Spring 框架简介

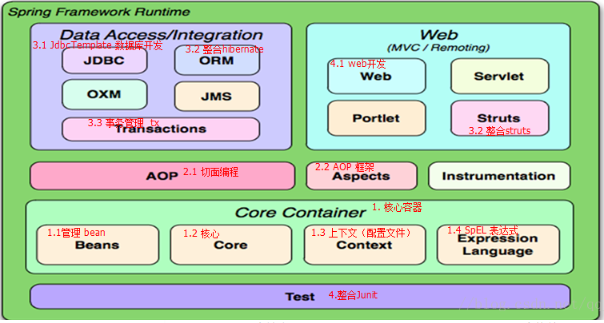
Spring是J2EE应用程序框架，是轻量级IOC和AOP容器框架，主要针对Java Bean的生命周期进行管理(创建、配置和管理)的轻量级容器，可以单独使用，也可以和Struts、Ibatis框架等组合使用，其架构如下图所示：



Spring框架是一个分层架构，由7个定义良好的模块组成，每个模块都可以单独存在，或者与其他一个或者多个模块共同使用，其模块功能如下：

1. Spring Core，核心容器提供Spring框架的基本功能，主要组件是BeanFactory，它是工厂模式的实现，其使用IOC模式将将应用程序的配置和依赖性规范与实际的应用程序代码分开。核心容器中包含Core、Beans、Context和Spring Expression Language，其中Core和Beans是Spring框架的基础部分，提供IoC控制反转和依赖注入的特性
2. Spring Context，Spring上下文是一个配置文件，向Spring框架提供上下文信息，例如JNDI、EJB、检验和调度功能
3. Spring AOP，可以通过配置管理特性，Sping AOP模块直接将面向方面的编程功能集成到Spring框架中。该模块为基于Spring应用程序中的对象提供事务管理服务
4. Spring DAO，JDBC DAO抽象层提供有意义的异常层次结构，可用该结构来管理异常处理和不同数据库供应商抛出的错误消息
5. Spring ORM，在Spring框架中有若干个ORM框架，从而提供了ORM的对象关系工具，其中包括JDO、Hibernate和iBatis SQL Map，所有这些都遵循Spring通用事务和DAO异常层次结构
6. Spring Web模块，Web上下文模块建立在应用程序上下文模块之上，为基于Web的应用程序提供了上下文。所以Spring 框架支持与Jakarata Struts的基础。Web模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作
7. Spring MVC框架，全功能的构建Web应用程序的MVC实现，通过策略接口，MVC框架变成了高度可配置的，MVC容纳了大量视图技术，其中包括JSP、Velocity、Tiles和POI等

Spring框架是分层架构，包含一系列的功能并被分为大约20个模块，Core Container、Data Access/Integration、Web、AOP(Aspect Oriented Programming)及Instrumentation和测试部分，如下图所示：



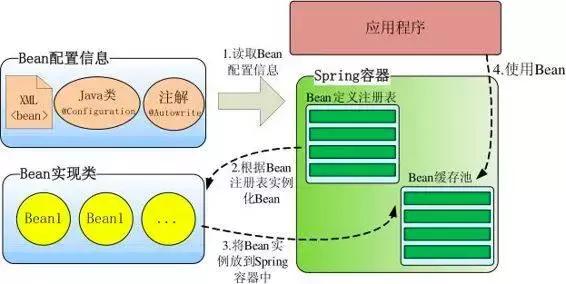
下面对核心组件进行介绍。

# Spring IoC和依赖注入(DI)

1. **控制反转(IoC)**

控制反转模式(IoC)是一种设计思想，其不创建对象，仅描述创建它们的方式，例如在配置文件中描述哪一个组件需要哪一项服务。控制反转是将创建对象的控制权进行转移，原来创建对象的主动权在程序中把控，现在将这种权利转移到第三方。

