目录

目录
Spring 源码解析
- Spring扩展点机制
  - Spring中的InstantiationAwareBeanPostProcessor和BeanPostProcessor的区别
    - 接口介绍
    - 举例说明
      - 创建接口实现类
      - 创建目标类
      - 配置文件注册
      - 测试
    - 分析相关方法
      - postProcessBeforeInstantiation
      - postProcessAfterInstantiation
      - postProcessPropertyValues
    - 最后总结
- Spring AOP源码解析
  - aop 执行过程描述
  - aop调用过程原理图
  - aop源码详解
    - @EnableAspectJAutoProxy注解
    - AspectJAutoProxyRegistrar类导入了什么？
    - registerAspectJAnnotationAutoProxyCreatorIfNecessary方法详解
    - AnnotationAwareAspectJAutoProxyCreator 做了什么事？
      - 我们先来分析顶层父类AbstractAutoProxyCreator
- Bean 后置处理器是如何实例化的？
  - 1.创建IOC容器
  - 2.registerBeanPostProcessors 向容器中注册Bean 后置处理器
  - 3.BeanPostProcessor实例如何被创建？
  - 4.getSingleton 创建Or获取单例bean
  - 5.重点debug图像
    - beanFactory为何物
    - 容器中的后置处理器
    - beanDedinition为何物
- Spring 事务机制
- Spring IOC容器初始化过程

Spring 源码解析

Spring扩展点机制

Spring中的InstantiationAwareBeanPostProcessor和BeanPostProcessor的区别

接口介绍

先来看下接口的源代码：

```
package org.springframework.beans.factory.config;
public interface InstantiationAwareBeanPostProcessor extends BeanPostProcessor {

    Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName)
throws BeansException;

    boolean postProcessAfterInstantiation(Object bean, String beanName) throws
BeansException;

    PropertyValues postProcessPropertyValues(
            PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String
beanName) throws BeansException;

}
```

从源码中我们可以获知的信息是该接口除了具有父接口中的两个方法以外还自己额外定义了三个方法。所以该接口一共定义了5个方法，这5个方法的作用分别是。

---

postProcessBeforeInitialization

BeanPostProcessor接口中的方法,在Bean的自定义初始化方法之前执行

---

postProcessAfterInitialization

BeanPostProcessor接口中的方法 在Bean的自定义初始化方法执行完成之后执行

---

postProcessBeforeInstantiation

是最先执行的方法，它在目标对象实例化之前调用，该方法的返回值类型是Object，我们可以返回任何类型的值。由于这个时候目标对象还未实例化，所以这个返回值可以用来代替原本该生成的目标对象的实例(比如代理对象)。如果该方法的返回值代替原本该生成的目标对象，后续只有postProcessAfterInitialization方法会调用，其它方法不再调用；否则按照正常的流程走

---

postProcessAfterInstantiation

在目标对象实例化之后调用，这个时候对象已经被实例化，但是该实例的属性还未被设置，都是null。因为它的返回值是决定要不要调用postProcessPropertyValues方法的其中一个因素（因为还有一个因素是mbd.getDependencyCheck()）；如果该方法返回false,并且不需要check，那么postProcessPropertyValues就会被忽略不执行；如果返回true，postProcessPropertyValues就会被执行

---

postProcessPropertyValues

对属性值进行修改，如果postProcessAfterInstantiation方法返回false，该方法可能不会被调用。可以在该方法内对属性值进行修改

注意两个单词

Instantiation 表示实例化,对象还未生成 Initialization 表示初始化,对象已经生成

举例说明

创建接口实现类

```
/**
 * 自定义处理器
 * @author dengp
 *
 */
public class MyInstantiationAwareBeanPostProcessor implements
InstantiationAwareBeanPostProcessor{

    /**
     * BeanPostProcessor接口中的方法
     * 在Bean的自定义初始化方法之前执行
     * Bean对象已经存在了
     */
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
throws BeansException {
        // TODO Auto-generated method stub
        System.out.println(">>postProcessBeforeInitialization");
        return bean;
    }

    /**
     * BeanPostProcessor接口中的方法
     * 在Bean的自定义初始化方法执行完成之后执行
     * Bean对象已经存在了
     */
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
        System.out.println("<<postProcessAfterInitialization");
        return bean;
    }

    /**
     * InstantiationAwareBeanPostProcessor中自定义的方法
     * 在方法实例化之前执行   Bean对象还没有
     */
    @Override
    public Object postProcessBeforeInstantiation(Class<?> beanClass, String
beanName) throws BeansException {
        System.out.println("--->postProcessBeforeInstantiation");
        return null;
    }

    /**
     * InstantiationAwareBeanPostProcessor中自定义的方法
     * 在方法实例化之后执行   Bean对象已经创建出来了
     */
    @Override
    public boolean postProcessAfterInstantiation(Object bean, String beanName)
throws BeansException {
        System.out.println("<---postProcessAfterInstantiation");
```

```
        return true;
    }

    /**
     * InstantiationAwareBeanPostProcessor中自定义的方法
     * 可以用来修改Bean中属性的内容
     */
    @Override
    public PropertyValues postProcessPropertyValues(PropertyValues pvs,
PropertyDescriptor[] pds, Object bean,
            String beanName) throws BeansException {
        System.out.println("<---postProcessPropertyValues--->");
        return pvs;
    }
}
```

创建目标类

```
public class User {

    private int id;

    private String name;

    private String beanName;

    public User(){
        System.out.println("User 被实例化");
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        System.out.println("设置: "+name);
        this.name = name;
    }

    public String getBeanName() {
        return beanName;
    }

    public void setBeanName(String beanName) {
        this.beanName = beanName;
    }
    public void start(){
        System.out.println("自定义初始化的方法....");
    }
```

```
    @Override
    public String toString() {
        return "User [id=" + id + ", name=" + name + ", beanName=" + beanName +
"]";
    }
}
```

配置文件注册

```
<?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 class="com.dpb.pojo.User" id="user" init-method="start">
        <property name="name" value="波波烤鸭"></property>
    </bean>

    <!-- 注册InstantiationAwareBeanPostProcessor对象 -->
    <bean class="com.dpb.processor.MyInstantiationAwareBeanPostProcessor">
</bean>
</beans>
```

测试

```
@Test
public void test1() {
    ClassPathXmlApplicationContext ac = new
ClassPathXmlApplicationContext("applicationContext.xml");
    User user = ac.getBean(User.class);
    System.out.println(user);
    // 关闭销毁
    ac.registerShutdownHook();
}
```

输出结果

```
--->postProcessBeforeInstantiation
User 被实例化
<---postProcessAfterInstantiation
<---postProcessPropertyValues--->
设置：波波烤鸭
>>postProcessBeforeInitialization
自定义初始化的方法....
<<postProcessAfterInitialization
User [id=0, name=波波烤鸭, beanName=null]
```

通过输出结果，我们可以看到几个方法的执行顺序，而且五个方法都执行了，那么每个方法的返回结果对其他方法有什么影响没有呢，接下来分别看下这几个方法。

分析相关方法

postProcessBeforeInstantiation

该方法返回的结果如果为null,后面的方法都正常执行了，但是如果该方法返回了实例对象了呢？我们来看下

```
/**
 * InstantiationAwareBeanPostProcessor中自定义的方法 在方法实例化之前执行 Bean对象还没有
 */
@Override
public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException {
    System.out.println("--->postProcessBeforeInstantiation");
    // 利用cglib动态代理生成对象返回
    if (beanClass == User.class) {
        Enhancer e = new Enhancer();
        e.setSuperclass(beanClass);
        e.setCallback(new MethodInterceptor() {
            @Override
            public Object intercept(Object obj, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {

                System.out.println("目标方法执行前:" + method + "\n");
                Object object = methodProxy.invokeSuper(obj, objects);
                System.out.println("目标方法执行后:" + method + "\n");
                return object;
            }
        });
        User user = (User) e.create();
        // 返回代理类
        return user;
    }
    return null;
}
```

测试输出结果：

```
容器开始初始化....
--->postProcessBeforeInstantiation
User 被实例化
<<postProcessAfterInitialization
容器初始化结束....
```

通过数据结果我们发现，postProcessBeforeInstantiation方法返回实例对象后跳过了对象的初始化操作，直接执行了postProcessAfterInitialization(该方法在自定义初始化方法执行完成之后执行)，跳过了postProcessAfterInstantiation，postProcessPropertyValues以及自定义的初始化方法(start方法)，为什么会这样呢？我们要来查看下源代码。

