**2019/10/20**

**Spring ICO 容器**

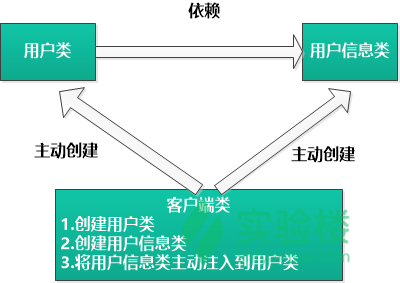
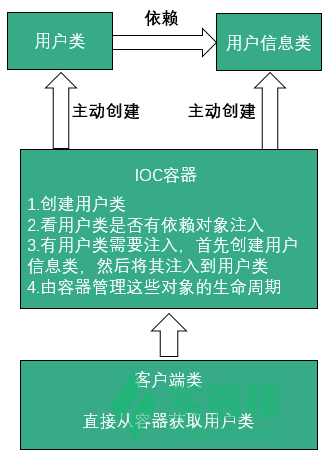
**1.1 IoC 是什么?**

Ioc—**它不是什么技术，而是一种设计思想**。在 Java 开发中， Ioc 意味着将你**设计好的对象交给容器控制**，而不是传统的在你的对象内部直接控制。如何理解好 Ioc 呢？理解好 Ioc 的关键是要明确**“谁控制谁，控制什么，为何是反转**（有反转就应该有正转了），哪些方面反转了”，那我们来深入分析一下：

**谁控制谁，控制什么：传统 Java SE** 程序设计，我们直接在对象内部通过 new 进行创建对象，是程序**主动**去创建依赖对象；**而 IoC 是有专门一个容器来创建这些对象**，即由 IoC 容器来控制对象的创建；谁控制谁？当然是 **IoC 容器控制了对象**；控制什么？那就是主要**控制了外部资源获取（不只是对象包括比如文件等）。**

**为何是反转，哪些方面反转了：**有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而**反转则是由容器来帮忙创建及注入依赖对象**；为何是反转？**因为由容器帮我们查找及注入依赖对象**，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖**对象的获取被反转了**(即不用通过对象本身来创建)。

用图例说明一下，传统程序设计都是主动去创建相关对象然后再组合起来：

当有了 IoC/DI 的容器后，在客户端类中不再主动去创建这些对象了，如图：

#### 1.2 IoC 能做什么？

IoC 不是一种技术，只是一种思想，一个**重要的面向对象编程的法则**，它能指导我们如何设计**出松耦合、更优良**的程序。**传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。**

其实 **IoC 对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。**应用程序原本是老大，要获取什么资源都是主动出击，但是在 IoC/DI 思想中，应用程序就变成被动的了，被动的等待 IoC 容器来创建并注入它所需要的资源了。

IoC 很好的体现了面向对象设计法则之一的好莱坞法则：“别找我们，我们找你”；即**由 IoC 容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。**

#### 1.3 IoC 和 DI

DI—Dependency Injection，即**“依赖注入”**：是**组件之间依赖关系由容器在运行期决定**，形象的说，**即由容器动态的将某个依赖关系注入到组件之中。**依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为**系统搭建一个灵活、可扩展的平台**。通过**依赖注入机制**，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解 DI 的关键是：**“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”**，那我们来深入分析一下：

* 谁依赖于谁：当然是某个容器管理对象依赖于 IoC 容器；“被注入对象的对象”依赖于“依赖对象”；
* 为什么需要依赖：容器管理对象需要 IoC 容器来提供对象需要的外部资源；
* 谁注入谁：很明显是 IoC 容器注入某个对象，也就是注入“依赖对象”；
* 注入了什么：就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）。

IoC 和 DI 有什么关系呢？其实它们是同一个概念的不同角度描述，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以 2004 年大师级人物 Martin Fowler 又给出了一个新的名字：“依赖注入”，相对 IoC 而言，“依赖注入”明确描述了“被注入对象依赖 IoC 容器配置依赖对象”。

