关于Spring的IoC容器总结

1 概述

1.1 IoC简介

> **控制反转**（Inversion of Control，缩写为**IoC**），是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。

IoC不是一种技术， 只是一种思想，能够指导我们设计出更优良的程序。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的高耦合，难于测试；有了IoC容器后，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，从而降低了对象之间的耦合度，同时也方便测试，利于功能复用，并使得程序的整体结构变得更加灵活。

1.2 IoC常见的两种方式

> IoC最常见的方式叫做依赖注入（Dependency Injection，简称**DI**），还有一种方式叫依赖查找（Dependency LookUp）。

《Spring揭秘》中把IoC和依赖注入看做等同的概念进行讲解，网上对"依赖查找"的介绍比较少，wiki上对"依赖查找"的介绍只有一句话：

> 依赖查找更加主动，在需要的时候通过调用框架提供的方法来获取对象，获取时需要提供相关的配置文件路径、key等信息来确定获取对象的状态。

所以下面都是把IoC和DI看成是等同概念。

1.3 三种依赖注入的方式

1.3.1 构造方法注入

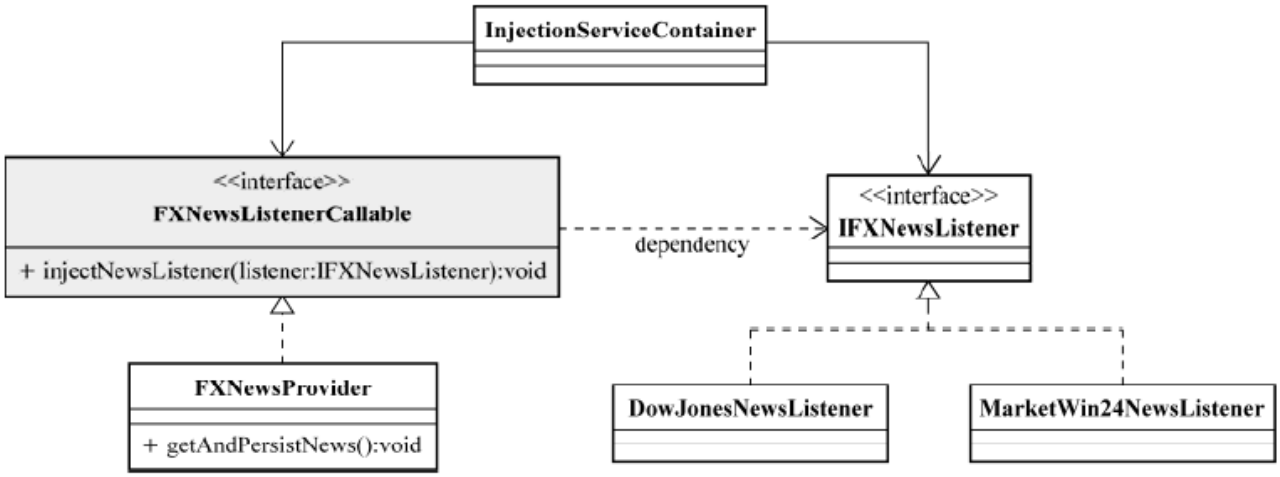
```
public FXNewsProvider(IFXNewsListener newsListner,IFXNewsPersister newsPersister)
{
   this.newsListener    = newsListner;
   this.newPersistener = newsPersister;
}
```

IoC Service Provider会检查被注入对象的构造方法，取得它所需要的依赖对象列表，进而为其注入相应的对象。

1.3.2 Setter方法注入

当前对象为其依赖对象所对应的属性添加setter方法，从而可以通过setter方法将相应的依赖对象设置到被注入对象中。

1.3.3 接口注入



如上图所示，FXNewsProvider为了让IoC Service Provider为其注入所依赖的IFXNewsListener，首先需要实现FXNewsListenerCallable接口，这个接口会声明一个injectNewsListener()方法，该方法的参数就是所依赖对象的类型。这样，InjectionServiceContainer对象就可以通过这个接口方法将依赖对象注入到被注入对象FXNewsProvider。