在AbstractBeanFactory中的对InstantiationAwareBeanPostProcessor实现该接口的BeanPostProcessor 设置了标志

```
@Override
public void addBeanPostProcessor(BeanPostProcessor beanPostProcessor) {
    Assert.notNull(beanPostProcessor, "BeanPostProcessor must not be null");
    this.beanPostProcessors.remove(beanPostProcessor);
    this.beanPostProcessors.add(beanPostProcessor);
    if (beanPostProcessor instanceof InstantiationAwareBeanPostProcessor) {
        this.hasInstantiationAwareBeanPostProcessors = true;
    }
    if (beanPostProcessor instanceof DestructionAwareBeanPostProcessor) {
        this.hasDestructionAwareBeanPostProcessors = true;
    }
}
```

在AbstractAutowireCapableBeanFactory类中有个createBean方法，

```
protected Object createBean(String beanName, RootBeanDefinition mbd, Object[]
args) throws BeanCreationException {
       // ... 省略
    try {
        // Give BeanPostProcessors a chance to return a proxy instead of the
target bean instance.
        Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
        if (bean != null) {
                return bean;
        }
       // ... 省略
    Object beanInstance = doCreateBean(beanName, mbdToUse, args);
    if (logger.isDebugEnabled()) {
        logger.debug("Finished creating instance of bean '" + beanName + "'");
    }
    return beanInstance;
}
```

Object bean = resolveBeforeInstantiation(beanName, mbdToUse);这行代码之后之后根据bean判断如果不为空null就直接返回了，而不执行doCreateBean()方法了，而该方法是创建Bean对象的方法。

```
protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition
mbd) {
    Object bean = null;
    // //如果beforeInstantiationResolved还没有设置或者是false（说明还没有需要在实例化前
执行的操作）
    if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
        // 判断是否有注册过InstantiationAwareBeanPostProcessor类型的bean
        if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
            Class<?> targetType = determineTargetType(beanName, mbd);
            if (targetType != null) {
                bean = applyBeanPostProcessorsBeforeInstantiation(targetType,
beanName);
                if (bean != null) {
                    // 直接执行自定义初始化完成后的方法,跳过了其他几个方法
                    bean = applyBeanPostProcessorsAfterInitialization(bean,
beanName);
                }
            }
        }
```

```
            mbd.beforeInstantiationResolved = (bean != null);
        }
        return bean;
    }
    protected Object applyBeanPostProcessorsBeforeInstantiation(Class<?> beanClass,
    String beanName) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp =
    (InstantiationAwareBeanPostProcessor) bp;
                Object result = ibp.postProcessBeforeInstantiation(beanClass,
    beanName);
                //只要有一个result不为null；后面的所有 后置处理器的方法就不执行了，直接返回(所
    以执行顺序很重要)
                if (result != null) {
                    return result;
                }
            }
        }
        return null;
    }
    @Override
    public Object applyBeanPostProcessorsAfterInitialization(Object existingBean,
    String beanName)
            throws BeansException {

        Object result = existingBean;
        for (BeanPostProcessor processor : getBeanPostProcessors()) {
            result = processor.postProcessAfterInitialization(result, beanName);
             //如果返回null；后面的所有 后置处理器的方法就不执行，直接返回(所以执行顺序很重要)
            if (result == null) {
                return result;
            }
        }
        return result;
    }
```

代码说明：

1. 如果postProcessBeforeInstantiation方法返回了Object是null;那么就直接返回，调用doCreateBean方法();
2. 如果postProcessBeforeInstantiation返回不为null;说明修改了bean对象;然后这个时候就立马执行postProcessAfterInitialization方法(注意这个是初始化之后的方法,也就是通过这个方法实例化了之后，直接执行初始化之后的方法;中间的实例化之后 和 初始化之前都不执行);
3. 在调用postProcessAfterInitialization方法时候如果返回null;那么就直接返回，调用doCreateBean方法();(初始化之后的方法返回了null,那就需要调用doCreateBean生成对象了)
4. 在调用postProcessAfterInitialization时返回不为null;那这个bean就直接返回给ioc容器了初始化之后的操作是这里面最后一个方法了；

postProcessAfterInstantiation

同样在

```
protected void populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper
bw) {
    // 省略 。。
```

```
    boolean continueWithPropertyPopulation = true;

    if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
                // 此处执行 postProcessAfterInstantiation方法
                if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(),
beanName)) {
                    // postProcessAfterInstantiation 返回true与false决定
                    // continueWithPropertyPopulation
                    continueWithPropertyPopulation = false;
                    break;
                }
            }
        }
    }
    // postProcessAfterInstantiation false
    // continueWithPropertyPopulation 就为false 然后该方法就结束了！！！
    if (!continueWithPropertyPopulation) {
        return;
    }
    // 省略 ...
    boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
    boolean needsDepCheck = (mbd.getDependencyCheck() !=
RootBeanDefinition.DEPENDENCY_CHECK_NONE);

    if (hasInstAwareBpps || needsDepCheck) {
        PropertyDescriptor[] filteredPds =
filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
        if (hasInstAwareBpps) {
            for (BeanPostProcessor bp : getBeanPostProcessors()) {
                if (bp instanceof InstantiationAwareBeanPostProcessor) {
                    InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
                    // 调用 postProcessPropertyValues方法
                    pvs = ibp.postProcessPropertyValues(pvs, filteredPds,
bw.getWrappedInstance(), beanName);
                    if (pvs == null) {
                        return;
                    }
                }
            }
        }
        if (needsDepCheck) {
            checkDependencies(beanName, mbd, filteredPds, pvs);
        }
    }
    // 真正设置属性的方法。
    applyPropertyValues(beanName, mbd, bw, pvs);
}
```

这个postProcessAfterInstantiation返回值要注意，因为它的返回值是决定要不要调用postProcessPropertyValues方法的其中一个因素（因为还有一个因素是mbd.getDependencyCheck());如果该方法返回false,并且不需要check，那么postProcessPropertyValues就会被忽略不执行；如果返true，postProcessPropertyValues就会被执行

postProcessPropertyValues

在populateBean方法中我们已经看到了postProcessPropertyValues执行的位置了。我们来看下postProcessPropertyValues的效果

```
/**
 * InstantiationAwareBeanPostProcessor中自定义的方法 可以用来修改Bean中属性的内容
 */
@Override
public PropertyValues postProcessPropertyValues(PropertyValues pvs,
PropertyDescriptor[] pds, Object bean,
        String beanName) throws BeansException {
    System.out.println("<---postProcessPropertyValues--->");
    if(bean instanceof User){
        PropertyValue value = pvs.getPropertyValue("name");
        System.out.println("修改前name的值是:"+value.getValue());
        value.setConvertedValue("bobo");
    }
    return pvs;
}
```

配置文件中设值注入name属性

```
<?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 class="com.dpb.pojo.User" id="user" init-method="start">
        <property name="name" value="波波烤鸭"></property>
    </bean>

    <!-- 注册InstantiationAwareBeanPostProcessor对象 -->
    <bean class="com.dpb.processor.MyInstantiationAwareBeanPostProcessor">
</bean>
</beans>
```

测试看输出

```
--->postProcessBeforeInstantiation
User 被实例化
<---postProcessAfterInstantiation
<---postProcessPropertyValues--->
修改前name的值是:TypedStringValue: value [波波烤鸭], target type [null]
设置：bobo
>>postProcessBeforeInitialization
自定义初始化的方法....
<<postProcessAfterInitialization
User [id=0, name=bobo, beanName=null]
容器初始化结束....
```

name的属性值被修改了。该方法执行的条件要注意postProcessAfterInstantiation返回true且postProcessBeforeInstantiation返回null。