#### 1.4 IoC 容器

IoC 容器就是具有依赖注入功能的容器，IoC 容器负责**实例化、定位、配置应用程序中的对象及建立这些对象间的依赖**。应用程序无需直接在代码中 new 相关的对象，应用程序由 IoC 容器进行组装。在 Spring 中 BeanFactory 是 IoC 容器的实际代表者。

**Spring IoC 容器如何知道哪些是它管理的对象呢？**

这就需要配置文件，Spring IoC 容器通过读取配置文件中的配置元数据，通过元数据对应用中的各个对象进行实例化及装配。一般使用基于 xml 配置文件进行配置元数据，而且 Spring 与配置文件完全解耦的，可以使用其他任何可能的方式进行配置元数据，比如注解、基于 java 文件的、基于属性文件的配置都可以。

#### 1.5 bean资源的作用域

在 Spring 中，Bean 的作用域决定了从 Spring 容器中返回的 Bean 实例的类型。在 Spring 中，支持以下 5 种类型的作用域：

1. **singleton — 单例模式**，由 IOC 容器返回一个唯一的 bean 实例。
2. **prototype — 原型模式**，被请求时，每次返回一个新的 bean 实例。
3. **request** — 每个 HTTP Request 请求返回一个唯一的 Bean 实例。
4. **session** — 每个 HTTP Session 返回一个唯一的 Bean 实例。
5. **globalSession** — Http Session 全局 Bean 实例。

注：大多数情况下，你可能只需要处理 Spring 的核心作用域 — 单例模式（ singleton ）和原型模式（ prototype ），默认情况下，作用域是单例模式。

**代码总结：**

**1. 通过IoC容器获取资源对象的三种方式：（如果资源配置于xml文件中）**

Resource resource = *new ClassPathResource*("xmlFileName");BeanFactory beanFactory = *new XmlBeanFactory*(resource);

//此方法已过时

BeanFactory beanFactory1 = *new ClassPathXmlApplicationContext*("xmlFileName");

//这是最常用的用法。

BeanFactory beanFactory2 = *new FileSystemXmlApplicationContext*("xmlFileName ");

**2 bean资源元数据的配置方式:**

**2.1.1 基于XML的配置**

**代码框架:**

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

*<!--bean 一些不同id的bean元素-->*

</beans>

注：每一个不同的bean都由唯一的id属性标识，代表不同的资源，id属性之后指明资源的类型。如果是class，这会调用构造函数，所以在这个bean的子元素中要有足够的信息可以指出需要调用的构造函数。如果有子元素：<constructor-arg index=”intValue” value=”对应的**参数值**” /> , 说明这个bean依赖的对象。

在构造时需要调用的有参数的构造函数，所以类的实现时也要进行定义这类的构造函数。

**2.1.2 通过bean注解 配置资源**

<bean *id*="bean1" *class*="main.java.com.sampletest.JiangJianFeng">

一个bean模块，代表一个特定的资源：类对象，对对象的引用，文件，视频等资源。

如果是类对象：属性为class， 如果是对某个对象的引用，即该对象已经在别的模块配置了，则属性为ref，对应的值为别的bean模块id值。

但同时，这个bean模块代表了这个对象的配置方式，和配置之后的操作。如：

1. 如果是类，通过这个bean模块会调用构造函数，创建对象，构造函数的选择由子属性：<constructor-arg *index*="n" *value*="V"/>决定，如果没有此子属性，则调用默认构造函数!注意：且如果index属性对应的参数的值是对象，则value属性可以用ref引用已用的，或者在此属性下嵌套一个bean字属性，代表index=”n”的这个第n个参数的值由这个嵌套的bean配置！或者使用name指定参数的名称。

2. 如果要改变对象域的值，则需要子属性：<property *name*="对象域的名称" *value*="要设置的域的值"/>，而与构造函数类似，此时在类的定义中必须为该实例域定义更改器方法：setXxx()方法。