1.3.4 三种注入方式的比较

- 构造方法注入：优点是对象在构造完成之后，就已进入就绪状态，可以马上使用。缺点是，当依赖对象较多时，构造方法的参数列表会比较长。而通过反射构造对象时，对相同类型的参数的处理会比较困难，维护和使用上也比较麻烦，对于非必须的依赖处理，可能需要引入多个构造方法，参数数量的变动可能造成维护上的不便。
- setter方法注入：方法可以命名，所以setter方法注入在描述性上要比构造方法注入好一些。缺点是对象无法再构造完成后马上进行就绪状态。
- 接口注入：接口注入因为强制被注入对象实现不必要的接口，带有侵入性，现在基本处于“退役状态”。

2 IoC Service Provider

2.1 IoC Service Provider的职责

- **业务对象的构建管理：**IoC Service Provider需要将对象的构建逻辑从客户端对象哪里剥离出来，以免这部分逻辑污染业务对象。
- **业务对象间的依赖绑定：**IoC Service Provider通过结合之前构建和管理的所有业务对象，以及各个业务对象间可以识别的依赖关系，将这些对象所依赖的对象注入绑定，从而保证每个业务对象在使用的时候，可以处于就绪状态。

2.2 IoC Service Provider管理对象间依赖关系的方式

2.2.1 直接编码方式

```
IoContainer container = ...;
container.register(FXNewsProvider.class,new FXNewsProvider());
container.register(IFXNewsListener.class,new DowJonesNewsListener());
...
FXNewsProvider newsProvider = (FXNewsProvider)container.get(FXNewsProvider.class);
newProvider.getAndPersistNews();
```

在容器启动前，可以通过程序编码的方式将被注入对象和依赖对象注册到容器中，并明确他们相互之间的依赖注入关系。当我们要这种类型的对象实例时，就可以告知IoC容器，让其将容器中注册的对应哪个具体实例返回过来。

2.2.2 配置文件方式

这是一种较为普遍的依赖注入关系管理方式。普通文本文件、properties文件、XML文件等，都可以成为管理依赖注入关系的载体，但最为常见的是通过XML文件来管理对象注册和对象间依赖关系。

2.2.3 元数据方式

元数据方式的代表实现是Google Guice，我们可以直接在类中使用元数据信息来标注各个对象之间的依赖关系，然后由Guice框架根据这些注解所提供的信息将这些对象组装后，交给客户端对象使用。

```
public class FXNewsProvider
{
   private IFXNewsListener  newsListener;
   private IFXNewsPersister newPersistener;
   @Inject
   public FXNewsProvider(IFXNewsListener listener,IFXNewsPersister persister)
   {
      this.newsListener   = listener;
      this.newPersistener = persister;
   }
   ...
}
```

通过@Inject，指明需要IoC Service Provider通过构造方法注入方式，为FXNewsProvider注入其所依赖的对象。余下的依赖相关信息，在Guice中由相应的Module提供。

```
public class NewsBindingModule extends AbstractModule
{
   @Override
   protected void configure() {
      bind(IFXNewsListener.class) ➥
      .to(DowJonesNewsListener.class).in(Scopes.SINGLETON);
      bind(IFXNewsPersister.class) ➥
      .to(DowJonesNewsPersister.class).in(Scopes.SINGLETON);
   }
}
```