最后总结

1. InstantiationAwareBeanPostProcessor接口继承BeanPostProcessor接口，它内部提供了3个方法，再加上BeanPostProcessor接口内部的2个方法，所以实现这个接口需要实现5个方法。InstantiationAwareBeanPostProcessor接口的主要作用在于目标对象的实例化过程中需要处理的事情，包括实例化对象的前后过程以及实例的属性设置
2. postProcessBeforeInstantiation方法是最先执行的方法，它在目标对象实例化之前调用，该方法的返回值类型是Object，我们可以返回任何类型的值。由于这个时候目标对象还未实例化，所以这个返回值可以用来代替原本该生成的目标对象的实例(比如代理对象)。如果该方法的返回值代替原本该生成的目标对象，后续只有postProcessAfterInitialization方法会调用，其它方法不再调用；否则按照正常的流程走
3. postProcessAfterInstantiation方法在目标对象实例化之后调用，这个时候对象已经被实例化，但是该实例的属性还未被设置，都是null。因为它的返回值是决定要不要调用postProcessPropertyValues方法的其中一个因素（因为还有一个因素是mbd.getDependencyCheck()）；如果该方法返回false,并且不需要check，那么postProcessPropertyValues就会被忽略不执行；如果返回true, postProcessPropertyValues就会被执行
4. postProcessPropertyValues方法对属性值进行修改(这个时候属性值还未被设置，但是我们可以修改原本该设置进去的属性值)。如果postProcessAfterInstantiation方法返回false，该方法可能不会被调用。可以在该方法内对属性值进行修改
5. 父接口BeanPostProcessor的2个方法postProcessBeforeInitialization和postProcessAfterInitialization都是在目标对象被实例化之后，并且属性也被设置之后调用的。

Spring AOP源码解析

aop 执行过程描述

```
/**
 * AOP：【动态代理】
 *       指在程序运行期间动态的将某段代码切入到指定方法指定位置进行运行的编程方式；
 *
 * 1、导入aop模块；Spring AOP：(spring-aspects)
 * 2、定义一个业务逻辑类（MathCalculator）；在业务逻辑运行的时候将日志进行打印（方法之前、方法运行结束、方法出现异常，xxx）
```

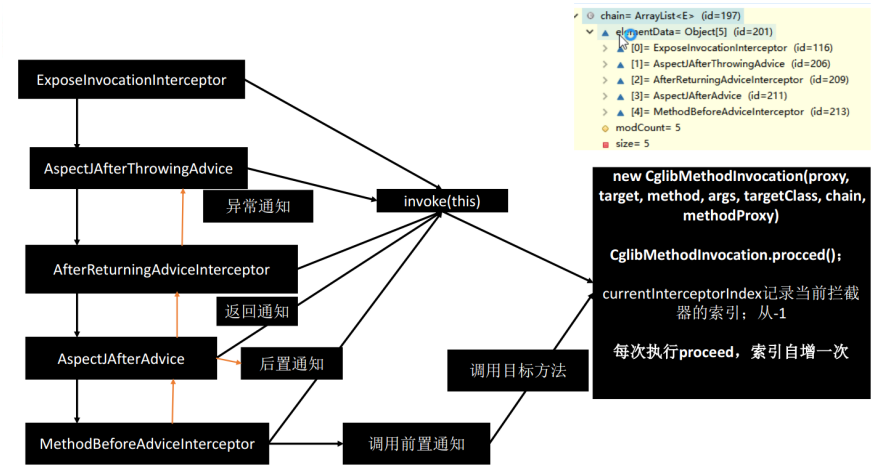
```
 * 3、定义一个日志切面类（LogAspects）：切面类里面的方法需要动态感知MathCalculator.div运行到哪里然后执行；
 *       通知方法：
 *           前置通知(@Before)：logStart：在目标方法(div)运行之前运行
 *           后置通知(@After)：logEnd：在目标方法(div)运行结束之后运行（无论方法正常结束还是异常结束）
 *           返回通知(@AfterReturning)：logReturn：在目标方法(div)正常返回之后运行
 *           异常通知(@AfterThrowing)：logException：在目标方法(div)出现异常以后运行
 *           环绕通知(@Around)：动态代理，手动推进目标方法运行（joinPoint.procced()）
 * 4、给切面类的目标方法标注何时何地运行（通知注解）；
 * 5、将切面类和业务逻辑类（目标方法所在类）都加入到容器中;
 * 6、必须告诉Spring哪个类是切面类(给切面类上加一个注解：@Aspect)
 * [7]、给配置类中加 @EnableAspectJAutoProxy 【开启基于注解的aop模式】
 *      在Spring中很多的 @EnableXXX;
 *
 * 三步：
 *  1）、将业务逻辑组件和切面类都加入到容器中；告诉Spring哪个是切面类（@Aspect）
 *  2）、在切面类上的每一个通知方法上标注通知注解，告诉Spring何时何地运行（切入点表达式）
 *  3）、开启基于注解的aop模式：@EnableAspectJAutoProxy
 *
 * AOP原理：【看给容器中注册了什么组件，这个组件什么时候工作，这个组件的功能是什么？】
 *      @EnableAspectJAutoProxy;
 * 1、@EnableAspectJAutoProxy是什么？
 *      @Import(AspectJAutoProxyRegistrar.class)：给容器中导入AspectJAutoProxyRegistrar
 *          利用AspectJAutoProxyRegistrar自定义给容器中注册bean；BeanDefinetion
 *          internalAutoProxyCreator=AnnotationAwareAspectJAutoProxyCreator
 *
 *      给容器中注册一个AnnotationAwareAspectJAutoProxyCreator;
 *
 * 2、 AnnotationAwareAspectJAutoProxyCreator:
 *      AnnotationAwareAspectJAutoProxyCreator
 *          ->AspectJAwareAdvisorAutoProxyCreator
 *              ->AbstractAdvisorAutoProxyCreator
 *                  ->AbstractAutoProxyCreator
 *                          implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware
 *                      关注后置处理器（在bean初始化完成前后做事情）、自动装配BeanFactory
 *
 * AbstractAutoProxyCreator.setBeanFactory()
 * AbstractAutoProxyCreator.有后置处理器的逻辑;
 *
 * AbstractAdvisorAutoProxyCreator.setBeanFactory()-》initBeanFactory()
 *
 * AnnotationAwareAspectJAutoProxyCreator.initBeanFactory()
 *
 *
 * 流程：
 *      1）、传入配置类，创建ioc容器
 *      2）、注册配置类，调用refresh（）刷新容器；
 *      3）、registerBeanPostProcessors(beanFactory);注册bean的后置处理器来方便拦截bean的创建；
 *          1）、先获取ioc容器已经定义了的需要创建对象的所有BeanPostProcessor
 *          2）、给容器中加别的BeanPostProcessor
 *          3）、优先注册实现了PriorityOrdered接口的BeanPostProcessor;
 *          4）、再给容器中注册实现了Ordered接口的BeanPostProcessor;
 *          5）、注册没实现优先级接口的BeanPostProcessor;
```

```
 *          6）、注册BeanPostProcessor，实际上就是创建BeanPostProcessor对象，保存在容器
中；
 *              创建internalAutoProxyCreator的
BeanPostProcessor【AnnotationAwareAspectJAutoProxyCreator】
 *              1）、创建Bean的实例
 *              2）、populateBean；给bean的各种属性赋值
 *              3）、initializeBean：初始化bean；
 *                      1）、invokeAwareMethods()：处理Aware接口的方法回调
 *                      2）、applyBeanPostProcessorsBeforeInitialization()：应用后
置处理器的postProcessBeforeInitialization（）
 *                      3）、invokeInitMethods()；执行自定义的初始化方法
 *                      4）、applyBeanPostProcessorsAfterInitialization()；执行后
置处理器的postProcessAfterInitialization（）；
 *              4）、BeanPostProcessor(AnnotationAwareAspectJAutoProxyCreator)创
建成功；--》aspectJAdvisorsBuilder
 *          7）、把BeanPostProcessor注册到BeanFactory中；
 *              beanFactory.addBeanPostProcessor(postProcessor);
 * =======以上是创建和注册AnnotationAwareAspectJAutoProxyCreator的过程========
 *
 *          AnnotationAwareAspectJAutoProxyCreator =>
InstantiationAwareBeanPostProcessor
 *      4）、finishBeanFactoryInitialization(beanFactory);完成BeanFactory初始化工
作；创建剩下的单实例bean
 *          1）、遍历获取容器中所有的Bean，依次创建对象getBean(beanName);
 *              getBean->doGetBean()->getSingleton()->
 *          2）、创建bean
 *              【AnnotationAwareAspectJAutoProxyCreator在所有bean创建之前会有一个拦
截，InstantiationAwareBeanPostProcessor，会调用postProcessBeforeInstantiation()】
 *              1）、先从缓存中获取当前bean，如果能获取到，说明bean是之前被创建过的，直接使
用，否则再创建；
 *                  只要创建好的Bean都会被缓存起来
 *              2）、createBean（）;创建bean；
 *                  AnnotationAwareAspectJAutoProxyCreator 会在任何bean创建之前先尝
试返回bean的实例
 *                  【BeanPostProcessor是在Bean对象创建完成初始化前后调用的】
 *                  【InstantiationAwareBeanPostProcessor是在创建Bean实例之前先尝试
用后置处理器返回对象的】
 *                  1）、resolveBeforeInstantiation(beanName, mbdToUse);解析
BeforeInstantiation
 *                      希望后置处理器在此能返回一个代理对象；如果能返回代理对象就使用，如
果不能就继续
 *                      1）、后置处理器先尝试返回对象；
 *                          bean =
applyBeanPostProcessorsBeforeInstantiation（）；
 *                              拿到所有后置处理器，如果是
InstantiationAwareBeanPostProcessor;
 *                              就执行postProcessBeforeInstantiation
 *                          if (bean != null) {
 *                              bean =
applyBeanPostProcessorsAfterInitialization(bean, beanName);
                            }
 *
 *                  2）、doCreateBean(beanName, mbdToUse, args);真正的去创建一个
bean实例；和3.6流程一样；
 *                  3）、
 *
 *
```

```
 * AnnotationAwareAspectJAutoProxyCreator【InstantiationAwareBeanPostProcessor】的作用：
 * 1）、每一个bean创建之前，调用postProcessBeforeInstantiation()；
 *      关心MathCalculator和LogAspect的创建
 *      1）、判断当前bean是否在advisedBeans中（保存了所有需要增强bean）
 *      2）、判断当前bean是否是基础类型的Advice、Pointcut、Advisor、AopInfrastructureBean，
 *          或者是否是切面（@Aspect）
 *      3）、是否需要跳过
 *          1）、获取候选的增强器（切面里面的通知方法）【List<Advisor> candidateAdvisors】
 *              每一个封装的通知方法的增强器是 InstantiationModelAwarePointcutAdvisor；
 *              判断每一个增强器是否是 AspectJPointcutAdvisor 类型的；返回true
 *          2）、永远返回false
 *
 * 2）、创建对象
 * postProcessAfterInitialization；
 *      return wrapIfNecessary(bean, beanName, cacheKey);//包装如果需要的情况下
 *      1）、获取当前bean的所有增强器（通知方法）  Object[]  specificInterceptors
 *          1、找到候选的所有的增强器（找哪些通知方法是需要切入当前bean方法的）
 *          2、获取到能在bean使用的增强器。
 *          3、给增强器排序
 *      2）、保存当前bean在advisedBeans中；
 *      3）、如果当前bean需要增强，创建当前bean的代理对象；
 *          1）、获取所有增强器（通知方法）
 *          2）、保存到proxyFactory
 *          3）、创建代理对象：Spring自动决定
 *              JdkDynamicAopProxy(config);jdk动态代理；
 *              ObjenesisCglibAopProxy(config);cglib的动态代理；
 *      4）、给容器中返回当前组件使用cglib增强了的代理对象；
 *      5）、以后容器中获取到的就是这个组件的代理对象，执行目标方法的时候，代理对象就会执行通知方法的流程；
 *
 *
 *  3）、目标方法执行    ；
 *      容器中保存了组件的代理对象（cglib增强后的对象），这个对象里面保存了详细信息（比如增强器，目标对象，xxx）；
 *      1）、CglibAopProxy.intercept();拦截目标方法的执行
 *      2）、根据ProxyFactory对象获取将要执行的目标方法拦截器链；
 *          List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);
 *          1）、List<Object> interceptorList保存所有拦截器 5
 *              一个默认的ExposeInvocationInterceptor 和 4个增强器；
 *          2）、遍历所有的增强器，将其转为Interceptor；
 *              registry.getInterceptors(advisor);
 *          3）、将增强器转为List<MethodInterceptor>；
 *              如果是MethodInterceptor，直接加入到集合中
 *              如果不是，使用AdvisorAdapter将增强器转为MethodInterceptor；
 *              转换完成返回MethodInterceptor数组；
 *
 *      3）、如果没有拦截器链，直接执行目标方法；
 *          拦截器链（每一个通知方法又被包装为方法拦截器，利用MethodInterceptor机制）
 *      4）、如果有拦截器链，把需要执行的目标对象，目标方法，
 *          拦截器链等信息传入创建一个 CglibMethodInvocation 对象，
 *          并调用 Object retVal =  mi.proceed();
 *      5）、拦截器链的触发过程；
```

```
 *          1)、如果没有拦截器执行执行目标方法，或者拦截器的索引和拦截器数组-1大小一样（指定到了最后一个拦截器）执行目标方法；
 *          2)、链式获取每一个拦截器，拦截器执行invoke方法，每一个拦截器等待下一个拦截器执行完成返回以后再来执行；
 *              拦截器链的机制，保证通知方法与目标方法的执行顺序；
 *
 *  总结：
 *      1）、  @EnableAspectJAutoProxy 开启AOP功能
 *      2）、 @EnableAspectJAutoProxy 会给容器中注册一个组件 AnnotationAwareAspectJAutoProxyCreator
 *      3）、AnnotationAwareAspectJAutoProxyCreator是一个后置处理器；
 *      4）、容器的创建流程：
 *          1）、registerBeanPostProcessors（）注册后置处理器；创建AnnotationAwareAspectJAutoProxyCreator对象
 *          2）、finishBeanFactoryInitialization（）初始化剩下的单实例bean
 *              1）、创建业务逻辑组件和切面组件
 *              2）、AnnotationAwareAspectJAutoProxyCreator拦截组件的创建过程
 *              3）、组件创建完之后，判断组件是否需要增强
 *                  是：切面的通知方法，包装成增强器（Advisor）;给业务逻辑组件创建一个代理对象（cglib）；
 *      5）、执行目标方法：
 *          1）、代理对象执行目标方法
 *          2）、CglibAopProxy.intercept()；
 *              1）、得到目标方法的拦截器链（增强器包装成拦截器MethodInterceptor）
 *              2）、利用拦截器的链式机制，依次进入每一个拦截器进行执行；
 *              3）、效果：
 *                  正常执行：前置通知-》目标方法-》后置通知-》返回通知
 *                  出现异常：前置通知-》目标方法-》后置通知-》异常通知
 *
 *
 *
 */
@EnableAspectJAutoProxy
@Configuration
public class MainConfigOfAOP {

    //业务逻辑类加入容器中
    @Bean
    public MathCalculator calculator(){
        return new MathCalculator();
    }

    //切面类加入到容器中
    @Bean
    public LogAspects logAspects(){
        return new LogAspects();
    }
}
```

