赖:

```
<bean class="com.qf.entity.Product" id="product1" name="product1" depends-on="u1"/>
```

2.懒加载

当bean在使用的时候被加载，而不是在Spring容器启动时就加载。此时可以通过设置为懒加载来实现。

默认情况下, `ApplicationContext` 会在实例化的过程中创建和配置所有的单例singleton bean. 总的来说, 这个预初始化是很不错的。因为这样能及时发现环境上的一些配置错误,而不是系统运行了很久之后才发现. 如果这个行为不是迫切需要的,开发者可以通过将 Bean 标记为延迟加载就能阻止这个预初始化 懒加载 bean 会通知 IoC 不要让 bean 预初始化而是在被引用的时候才会实例化.

通过配置`lazy-init="true"`实现。

```
<bean class="com.qf.entity.Product" id="product1" name="product1" depends-on="u1" lazy-
init="true"/>
```

3.bean的作用域

创建 bean 定义时,同时也会定义该如何创建 Bean 实例. 这些具体创建的过程是很重要的,因为它意味着像创建类一样,您可以通过简单的定义来创建许多 bean 的实例.

您不仅可以将不同的依赖注入到 bean 中,还可以配置 bean 的作用域. 这种方法是非常强大而且也非常灵活,开发者可以通过配置来指定对象的作用域,无需在 Java 类的层次上配置. bean 可以配置多种作用域,Spring 框架支持六种作用域,有四种作用域是当开发者使用基于 Web 的 `ApplicationContext` 的时候才有效的. 您还可以创建自定义作用域。

下表描述了支持的作用域:

Scope	Description
singleton	(默认) 每一 Spring IOC 容器都拥有唯一的实例对象.
prototype	一个 Bean 定义可以创建任意多个实例对象.
request	将单个 bean 作用域限定为单个 HTTP 请求的生命周期. 也就是说,每个 HTTP 请求都有自己的 bean 实例,它是在单个 bean 定义的后面创建的. 只有基于 Web 的 Spring `ApplicationContext` 的才可用.
session	将单个 bean 作用域限定为HTTP `Session` 的生命周期. 只有基于 Web 的Spring `ApplicationContext` 的才可用.
application	将单个 bean 作用域限定为 `ServletContext` 的生命周期. 只有基于 Web 的 Spring `ApplicationContext` 的才可用.
websocket	将单个 bean 作用域限定为 `WebSocket` 的生命周期. 只有基于 Web 的 Spring `ApplicationContext` 的才可用.

- singleton: 单例（默认的），同一个id始终只会创建一个bean，节省了内存开销

```
<bean class="com.qf.entity.Product" scope="singleton"></bean>
```

- prototype：多例，每次使用时都会创建一个新的bean。

```
<bean class="com.qf.entity.Product" scope="prototype"></bean>
```

注意：单例可能存在线程安全问题。当多线程并发访问同一对象时，可将bean设置成多例。

4.实例化bean

bean 定义基本上就是用来创建一个或多个对象的配置,当需要 bean 的时候,容器会查找配置并且根据 bean 定义封装的元数据来创建(或获取) 实际对象。

如果你使用基于 XML 的配置,那么可以在 `<bean/>` 元素中通过 `class` 属性来指定对象类型. `class` 属性实际上就是 `BeanDefinition` 实例中的 `class` 属性. 他通常是必需的(一些例外情况,通过实例工厂方法实例化和 Bean 继承的定义。有两种方式使用 `Class` 属性

- 通常情况下,会直接通过反射调用构造方法来创建 bean,这种方式与 Java 代码的 new 创建相似。
- 通过静态工厂方法创建,类中包含静态方法. 通过调用静态方法返回对象的类型可能和 Class 一样,也可能完全不一样.

4.1 通过构造器实例化

当通过构造器创建 Bean 时,Spring 兼容所有可以使用的普通类,也就是说,正在开发的类不需要实现任何特定接口或以特定方式编码. 只要指定 bean 类就足够了. 但是,根据您为该特定 bean 使用的 IoC 类型,您可能需要一个默认(空)构造函数.

Spring IoC 容器几乎可以管理您希望它管理的任何类. 它不仅限于管理真正的 JavaBeans. 大多数 Spring 用户更喜欢管理那些只有一个默认构造函数(无参数) 和有合适的 setter 和 getter 方法的真实的 JavaBeans,还可以在容器中放置更多的外部非 bean 形式(non-bean-style)类,例如: 如果需要使用一个绝对违反 JavaBean 规范的遗留连接池时 Spring 也是可以管理它的.