通过Module指定进一步的依赖注入相关信息之后，我们就可以直接从Guice哪里取得最终已经注入完毕，直接可用的对象了。

3 BeanFactory

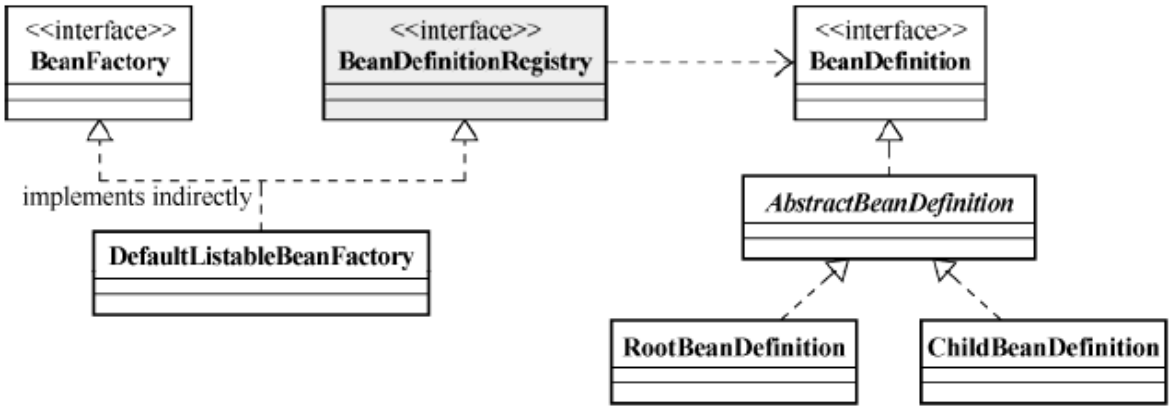
3.1 BeanFactory的对象注册与依赖绑定方式

BeanFactory作为一个IoC Service Provider，为了能够明确管理各个业务对象以及业务对象之间的依赖绑定关系，同样需要某种途径来记录和管理这些信息。

3.1.1 直接编码方式

BeanFactory只是一个接口，我们需要一个该接口的实现来进行实际的Bean的管理，DefaultListableBeanFactory就是一个比较通用的实现类。DefaultListableBeanFactory除了间接地实现了BeanFactory接口，还实现了BeanDefinitionRegistry接口，该接口才是在BeanFactory的实现中担当Bean注册管理的橘色。BeanFactory接口只定义如何访问容器内管理的Bean的方法，各个BeanFactory的具体实现类负责具体Bean的注册以及管理工作。

BeanDefinitionRegistry接口定义抽象了Bean的注册逻辑。



每一个受管的对象，在容器中都会有一个BeanDefinition的实例与之相对应，该BeanDefinition的实例负责保存对象的所有必要信息。当客户端向BeanFactory请求相应对象的时候，BeanFactory会通过这些信息为客户端返回一个完备可用的对象实例。

3.1.2 外部配置文件方式

Spring的IoC容器支持两种配置文件格式：Properties文件格式和XML文件格式。

采用外部配置文件时，Spring的IoC容器有一个统一的处理方式。通常，需要根据不同的外部配置文件，给出相应的BeanDefinitionReader实现类，由BeanDefinitionReader的相应实现类负责将相应的配置文件内容读取并映射到BeanDefinition，然后将映射后的BeanDefinition注册到一个BeanDefinitionRegistry，之后，BeanDefinition即完成Bean的注册和加载。整个过程类似于如下代码:

```
BeanDefinitionRegistry beanRegistry = <某个BeanDefinitionRegistry的实现类，通常为
DefaultListableBeanFactory>;
BeanDefinitionReader beanDefinitionReader = new BeanDefinitionReaderImpl(beanRegistry);
beanDefinitionReader.loadBeanDefinitions("配置文件路径");
```

3.1.3 注解方式

Spring 2.5以及Java 5以后，可以使用@Autowired 和@Component对相关类进行标记。其中，@Autowired用于告知Spring容器需要为当前对象注入哪些依赖对象，而@Component则是配合Spring 2.5中新的classpath-scanning功能使用的。

现在只需再向Spring的配置文件中增加一个“触发器”，被标注的类就能获得依赖对象的注入了。

```
<?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" ➥
xmlns:tx="http://www.springframework.org/schema/tx" ➥
xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd ➥
http://www.springframework.org/schema/context ➥
http://www.springframework.org/schema/context/spring-context-2.5.xsd ➥
http://www.springframework.org/schema/tx ➥
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
<context:component-scan base-package="cn.spring21.project.base.package"/>
</beans>
```