aop调用过程原理图



aop源码详解

@EnableAspectJAutoProxy注解

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Import(AspectJAutoProxyRegistrar.class)
public @interface EnableAspectJAutoProxy {

    /**
     * Indicate whether subclass-based (CGLIB) proxies are to be created as
opposed
     * to standard Java interface-based proxies. The default is {@code false}.
     */
    boolean proxyTargetClass() default false;

    /**
     * Indicate that the proxy should be exposed by the AOP framework as a
{@code ThreadLocal}
     * for retrieval via the {@link
org.springframework.aop.framework.AopContext} class.
     * Off by default, i.e. no guarantees that {@code AopContext} access will
work.
     * @since 4.3.1
     */
    boolean exposeProxy() default false;

}
```

AspectJAutoProxyRegistrar类导入了什么？

AspectJAutoProxyRegistrar 给容器中导入了
org.springframework.aop.config.internalAutoProxyCreator

```
class AspectJAutoProxyRegistrar implements ImportBeanDefinitionRegistrar {

    @Override
    public void registerBeanDefinitions(
            AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry
registry) {

        //向容器中导入internalAutoProxyCreator

AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);

        //获取@EnableAspectJAutoProxy的属性（proxyTargetClass、exposeProxy）
        AnnotationAttributes enableAspectJAutoProxy =
                AnnotationConfigUtils.attributesFor(importingClassMetadata,
EnableAspectJAutoProxy.class);
        if (enableAspectJAutoProxy.getBoolean("proxyTargetClass")) {
            AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
        }
        if (enableAspectJAutoProxy.getBoolean("exposeProxy")) {
            AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry···);
        }
    }