3. 如果bean元素拥有属性：scope="typeName",着表示设置了bean资源的作用域，tpyeName的值为上述五种类型之一，默认值是singleton。如<bean *id*="bean1" *class*="main.java.com.sampletest.JiangJianFeng" scope="prototype">,表示原型模式，每次通过该bean获取资源时都会构造单独的资源。

**2.2 基于@bean注解的配置**

**2.2.1 在需要的资源前面添加注解：如资源的HelloWorld的java类定义。**

*@Configuration* //使用注解@Configuration 告诉Spring， 这个Class是Spring的核心配置文件，并且通过使用注解@Bean定义bean

*public class* HelloWorld {

//类的定义

*@Bean*(name = "beanId")  
 *public* HelloWorld getHello(){  
 HelloWorld helloWorld = *new HelloWorld*();  
 helloWorld.setName("jjf");  
 *return* helloWorld;

//其他定义  
 }}

bean注解只能定义紧跟着有返回值的方法之前才有可能是正确的，当使用通过这个beanId唯一标识的注解来获取资源师，这个注解紧跟着的方法就会被调用，返回值就是需要的资源。

@Bean注解往往定义在阅读器方法之前，即：getter方法

**2.2.2 获取 @bean注解 指定的资源：**

ApplicationContext context = *new AnnotationConfigApplicationContext*(HelloWorld.*class*);  
HelloWorld obj = context.getBean("bean1",HelloWorld.*class*);  
HelloWorld obj1 = (HelloWorld) context.getBean("bean2");

注：*AnnotationConfigApplicationContex*类的参数表示定位资源的位置为：HelloWorld类，因为这个类内部已经有@benan注解，所以有机会获得需要的资源。再通过getBean方法调用相应方法返回需要的资源。

**3 通过bean配置集合类型的资源**

**3.1 配置List 和 Set**

*private* List<Object> lists ;

/\*如果某个class资源中有如上数据域List，则为了在这个class资源被创建之后，这个实例域不会为null，希望在配置文件中通过property 属性为其设置值，则必需要声明如下构造器方法：\*/  
*public void* setLists(List<Object> lists) {  
 *this*.lists = lists;  
}

此时对应的xml配置文件正确的bean资源中添加如下字元素<property name=”” />定义格式如下：

<property *name*="lists">  
 <util:list>  
 <value>1</value> <!--第1个值 --><ref *bean*="othersBean" /> <!--第2个值：引用别的bean -->  
 <value>2</value> <!--第3个值 -->  
 <bean *class*="com.shiyanlou.spring.collections.Person"> <!--第4个值 -->  
 <property *name*="name" *value*="shiyanlouList" />  
 <property *name*="address" *value*="chengdu" />  
 <property *name*="age" *value*="25" />  
 </bean>  
 </util:list>  
</property>

注：这种结构我们可定义为：set-property配置！

**3.2 配置Map**

原始类的定义和List，Set的一样，较大的区别是xml配置文件：在对应的bean元素中添加如下子元素：

即value子元素改为entry子元素。

<property *name*="maps">  
 <util:map>  
 <entry *key*="Key 1" *value*="1" />*<!--一个 entry 就是一个 Map 元素-->* <entry *key*="Key 2" *value-ref*="othersBean" />  
 <entry *key*="Key 3">  
 <bean *class*="com.shiyanlou.spring.collections.Person">  
 <property *name*="name" *value*="shiyanlouMap" />  
 <property *name*="address" *value*="chengdu" />  
 <property *name*="age" *value*="25" />  
 </bean>  
 </entry>  
 </util:map>  
</property>

**3.2 配置Properties**

原始类的定义和List，Set的一样，较大的区别是xml配置文件：在对应的bean元素中添加如下子元素：

即value子元素改为entry子元素。

<property *name*="pros">*<!-- Properties 类型类似于Map 类型的特例，Map 元素的键值可以对应任何类型的对象，但是Properties只能是字符串-->* <props>  
 <prop *key*="admin">admin@nospam.com</prop>  
 <prop *key*="support">support@nospam.com</prop>  
 </props>  
</property>

</property>