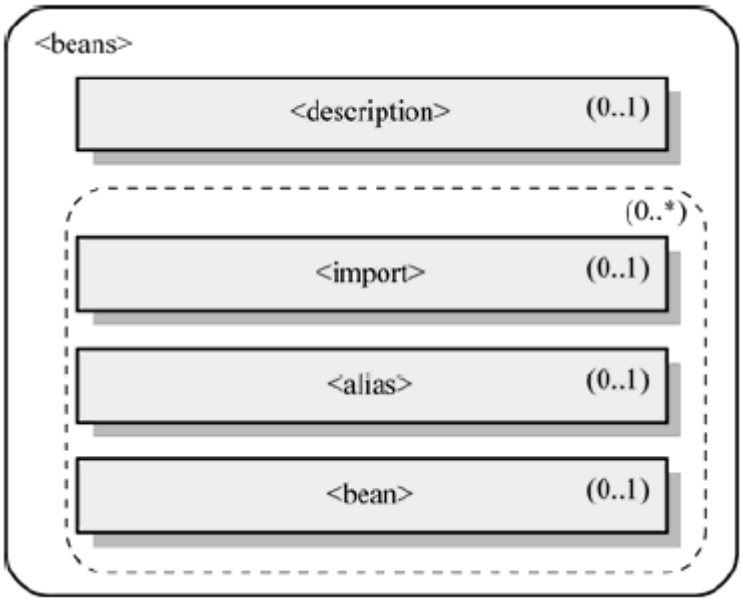
context:component-scan会到指定的package下面扫描标注有@Component的类，如果找到，则将他们添加到容器进行管理，并根据它们所标注的@Autowired为这些类注入符合条件的依赖对象。

3.2 BeanFactory的XML信息管理方式

XMl格式的信息管理方式是Spring提供的最为强大、支持最为全面的方式。

3.2.1 bean与beans

bean：所有注册到容器的业务对象，在Spring中称之为Bean。容器最终可以管理所有的业务对象，所以在XML中把这些叫做<bean>的元素组织起来，叫做<beans>。<beans>是XML配置文件中最顶层的元素，它下面可以包含0或1个<description>和多个<bean>以及<import>或者<alias>。



3.2.2 XML配置

```
<bean id="djNewsListener" class="...impl.DowJonesNewsListener">
</bean>
```

- id属性

id属性是用来指定当前注册对象的beanName是什么，以与其他注册到统一容器的对象区分开来。也可以用name属性来指定<bean>的别名，区别是name属性尅使用id不能使用的一些字符。

- class属性

  class属性用来指定当前注册对象的类型。

3.2.3 处理各个对象之间的依赖

大部分情况下，各个业务对象之间会相互协作来完成同一使命，这个时候彼此之间的相互依赖就无法避免。而业务对象现在都符合IoC规则，所以可以通过构造方法注入和setter方法注入来解决。

3.2.3.1 构造方法注入

```
<bean id="djNewsProvider" class="..FXNewsProvider">
   <constructor-arg>
      <ref bean="djNewsListener"/>
   </constructor-arg>
   <constructor-arg>
      <ref bean="djNewsPersister"/>
   </constructor-arg>
</bean>
```

通过构造方法注入方式，为当前业务对象注入其所依赖的对象，需要使用<constructor-arg>，<ref>用来指定所引用的bean实例。对于无法明确配置项与对象的构造方法参数列表的一一对应关系时，可以使用type或者index来解决。

- type属性可以通过指定构造方法的参数类型来区分不同的构造方法。
- index属性可以指定构造器各个参数的序号。

3.2.3.2 setter方法注入

Spring为setter方法注入提供了<property>元素，如图：

```
<bean id="djNewsProvider" class="..FXNewsProvider">
   <property name="newsListener">
      <ref bean="djNewsListener"/>
   </property>
   <property name="newPersistener">
      <ref bean="djNewsPersister"/>
   </property>
</bean>
```

其中，name属性用来指定该<peoperty>将会注入的对象所对应的实例变量名称，然后同福哦value或者ref或者内嵌的其他元素来指定具体的依赖对象值或引用。