}
```

registerAspectJAnnotationAutoProxyCreatorIfNecessary方法详解

```
public abstract class AopConfigUtils {

    /**
     * The bean name of the internally managed auto-proxy creator.
     */
    public static final String AUTO_PROXY_CREATOR_BEAN_NAME =
            "org.springframework.aop.config.internalAutoProxyCreator";

public static BeanDefinition
registerAspectJAnnotationAutoProxyCreatorIfNecessary(BeanDefinitionRegistry
registry) {
        return registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry,
null);
    }

    public static BeanDefinition
registerAspectJAnnotationAutoProxyCreatorIfNecessary(BeanDefinitionRegistry
registry, Object source) {
        return
registerOrEscalateApcAsRequired(AnnotationAwareAspectJAutoProxyCreator.class,
registry, source);
    }

    //创建internalAutoProxyCreator(AnnotationAwareAspectJAutoProxyCreator)定义信息
    private static BeanDefinition registerOrEscalateApcAsRequired(Class<?>
clazz, BeanDefinitionRegistry registry, Object source) {
```

```
        Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
        if (registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME)) {
            BeanDefinition apcDefinition =
registry.getBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME);
            return null;
        }
        RootBeanDefinition beanDefinition = new RootBeanDefinition(clazz);
        beanDefinition.setSource(source);
        beanDefinition.getPropertyValues().add("order",
Ordered.HIGHEST_PRECEDENCE);
        beanDefinition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
        registry.registerBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME,
beanDefinition);
        return beanDefinition;
    }

}
```

AnnotationAwareAspectJAutoProxyCreator 做了什么事？

AnnotationAwareAspectJAutoProxyCreator继承树如下所示：

AnnotationAwareAspectJAutoProxyCreator

--->AspectJAwareAdvisorAutoProxyCreator

---->AbstractAdvisorAutoProxyCreator

----->AbstractAutoProxyCreator

------>implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware

我们需要关注后置处理器（在bean初始化完成前后做事情）、自动装配BeanFactory

我们先来分析顶层父类AbstractAutoProxyCreator

```
@SuppressWarnings("serial")
public abstract class AbstractAutoProxyCreator extends ProxyProcessorSupport
        implements SmartInstantiationAwareBeanPostProcessor, BeanFactoryAware {

    private final Set<String> targetSourcedBeans =
            Collections.newSetFromMap(new ConcurrentHashMap<String, Boolean>
(16));

    private final Set<Object> earlyProxyReferences =
            Collections.newSetFromMap(new ConcurrentHashMap<Object, Boolean>
(16));

    private final Map<Object, Class<?>> proxyTypes = new
ConcurrentHashMap<Object, Class<?>>(16);

    private final Map<Object, Boolean> advisedBeans = new
ConcurrentHashMap<Object, Boolean>(256);
```

```
    @Override
    public void setBeanFactory(BeanFactory beanFactory) {
        this.beanFactory = beanFactory;
    }

    /**
     * Return the owning {@link BeanFactory}.
     * May be {@code null}, as this post-processor doesn't need to belong to a
bean factory.
     */
    protected BeanFactory getBeanFactory() {
        return this.beanFactory;
    }


    @Override
    public Class<?> predictBeanType(Class<?> beanClass, String beanName) {
        if (this.proxyTypes.isEmpty()) {
            return null;
        }
        Object cacheKey = getCacheKey(beanClass, beanName);
        return this.proxyTypes.get(cacheKey);
    }

    @Override
    public Constructor<?>[] determineCandidateConstructors(Class<?> beanClass,
String beanName) throws BeansException {
        return null;
    }



    @Override
    public Object postProcessBeforeInstantiation(Class<?> beanClass, String
beanName) throws BeansException {
        Object cacheKey = getCacheKey(beanClass, beanName);

        if (beanName == null || !this.targetSourcedBeans.contains(beanName)) {
            if (this.advisedBeans.containsKey(cacheKey)) {
                return null;
            }
            if (isInfrastructureClass(beanClass) || shouldSkip(beanClass,
beanName)) {
                this.advisedBeans.put(cacheKey, Boolean.FALSE);
                return null;
            }
        }

        // Create proxy here if we have a custom TargetSource.
        // Suppresses unnecessary default instantiation of the target bean:
        // The TargetSource will handle target instances in a custom fashion.
        if (beanName != null) {
            TargetSource targetSource = getCustomTargetSource(beanClass,
beanName);
            if (targetSource != null) {
                this.targetSourcedBeans.add(beanName);
                Object[] specificInterceptors =
getAdvicesAndAdvisorsForBean(beanClass, beanName, targetSource);
```

```
                Object proxy = createProxy(beanClass, beanName,
specificInterceptors, targetSource);
                this.proxyTypes.put(cacheKey, proxy.getClass());
                return proxy;
            }
        }

        return null;
    }

    @Override
    public boolean postProcessAfterInstantiation(Object bean, String beanName) {
        return true;
    }

    @Override
    public PropertyValues postProcessPropertyValues(
            PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String
beanName) {

        return pvs;
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
{
        return bean;
    }

    /**
     * Create a proxy with the configured interceptors if the bean is
     * identified as one to proxy by the subclass.
     * @see #getAdvicesAndAdvisorsForBean
     */
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
        if (bean != null) {
            Object cacheKey = getCacheKey(bean.getClass(), beanName);
            if (!this.earlyProxyReferences.contains(cacheKey)) {
                return wrapIfNecessary(bean, beanName, cacheKey);
            }
        }
        return bean;
    }


    /**
     * Build a cache key for the given bean class and bean name.
     * <p>Note: As of 4.2.3, this implementation does not return a concatenated
     * class/name String anymore but rather the most efficient cache key
possible:
     * a plain bean name, prepended with {@link BeanFactory#FACTORY_BEAN_PREFIX}
     * in case of a {@code FactoryBean}; or if no bean name specified, then the
     * given bean {@code Class} as-is.
     * @param beanClass the bean class
     * @param beanName the bean name
     * @return the cache key for the given class and name
```

```
     */
    //BeanFactory.FACTORY_BEAN_PREFIX=&
    protected Object getCacheKey(Class<?> beanClass, String beanName) {
        if (StringUtils.hasLength(beanName)) {
            return (FactoryBean.class.isAssignableFrom(beanClass) ?
                    BeanFactory.FACTORY_BEAN_PREFIX + beanName : beanName);
        }
        else {
            return beanClass;
        }
    }

    /**
     * Wrap the given bean if necessary, i.e. if it is eligible for being
proxied.
     * @param bean the raw bean instance
     * @param beanName the name of the bean
     * @param cacheKey the cache key for metadata access
     * @return a proxy wrapping the bean, or the raw bean instance as-is
     */
    //重点
    protected Object wrapIfNecessary(Object bean, String beanName, Object
cacheKey) {
        if (beanName != null && this.targetSourcedBeans.contains(beanName)) {
            return bean;
        }
        if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
            return bean;
        }
        if (isInfrastructureClass(bean.getClass()) ||
shouldSkip(bean.getClass(), beanName)) {
            this.advisedBeans.put(cacheKey, Boolean.FALSE);
            return bean;
        }

        // Create proxy if we have advice.
        Object[] specificInterceptors =
getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, null);
        if (specificInterceptors != DO_NOT_PROXY) {
            this.advisedBeans.put(cacheKey, Boolean.TRUE);
            Object proxy = createProxy(
                    bean.getClass(), beanName, specificInterceptors, new
SingletonTargetSource(bean));
            this.proxyTypes.put(cacheKey, proxy.getClass());
            return proxy;
        }

        this.advisedBeans.put(cacheKey, Boolean.FALSE);
        return bean;
    }

    /**
      TODO
     */
    protected boolean isInfrastructureClass(Class<?> beanClass) {
        boolean retVal = Advice.class.isAssignableFrom(beanClass) ||
                Pointcut.class.isAssignableFrom(beanClass) ||
                Advisor.class.isAssignableFrom(beanClass) ||
```

```
                AopInfrastructureBean.class.isAssignableFrom(beanClass);
        if (retVal && logger.isTraceEnabled()) {
            logger.trace("Did not attempt to auto-proxy infrastructure class ["
+ beanClass.getName() + "]");
        }
        return retVal;
    }


    protected boolean shouldSkip(Class<?> beanClass, String beanName) {
        return false;
    }