Spring IoC是轻量级IoC容器，创建容器核心是通过让容器感知创建对象与对象(Bean)的关系，其中Bean是Spring框架的核心概念，Spring容器可以理解为工厂，而Bean是该工厂的产品。Spring容器根据各种形式的Bean配置信息在容器内部建立Bean定义注册表，然后根据注册表加载、实例化Bean并建立Bean之间的依赖关系，最后将其放入Bean缓存池中，以供外层应用程序的调用，如下图所示：



Spring提供了三种Bean定义方法，其中一种是基于注解的Bean定义，其IoC操作如下：

* @Component，用于实现IoC控制的反转，属性Value用于实现对象id定义
* @Scope，用于实现单例/多例、Request及Session等配置
* @Controller，用于Controller层的注解
* @Service，用于实现Service的注解
* @Repository，用于实现持久层的注解

1. **依赖注入(DI)**

依赖注入(Dependency Injection)是控制反转的一种实现方式，其和控制反转(IoC)是从不同的角度描述同一件事情，即通过引入IoC容器利用依赖关系注入的方式将其注入到对象中，从而实现对象之间的解耦。在Spring中支持的注入方式主要有两种: Setter注入喝构造器注入。在Spring编程中，常见的DI操作有以下两种：

* @Resource(name=”userService”)，按照name和type的方式检索
* @Autowired，自动装配，按照类实现注入

Service Loader是控制反转的另一种实现。

1. **反射机制**

Spring IoC的实现原理是Java反射机制和工厂模式，Spring使用BeanFactory作为工厂类，其读取配置文件，利用反射机制注入对象。

1. Java反射机制

IoC最基本的技术就是反射(Reflection)编程，其就是根据给出的类名(字符串)来动态生成对象，这种编程方式可以让对象在生成时才决定对象的类型。

反射是Java语言的一个重要的特性，主要是指程序可以访问，检测和修改它本身状态或行为的一种能力，并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。在Java中，只要给定类的名字，那么就可以通过反射机制来获得类的所有信息，其示例如下：

*c = Class.forName(className);*

*object = c.newInstance();*

*if(bean.getProperties() != null) {*

*for(Property p : bean.getProperties()) {*

*if (p.getValue() != null) {*

*Method getMethod = BeanUtil.getSetterMethod(object, p.getName());*

*getMethod.invoke(object, p.getValue());*

*}*

*...*

*}*

*}*

1. Spring IoC与反射

在Spring IoC中要生成的对象可以在配置文件中给出定义如下所示：

*<bean id="teacher" class="com.fys.spring.basic.Teacher">*

*<property name="name" value="Jerry"/>*

*</bean>*

IoC容器在启动时将配置文件加载到BeanDefinition中，其定义如下：

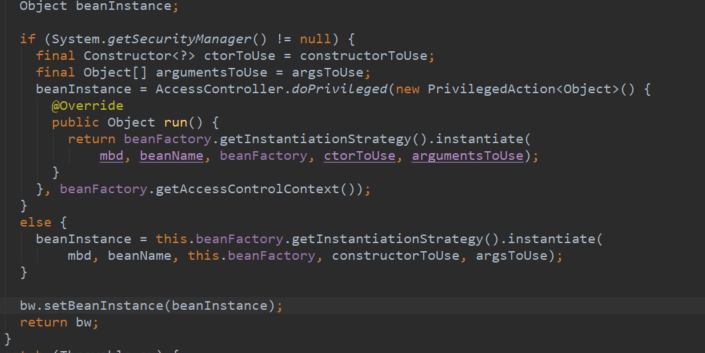
在配置中定义的Bean会转换成BeanDefiniton存在Spring的BeanFactory中，Bean在代码层面上是BeanDefinition的实例，其保存了Bean指向的类、是否单例、是否懒加载及依赖关系等，其接口定义如下：



核心属性：

* SCOPE\_SINGLETON和SCOPE\_PROTOTYPE，默认提供Sington和ProtoType两种类型
* parentName: 父Bean，涉及到Bean继承
* beanClassName: bean的类名称
* Scope，bean的scope
* lazyInit，是否懒加载
* dependsOn，依赖的Bean
* autowireCandidate，设置该Bean是否可以注入到其他Bean中
* factoryBeanName，如果bean使用工厂方法生成，指定工厂名称
* constructorArgumentValues，构造器参数