3.2.4 可用的配置项

Spring不仅可以为当前对象注入简单数据类型或者某个对象的引用，还能够注入其他多种类型，如bean、idref、null、list、set、map、props，这里举两个例子。

- **<idref>：**如果要为当前对象注入所依赖的对象的名称，而不是引用，通常可以使用<value>来达到目的，但使用<idref>更加合适，因为使用<idref>，容器在解析配置的时候就会检查这个beanName是否存在，而不用等到运行时才发现这个beanName对应的对象实例不存在。
- **<list>:**<list>对应注入对象类型为java.util.List及其子类或者数组类型的依赖对象。

```
public class MockDemoObject{
  private List param1;
  private String[] param2;
  //相应的getter和setter方法
  ...
}
```

例如为MockDemoObject注入List和String数组，其配置类似于下面：

```
<property name="param1">
  <list>
    <value>something</value>
    <ref bean="someBeanName"></ref>
    <bean class="someClass"></bean>
  </list>
</property>
<property>
    <list>
      <value>stringValue1</value>
      <value>stringValue2</value>
  </list>
</property>
```

3.2.5 bean的scope

BeanFactory除了拥有作为IoC Service Provider的职责，作为一个轻量级容器，它还有着其他一些职责，其中就包括对象的生命周期管理。

scope用来声明容器中的对象所应该处的限定场景或者说该对象的存活时间，即容器在对象进入相应的scope之前，生成并装配这些对象，在该对象不再处于这些scope的限定之后，容器通常会销毁这些对象。

3.2.5.1 五种bean的scope类型

- **singleton：**标记为拥有singleton的对象定义，在Spring的IoC容器中只存在一个实例，所有对该对象的引用将共享这个实例。该实例从容器启动，并因为第一次被请求而初始化之后，将一直存活到容器退出。
- **prototype：**针对声明为prototype scope的bean定义，容器在接到该类型对象的请求的时候，会每次都重新生成一个新的对象实例给请求方。对象实例返回给请求方之后，容器就不再拥有当前返回对象的引用，请求方需要自己负责当前返回对象的后记生命周期的管理工作，包括该对象的销毁。

  只有在支持Web应用的ApplicationContext中使用下面这三种scope才是合理的。
- **request：**Spring容器，即XmlApplicationContext会为每个HTTP请求创建一个全新的RequestProcessor对象供当前请求使用，当请求结束后，该对象实例的生命周期即告结束。
- **session:**Spring容器会为每个独立的session创建属于它们自己的全新的UserPreference对象实例。
- **global session：**只有在基于portlet的Web应用程序中才有意义，它映射到portlet的global范围的session。

3.2.6 工厂方法与FactoryBean

3.2.6.1 静态工厂方法

```
public class StaticBarInterfaceFactory{
  public static BarInterface getInstance(){
    return new BarInterfaceImpl();
  }
}
```

为了将该静态工厂方法返回的实现注入Foo，可以使用以下方式进行配置（采用setter方式为Foo注入BarInterface的实例）：

```
<bean>
    <property name="barInterface">
        <ref bean="bar"></ref>
  </property>
</bean>

<bean id="bar" class="...StaticBarInterfaceFactory" factory-method="getInstance">
</bean>
```

class指定静态方法工厂类，factory-method指定工厂方法名称，然后，容器调用该静态方法工厂类的指定工厂方法（getInstance），并返回方法调用后的结果，即BarInterfaceImpl的实例。

3.2.6.2 非静态工厂方法

```
public class NotStaticBarInterfaceFactory{
  public BarInterface getInstance(){
    return new BarInterfaceImpl();
  }
}
```

对于非静态工厂方法，容器在调用该方法之前需要先获得一个NotStaticBarInterfaceFactory实例，配置内容如下：

```
<bean id="foo" class="...Foo">
    <property name="barInterface">
        <ref bean="bar">
      </ref>
  </property>
</bean>

<bean id="barFactory" class="...NotStaticBarInterfaceFactroy"></bean>

<bean id="bar" factory-bean="barfactory" factory-method="getInstance"></bean>
```