    /**
     * Create an AOP proxy for the given bean.
     * @param beanClass the class of the bean
     * @param beanName the name of the bean
     * @param specificInterceptors the set of interceptors that is
     * specific to this bean (may be empty, but not null)
     * @param targetSource the TargetSource for the proxy,
     * already pre-configured to access the bean
     * @return the AOP proxy for the bean
     * @see #buildAdvisors
     */
    protected Object createProxy(
            Class<?> beanClass, String beanName, Object[] specificInterceptors,
TargetSource targetSource) {

        if (this.beanFactory instanceof ConfigurableListableBeanFactory) {
            AutoProxyUtils.exposeTargetClass((ConfigurableListableBeanFactory)
this.beanFactory, beanName, beanClass);
        }

        ProxyFactory proxyFactory = new ProxyFactory();
        proxyFactory.copyFrom(this);

        if (!proxyFactory.isProxyTargetClass()) {
            if (shouldProxyTargetClass(beanClass, beanName)) {
                proxyFactory.setProxyTargetClass(true);
            }
            else {
                evaluateProxyInterfaces(beanClass, proxyFactory);
            }
        }

        Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
        proxyFactory.addAdvisors(advisors);
        proxyFactory.setTargetSource(targetSource);
        customizeProxyFactory(proxyFactory);

        proxyFactory.setFrozen(this.freezeProxy);
        if (advisorsPreFiltered()) {
            proxyFactory.setPreFiltered(true);
        }

        return proxyFactory.getProxy(getProxyClassLoader());
```

```
    }

    /**
     * Determine whether the given bean should be proxied with its target class
rather than its interfaces.
     * <p>Checks the {@link AutoProxyUtils#PRESERVE_TARGET_CLASS_ATTRIBUTE
"preserveTargetClass" attribute}
     * of the corresponding bean definition.
     * @param beanClass the class of the bean
     * @param beanName the name of the bean
     * @return whether the given bean should be proxied with its target class
     * @see AutoProxyUtils#shouldProxyTargetClass
     */
    protected boolean shouldProxyTargetClass(Class<?> beanClass, String
beanName) {
        return (this.beanFactory instanceof ConfigurableListableBeanFactory &&

AutoProxyUtils.shouldProxyTargetClass((ConfigurableListableBeanFactory)
this.beanFactory, beanName));
    }




}
```

Bean 后置处理器是如何实例化的？

1.创建IOC容器

```
public class AnnotationConfigApplicationContext extends
GenericApplicationContext implements AnnotationConfigRegistry {
public AnnotationConfigApplicationContext(Class<?>... annotatedClasses) {
        this();
    //注册配置类
        register(annotatedClasses);
    //刷新容器
        refresh();
    }

    //我们先关注registerBeanPostProcessors，refresh的其他方法我们在IOC容器的初始化中详细
给出
    refresh(){
    // Register bean processors that intercept bean creation. --> 注册bean后置处理
器
    registerBeanPostProcessors(beanFactory);
        }
}
```

2.registerBeanPostProcessors 向容器中注册Bean 后置处理器

```
/**
     * Instantiate and invoke all registered BeanPostProcessor beans,
     * respecting explicit order if given.
     * <p>Must be called before any instantiation of application beans.
     */
    protected void registerBeanPostProcessors(ConfigurableListableBeanFactory
beanFactory) {

PostProcessorRegistrationDelegate.registerBeanPostProcessors(beanFactory, this);
    }
```

```
public static void registerBeanPostProcessors(
            ConfigurableListableBeanFactory beanFactory,
AbstractApplicationContext applicationContext) {

    // 按照类型拿到IOC容器中所有需要创建的后置处理器
        String[] postProcessorNames =
beanFactory.getBeanNamesForType(BeanPostProcessor.class, true, false);

        // Register BeanPostProcessorChecker that logs an info message when
        // a bean is created during BeanPostProcessor instantiation, i.e. when
        // a bean is not eligible for getting processed by all
BeanPostProcessors.
        int beanProcessorTargetCount = beanFactory.getBeanPostProcessorCount() +
1 + postProcessorNames.length;
        //给容器中添加其他后置处理器
        beanFactory.addBeanPostProcessor(new
BeanPostProcessorChecker(beanFactory, beanProcessorTargetCount));

        // Separate between BeanPostProcessors that implement PriorityOrdered,
        // Ordered, and the rest.
        //给容器中的后置处理器排序
        //优先注册实现了PriorityOrdered接口的BeanPostProcessor;
        //再给容器中注册实现了Ordered接口的BeanPostProcessor;
        //注册没实现优先级接口的BeanPostProcessor;
        List<BeanPostProcessor> priorityOrderedPostProcessors = new
ArrayList<BeanPostProcessor>();

        List<BeanPostProcessor> internalPostProcessors = new
ArrayList<BeanPostProcessor>();

        List<String> orderedPostProcessorNames = new ArrayList<String>();

        List<String> nonOrderedPostProcessorNames = new ArrayList<String>();

        for (String ppName : postProcessorNames) {
            if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
                BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
                priorityOrderedPostProcessors.add(pp);
                if (pp instanceof MergedBeanDefinitionPostProcessor) {
                    internalPostProcessors.add(pp);
```

```
                }
            }
            else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
                orderedPostProcessorNames.add(ppName);
            }
            else {
                nonOrderedPostProcessorNames.add(ppName);
            }
        }

    /**
    *按照优先级分别注册BeanPostProcessors
    */
        // First, register the BeanPostProcessors that implement
PriorityOrdered.
        sortPostProcessors(priorityOrderedPostProcessors, beanFactory);
        registerBeanPostProcessors(beanFactory, priorityOrderedPostProcessors);

        // Next, register the BeanPostProcessors that implement Ordered.
        List<BeanPostProcessor> orderedPostProcessors = new
ArrayList<BeanPostProcessor>();
        for (String ppName : orderedPostProcessorNames) {
            BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
            orderedPostProcessors.add(pp);
            if (pp instanceof MergedBeanDefinitionPostProcessor) {
                internalPostProcessors.add(pp);
            }
        }
        sortPostProcessors(orderedPostProcessors, beanFactory);
        registerBeanPostProcessors(beanFactory, orderedPostProcessors);

        // Now, register all regular BeanPostProcessors.
        List<BeanPostProcessor> nonOrderedPostProcessors = new
ArrayList<BeanPostProcessor>();
        for (String ppName : nonOrderedPostProcessorNames) {
            BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
            nonOrderedPostProcessors.add(pp);
            if (pp instanceof MergedBeanDefinitionPostProcessor) {
                internalPostProcessors.add(pp);
            }
        }
        registerBeanPostProcessors(beanFactory, nonOrderedPostProcessors);

        // Finally, re-register all internal BeanPostProcessors.
        sortPostProcessors(internalPostProcessors, beanFactory);
        registerBeanPostProcessors(beanFactory, internalPostProcessors);

        // Re-register post-processor for detecting inner beans as
ApplicationListeners,
        // moving it to the end of the processor chain (for picking up proxies
etc).
        beanFactory.addBeanPostProcessor(new
ApplicationListenerDetector(applicationContext));
    }
```

3.BeanPostProcessor实例如何被创建？

AbstractBeanFactory中：

BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class)

```
@Override
    public <T> T getBean(String name, Class<T> requiredType) throws
BeansException {
        return doGetBean(name, requiredType, null, false);
    }
```

```
protected <T> T doGetBean(
            final String name, final Class<T> requiredType, final Object[] args,
boolean typeCheckOnly)
            throws BeansException {

        final String beanName = transformedBeanName(name);
        Object bean;

            // Check if bean definition exists in this factory.
            BeanFactory parentBeanFactory = getParentBeanFactory();

            try {
                final RootBeanDefinition mbd =
getMergedLocalBeanDefinition(beanName);

                // Guarantee initialization of beans that the current bean
depends on.
                String[] dependsOn = mbd.getDependsOn();
                if (dependsOn != null) {
                    for (String dep : dependsOn) {
                        if (isDependent(beanName, dep)) {
                            throw new
BeanCreationException(mbd.getResourceDescription(), beanName,
                                    "Circular depends-on relationship between '"
+ beanName + "' and '" + dep + "'");
                        }
                        registerDependentBean(dep, beanName);
                        getBean(dep);
                    }
                }