Java Bean的最终实现是通过BeanFactory，其读取BeanDefinition并使用反射技术来生成对象，核心代码如下所示：



https://www.cnblogs.com/baizhanshi/p/6755716.html

# Spring AOP，面向切面编程

AOP(Aspect Oriented Programming)，面向切面编程，通过预编译的方式和运行期动态代理实现程序功能的统一维护技术。Spring面向切面编程的实现由两种方式：动态代理和CGLib。Spring AOP是纯Java实现，不需要专门的编译过程和类加载器，在运行期通过代理方式向目标类植入增强代码。

AOP是OOP(面向对象编程)的延续，是Spring框架的重要内容，是函数式编程的衍生泛型，利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高开发效率。AOP的核心概念：

1. 切入点，Pointcut，在哪些类哪些方法上切入
2. 通知，Advice，在方法的执行的哪个时机做什么功能
3. 切面，切入点+通知，在什么时机什么地方，执行什么增强
4. 织入，Weaving，把切面加入到对象中，并创建出代理对象的过程

## 2.1 AOP使用示例

下面介绍其AOP的使用示例：

1. 定义业务类

*@Component("landlord")*

*public class Landlord {*

*public void service(){ //连接点，增强这个方法*

*System.out.println("Landlord : Contact, the rent");*

*}*

*}*

1. 定义Broker类

*@Component //切面的类定义为Bean，需要@Component注解标注*

*@Aspect //创建切面，可以理解为拦截器，当程序运行到连接点时进行拦截*

*public class Broker {*

*//定义切点，execution正则表达式，判断拦截的类及方法*

*@Before("execution(\* com.fys.aop.pojo.Landlord.service())")*

*public void before() {*

*System.out.println("Broker: Bring a Renter, and Give the price");*

*}*

*@After("execution(\* com.fys.aop.pojo.Landlord.service())")*

*public void after() {*

*System.out.println("Borker: Hand the Key");*

*}*

*}*

1. 配置applicationContext.xml中配置自动注入，配置Spring IoC容器去扫描Bean

*<context:component-scan base-package="com.fys.aop.aspect" />*

*<context:component-scan base-package="com.fys.aop.pojo" />*

*<aop:aspectj-autoproxy/>*

1. 测试代码

*public class TestSpring {*

*public static void main(String[] args) {*

*ApplicationContext context =*

*new ClassPathXmlApplicationContext("applicationContext.xml");*

*Landlord landlord = (Landlord)context.getBean("landlord", Landlord.class);*

*landlord.service();*

*}*

*}*

执行结果如下：

*Broker: Bring a Renter, and Give the price*

*Landlord : Contact, the rent*

*Borker: Hand the Key*

另外一种使用方式是使用XML配置Spring AOP，上面的示例如下：

*<!-- 装配 Bean-->*

*<bean name="landlord" class="com.fys.aop.pojo.Landlord"/>*

*<bean id="broker" class="com.fys.aop.aspect.Broker"/>*

*<!-- 配置AOP -->*

*<aop:config>*

*<!-- where：在哪些地方（包.类.方法）做增加 -->*

*<aop:pointcut id="landlordPoint"*

*expression="execution(\* com.fys.aop.pojo.Landlord.service())"/>*

*<!-- what:做什么增强 -->*

*<aop:aspect id="logAspect" ref="broker">*

*<!-- when:在什么时机（方法前/后/前后） -->*

*<aop:around pointcut-ref="landlordPoint" method="before"/>*

*</aop:aspect>*

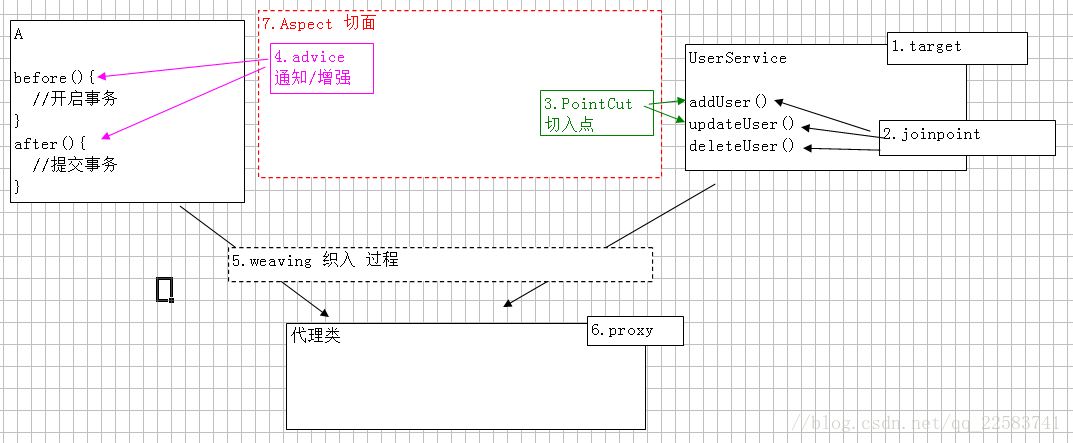
*</aop:config>*

执行结果如下：

*Broker: Bring a Renter, and Give the price*