使用基于 XML 的配置元数据,您可以按如下方式指定 bean 类:

```
<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

4.2 通过静态工厂方法实例化

当采用静态工厂方法创建 bean 时,除了需要指定 class 属性外,还需要通过 `factory-method` 属性来指定创建 bean 实例的工厂方法. Spring 将会调用此方法(其可选参数接下来会介绍) 返回实例对象. 从这样看来,它与通过普通构造器创建类实例没什么两样.

下面的 bean 定义展示了如何通过工厂方法来创建 bean 实例. 注意,此定义并未指定对象的返回类型,只是指定了该类包含的工厂方法,在这个例中, `createInstance()` 必须是一个静态(`static`) 的方法:

```
 <bean class="com.qf.entity.Book" id="book" factory-method="createInstance"/>
```

以下示例显示了一个可以使用前面的 bean 定义的类:

```
public class Book {
    private Book(){}
    private static Book book = new Book();
    public static Book createInstance(){
        return book;
    }
}
```

4.3 通过实例工厂方法实例化

通过调用工厂实例的非静态方法进行实例化与通过静态工厂方法实例化类似, 请将 `class` 属性保留为空,并在 `factory-bean` , 属性中指定当前(或父级或祖先) 容器中 bean 的名称,该容器包含要调用以创建对象的实例方法. 使用 `factory-method` ,属性设置工厂方法本身的名称. 以下示例显示如何配置此类 bean:

```
    <!--定义bean工厂-->
    <bean class="com.qf.entity.BookFactory" id="bookFactory"/>
    <!--通过bean工厂的实例化方法创建bean-->
    <bean id="englishBook" factory-bean="bookFactory" factory-
method="createEnglishBookInstance"/>
    <bean id="javaBook" factory-bean="bookFactory" factory-
method="createJavaBookInstance"/>
```

以下示例显示了相应的 Java 类:

```
public class EnglishBook extends Book{
}
public class JavaBook extends Book{
}


/**
 * @Author: 索尔 QQ: 214490523
 * @技术交流社区:  qfjava.cn
 */
public class BookFactory {

    public Book createEnglishBookInstance(){
        Book book = new EnglishBook();
        book.setId(1001L);
        book.setName("千锋英语");
        return book;
    }

    public Book createJavaBookInstance(){
        Book book = new JavaBook();
        book.setId(1002L);
        book.setName("千锋Java");
        return book;
    }
}
```

5.生命周期回调

可以实现 `InitializingBean` 和 `DisposableBean` 接口,让容器里管理 Bean 的生命周期. 容器会在调用 `afterPropertiesSet()` 之后和 `destroy()` 之前会允许 bean 在初始化和销毁 bean 时执行某些操作.

> **JSR-250 `@PostConstruct` 和 `@PreDestroy` 注解通常被认为是在现代 Spring 应用程序中接收生命周期回调的最佳实践. 使用这些注解意味着您的 bean 不会耦合到特定于 Spring 的接口. 有关详细信息,请参阅使用 @PostConstruct 和 @PreDestroy.如果您不想使用 JSR-250 注解但仍想删除耦合,请考虑使用 `init-method` 和 `destroy-method` 定义对象元数据.**

在内部,Spring 框架使用 `BeanPostProcessor` 实现来处理任何回调接口并调用适当的方法. 如果您需要 Spring 默认提供的自定义功能或其他生命周期行为,您可以自己实现 `BeanPostProcessor`。

除了初始化和销毁方法的回调,Spring 管理的对象也实现了 `Lifecycle` 接口来让管理的对象在容器的生命周期内启动和关闭.

```
package com.qf.entity;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

/**
 * @Author: 索尔 VX: 214490523
 * @技术交流社区:  qfjava.cn
 */
public class Teacher implements InitializingBean, DisposableBean {
    @Override
    public void destroy() throws Exception {
        System.out.println("teacher destroy....");
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("teacher init....");
    }
}
```

5.1 初始化方法回调

`org.springframework.beans.factory.InitializingBean` 接口允许 bean 在所有的必要的依赖配置完成后执行 bean 的初始化, `InitializingBean` 接口中指定使用如下方法:

```
void afterPropertiesSet() throws Exception;
```

Spring 团队是不建议开发者使用 `InitializingBean` 接口,因为这样会将代码耦合到 Spring 的特殊接口上. 他们建议使用`@PostConstruct`注解或者指定一个 POJO 的实现方法, 这会比实现接口更好. 在基于 XML 的元数据配置上,开发者可以使用 `init-method` 属性来指定一个没有参数的方法,使用 Java 配置的开发者可以在 `@Bean` 上添加 `initMethod` 属性。