                // Create bean instance.
                if (mbd.isSingleton()) {
                //单实例bean从缓存中获取，获取不到则创建bean-->new
ObjectFactory【DefaultListableBeanFactory】
                    sharedInstance = getSingleton(beanName, new
ObjectFactory<Object>() {
                        @Override
                        public Object getObject() throws BeansException {
                                return createBean(beanName, mbd, args);
                    });
                    //如果它是FactoryBean，我们将使用它创建一个bean实例
```

```
                    bean = getObjectForBeanInstance(sharedInstance, name,
beanName, mbd);
                }
                //isPrototype 创建新的bean实例
                else if (mbd.isPrototype()) {
                    // It's a prototype -> create a new instance.
                    Object prototypeInstance = null;

                prototypeInstance = createBean(beanName, mbd, args);

                    bean = getObjectForBeanInstance(prototypeInstance, name,
beanName, mbd);
                }

                else {
                    String scopeName = mbd.getScope();
                    final Scope scope = this.scopes.get(scopeName);
                    if (scope == null) {
                        throw new IllegalStateException("No Scope registered for
scope name '" + scopeName + "'");
                    }

                        Object scopedInstance = scope.get(beanName, new
ObjectFactory<Object>() {
                            @Override
                            public Object getObject() throws BeansException {
                                beforePrototypeCreation(beanName);
                                try {
                                    return createBean(beanName, mbd, args);
                                }
                                finally {
                                    afterPrototypeCreation(beanName);
                                }
                            }
                        });
                        bean = getObjectForBeanInstance(scopedInstance, name,
beanName, mbd);

}

        return (T) bean;
}
}



//返回合并的RootBeanDefinition，如果指定的bean对应于子bean定义，则遍历父bean定义
    protected RootBeanDefinition getMergedLocalBeanDefinition(String beanName)
throws BeansException {
        // Quick check on the concurrent map first, with minimal locking.
        RootBeanDefinition mbd = this.mergedBeanDefinitions.get(beanName);
        if (mbd != null) {
            return mbd;
        }
        //如果给定bean的定义是子bean定义，则通过与父级合并返回RootBeanDefinition。
        return getMergedBeanDefinition(beanName, getBeanDefinition(beanName));
    }
```

```
protected Object getObjectForBeanInstance(
            Object beanInstance, String name, String beanName,
RootBeanDefinition mbd) {


        //现在有了bean实例，它可以是普通bean或FactoryBean。
        //如果它是FactoryBean，我们将使用它创建一个bean实例，除非
        //呼叫者实际上想要引用工厂。
        if (!(beanInstance instanceof FactoryBean) ||
BeanFactoryUtils.isFactoryDereference(name)) {
            return beanInstance;
        }

        Object object = null;
        if (mbd == null) {
            object = getCachedObjectForFactoryBean(beanName);
        }
        if (object == null) {
            // Return bean instance from factory.
            FactoryBean<?> factory = (FactoryBean<?>) beanInstance;

            boolean synthetic = (mbd != null && mbd.isSynthetic());
            object = getObjectFromFactoryBean(factory, beanName, !synthetic);
        }
        return object;
    }


    /**
    是否要返回工厂bean，BeanFactory.FACTORY_BEAN_PREFI=&
     */
    public static boolean isFactoryDereference(String name) {
        return (name != null &&
name.startsWith(BeanFactory.FACTORY_BEAN_PREFIX));
    }


//从工厂bean获取bean实例
protected Object getObjectFromFactoryBean(FactoryBean<?> factory, String
beanName, boolean shouldPostProcess) {
        if (factory.isSingleton() && containsSingleton(beanName)) {
            synchronized (getSingletonMutex()) {
            //判断容器中是否有该bean
                Object object = this.factoryBeanObjectCache.get(beanName);
                //没有则利用工厂bean创建实例
                if (object == null) {
                    object = doGetObjectFromFactoryBean(factory, beanName);

                    this.factoryBeanObjectCache.put(beanName, (object != null ?
object : NULL_OBJECT));
                    }
                }
                return (object != NULL_OBJECT ? object : null);
    }
}
```

```
protected Object getCachedObjectForFactoryBean(String beanName) {
        Object object = this.factoryBeanObjectCache.get(beanName);
        return (object != NULL_OBJECT ? object : null);
    }



//Obtain an object to expose from the given FactoryBean.
    private Object doGetObjectFromFactoryBean(final FactoryBean<?> factory,
final String beanName)
            throws BeanCreationException {

    Object object;

    object = factory.getObject();

    return object;
    }
```

4.getSingleton 创建Or获取单例bean

AbstractBeanFactory中：

sharedInstance = getSingleton(beanName, new ObjectFactory()

```
public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
        Assert.notNull(beanName, "'beanName' must not be null");
        synchronized (this.singletonObjects) {
            //singletonObjects ---> ConcurrentHashMap 单实例bean缓存
            Object singletonObject = this.singletonObjects.get(beanName);
            if (singletonObject == null) {
                boolean newSingleton = false;

                try {
                    //获取不到则创建bean
,singletonFactory【DefaultListableBeanFactory】.getObject()会调用
AbstractAutowireCapableBeanFactory.createBean()
                    singletonObject = singletonFactory.getObject();
                    newSingleton = true;
                }
                catch (IllegalStateException ex) {

                }
                catch (BeanCreationException ex) {

                }
                finally {

                    afterSingletonCreation(beanName);
                }
                if (newSingleton) {

                    addSingleton(beanName, singletonObject);
                }
            }
```

```
            return (singletonObject != NULL_OBJECT ? singletonObject : null);
        }
    }

//Add the given singleton object to the singleton cache of this factory.
protected void addSingleton(String beanName, Object singletonObject) {
        synchronized (this.singletonObjects) {
            this.singletonObjects.put(beanName, (singletonObject != null ?
singletonObject : NULL_OBJECT));
            this.singletonFactories.remove(beanName);
            this.earlySingletonObjects.remove(beanName);
            this.registeredSingletons.add(beanName);
        }
    }
```

singletonObject = singletonFactory.getObject();

singletonFactory是谁？org.springframework.beans.factory.support.DefaultListableBeanFactory

在其(DefaultListableBeanFactory)子类AbstractAutowireCapableBeanFactory中：

```
public abstract class AbstractAutowireCapableBeanFactory extends
AbstractBeanFactory{
public <T> T createBean(Class<T> beanClass) throws BeansException {
        // Use prototype bean definition, to avoid registering bean as dependent
bean.
        RootBeanDefinition bd = new RootBeanDefinition(beanClass);
        bd.setScope(SCOPE_PROTOTYPE);
        bd.allowCaching = ClassUtils.isCacheSafe(beanClass,
getBeanClassLoader());
        return (T) createBean(beanClass.getName(), bd, null);
    }

    protected Object createBean(String beanName, RootBeanDefinition mbd,
Object[] args) throws BeanCreationException {

        RootBeanDefinition mbdToUse = mbd;

        // Make sure bean class is actually resolved at this point, and
        // clone the bean definition in case of a dynamically resolved Class
        // which cannot be stored in the shared merged bean definition.
        Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
        if (resolvedClass != null && !mbd.hasBeanClass() &&
mbd.getBeanClassName() != null) {
            mbdToUse = new RootBeanDefinition(mbd);
            mbdToUse.setBeanClass(resolvedClass);
        }

        try {
            // Give BeanPostProcessors a chance to return a proxy instead of the
target bean instance.
            Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
            if (bean != null) {
                return bean;
            }
        }
        catch (Throwable ex) {
```

```
            throw new BeanCreationException(mbdToUse.getResourceDescription(),
beanName,
                    "BeanPostProcessor before instantiation of bean failed",
ex);
        }