## **2.2 AOP原理-自定义**

AOP底层将采用代理机制进行实现，其执行原理如下：



1. 定义目标类，接口+实现类

*public interface UserService {*

*public void sayHello(String user);*

*}*

实现类UserServiceImpl，定义如下：

*public class UserServiceImpl implements UserService {*

*public void sayHello(String user) {*

*System.out.println("Hello " + user);*

*}*

*}*

1. 切面类，用于存通知MyAspect

*public class MyAspect {*

*public void before() {*

*System.out.println("Invoked Before from MyAspect");*

*}*

*public void after(){*

*System.out.println("Invoked After from MyAspect");*

*}*

*}*

1. 工厂类，编写工厂类生成代理

public class MyBeanFactory {

*public static UserService createService() {*

*final UserService userService = new UserServiceImpl();*

*final MyAspect myAspect = new MyAspect();*

*UserService proxyService = (UserService) Proxy.newProxyInstance(*

*MyBeanFactory.class.getClassLoader(),*

*userService.getClass().getInterfaces(),*

*new InvocationHandler() {*

*public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {*

*myAspect.before();*

*Object obj = method.invoke(userService, args);*

*myAspect.after();*

*return obj;*

*}*

*}*

*);*

*return proxyService;*

*}*

*}*

1. 测试类

*UserService userService = MyBeanFactoryByCglib.createService();*

*userService.sayHello("Tom");*

输出结果如下：

*Invoked Before from MyAspect*

*Hello Tom*

*Invoked After from MyAspect*

Spring AOP的另外一种实现是通过CGlib类，其定义如下：

*public class MyBeanFactoryByCglib {*

*public static UserService createService() {*

*final UserServiceImpl userService = new UserServiceImpl();*

*final MyAspect myAspect = new MyAspect();*

*Enhancer enhancer = new Enhancer();*

*enhancer.setSuperclass(userService.getClass());*

*enhancer.setCallback(*

*new MethodInterceptor() {*

*public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {*

*myAspect.before();*

*Object obj = method.invoke(userService, args);*

*methodProxy.invokeSuper(proxy, args);*

*myAspect.after();*

*return obj;*

*}*

*}*

*);*

*UserServiceImpl proxyService = (UserServiceImpl) enhancer.create();*

*return proxyService;*

*}*

*}*

https://www.cnblogs.com/cdf-opensource-007/p/6443314.html

https://blog.csdn.net/qq\_22583741/article/details/79589910#2-%E5%85%A5%E9%97%A8%E6%A1%88%E4%BE%8Bioc

<https://www.jianshu.com/p/994027425b44>

<https://spring.io/>

<http://www.th7.cn/Program/java/201611/1021639.shtml>

https://www.ibm.com/developerworks/cn/java/wa-spring1/

https://blog.csdn.net/umbrellasoft/article/details/81805954

Spring MVC

Spring WebFlux <= Reactive Stack