```
    <!--生命周期回调-->
    <bean class="com.qf.entity.Teacher" id="teacher" init-method="init" destroy-
method="destroy"></bean>
```

```
package com.qf.entity;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

/**
 * @Author: 索尔 QQ:214490523
 * @技术交流社区:  qfjava.cn
 */
public class Teacher{

    public void destroy() throws Exception {
        System.out.println("teacher destroy....");
    }

    public void init(){
        System.out.println("teacher init1");
    }
}
```

5.2 销毁方法的回调

实现 `org.springframework.beans.factory.DisposableBean` 接口的 Bean 就能让容器通过回调来销毁 bean 所引用的资源. `DisposableBean` 接口指定一个方法:

```
void destroy() throws Exception;
```

我们建议您不要使用 `DisposableBean` 回调接口,因为它会不必要地将代码耦合到 Spring. 或者,我们建议使用`@PreDestroy`注解 或指定 bean 定义支持的泛型方法. 在基于 XML 的元数据配置中,您可以在 `<bean/>` 上使用 `destroy-method` 属性. 使用 Java 配置,您可以使用 `@Bean` 的 `destroyMethod` 属性。具体代码参考上一小节。

5.3 默认初始化和销毁方法

当您不使用 Spring 特有的 `InitializingBean` 和 `DisposableBean` 回调接口来实现初始化和销毁方法时,您定义方法的名称最好类似于 `init()`, `initialize()`, `dispose()`. 这样可以在项目中标准化类方法,并让所有开发者都使用一样的名字来确保一致性.

您可以配置 Spring 容器来针对每一个 Bean 都查找这种名字的初始化和销毁回调方法. 也就是说, 任意的开发者都会在应用的类中使用一个叫 `init()` 的初始化回调. 而不需要在每个 bean 中都定义 `init-method="init"` 这种属性, Spring IoC 容器会在 bean 创建的时候调用那个回调方法(如前面描述的标准生命周期一样) . 这个特性也将强制开发者为其他的初始化以及销毁回调方法使用同样的名字.

假设您的初始化回调方法名为 `init()`,而您的 `destroy` 回调方法名为 `destroy()`. 然后,您可以在类似于以下内容的 bean 中使用该类:

```
<beans default-init-method="init">



</beans>
```

顶级 `<beans/>` 元素属性上存在 `default-init-method` 属性会导致 Spring IoC 容器将 bean 类上的 `init` 方法识别为初始化方法回调. 当 bean 被创建和组装时,如果 bean 拥有同名方法的话,则在适当的时候调用它.

您可以使用 `<beans/>` 元素上的 `default-destroy-method` 属性,以类似方式(在 XML 中) 配置destroy方法回调。

5.4 组合生命周期策略

从 Spring 2.5 开始,您有三种选择用于控制 bean 生命周期行为:

- `InitializingBean`和 `DisposableBean`回调接口
- 自定义 `init()` 和 `destroy()` 方法
- `@PostConstruct` 和 `@PreDestroy` 注解。 你也可以在 bean 上同时使用这些机制.

> **如果 bean 配置了多个生命周期机制,而且每个机制都配置了不同的方法名字时,每个配置的方法会按照以下描述的顺序来执行. 但是,如果配置了相同的名字, 例如初始化回调为 `init()`,在不止一个生命周期机制配置为这个方法的情况下,这个方法只会执行一次. 如上一节中所述。**

为同一个 bean 配置的多个生命周期机制具有不同的初始化方法,如下所示:

1. 包含 `@PostConstruct` 注解的方法
2. 在 `InitializingBean` 接口中的 `afterPropertiesSet()` 方法
3. 自定义的 `init()` 方法

Destroy 方法以相同的顺序调用:

1. 包含 `@PreDestroy` 注解的方法
2. 在 `DisposableBean` 接口中的 `destroy()` 方法
3. 自定义的 `destroy()` 方法

6.配置第三方bean

spring不仅仅可以使用在当前项目中创建的类的bean，还可以使用第三方库中的bean。以下以alibaba的druid连接池为例：

- 引入依赖

```
<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.18</version>
</dependency>
```

- 配置bean

```
    <bean class="com.alibaba.druid.pool.DruidDataSource" id="dataSource">
        <property name="username" value="root"/>
        <property name="password" value="root"/>
        <property name="url" value="jdbc:mysql://localhost:3306/my_db"/>
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    </bean>
```

- 使用第三方bean

```
    @Test
    public void test10(){
        ApplicationContext context = new ClassPathXmlApplicationContext("spring-
2.xml");
        Object dataSource = context.getBean("dataSource");
        System.out.println(dataSource);
    }
```