        Object beanInstance = doCreateBean(beanName, mbdToUse, args);

        return beanInstance;
    }
}


protected Class<?> resolveBeanClass(final RootBeanDefinition mbd, String
beanName, final Class<?>... typesToMatch)
            throws CannotLoadBeanClassException {

                return mbd.getBeanClass();
    }
```

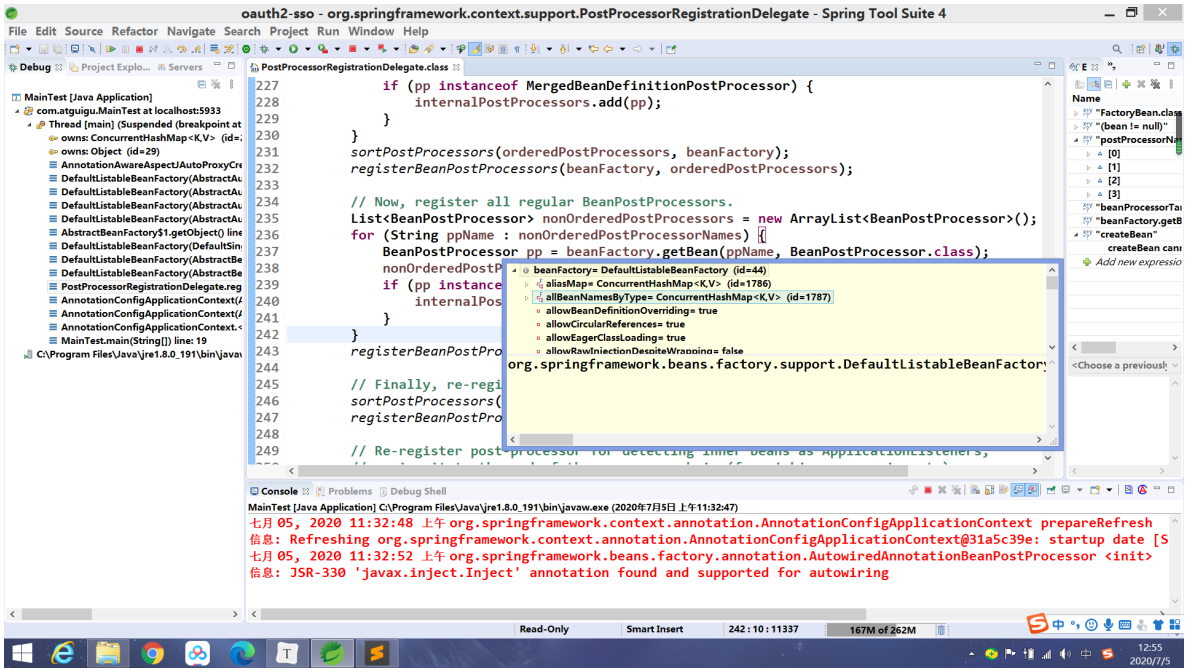
```
protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition
mbd) {
        Object bean = null;
        if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
            // Make sure bean class is actually resolved at this point.
            if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors())
{
                Class<?> targetType = determineTargetType(beanName, mbd);
                if (targetType != null) {
                    bean =
applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
                    if (bean != null) {
                        bean = applyBeanPostProcessorsAfterInitialization(bean,
beanName);
                    }
                }
            }
            mbd.beforeInstantiationResolved = (bean != null);
        }
        return bean;
    }
```

```
    protected Object applyBeanPostProcessorsBeforeInstantiation(Class<?>
beanClass, String beanName) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBeanPostProcessor) bp;
                Object result = ibp.postProcessBeforeInstantiation(beanClass,
beanName);
                if (result != null) {
                    return result;
                }
            }
        }
        return null;
    }
```

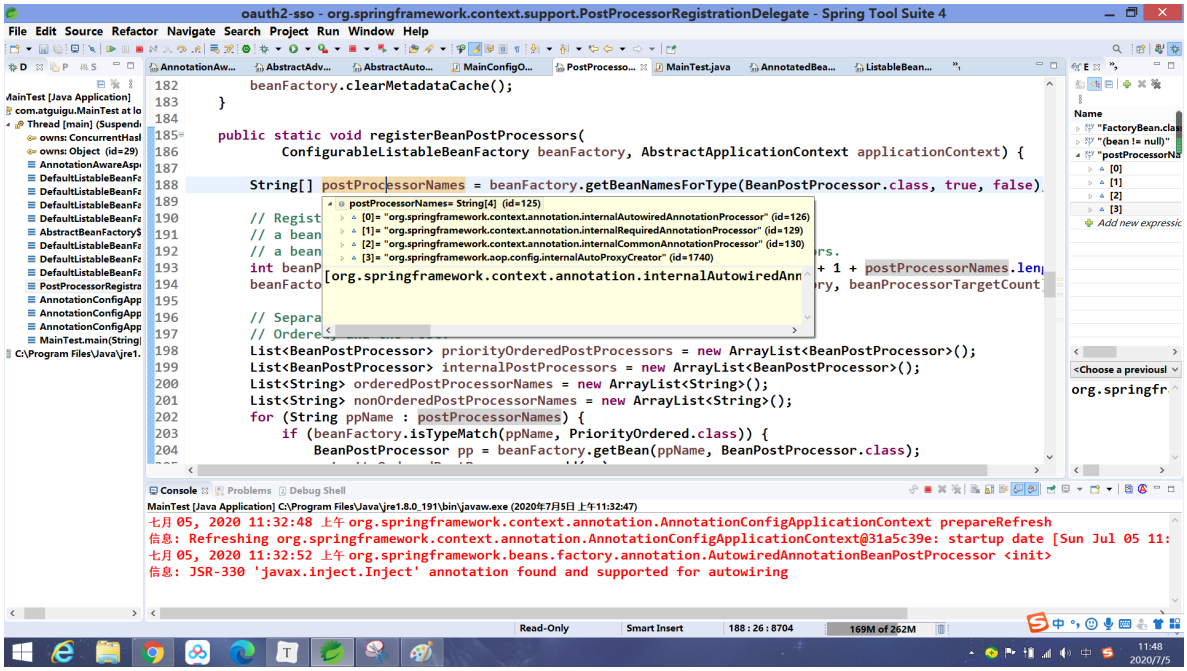
```
public Object applyBeanPostProcessorsAfterInitialization(Object existingBean,
String beanName)
            throws BeansException {

        Object result = existingBean;
        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
            result = beanProcessor.postProcessAfterInitialization(result,
beanName);
            if (result == null) {
                return result;
            }
        }
        return result;
    }
```

5.重点debug图像

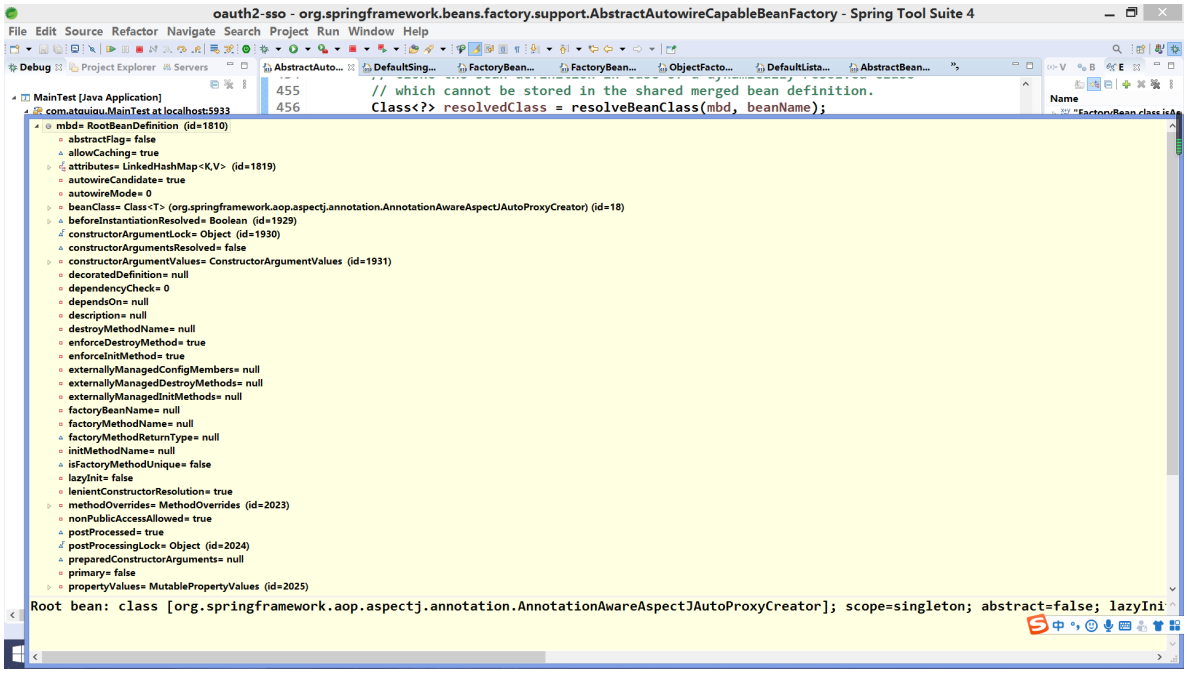
beanFactory为何物

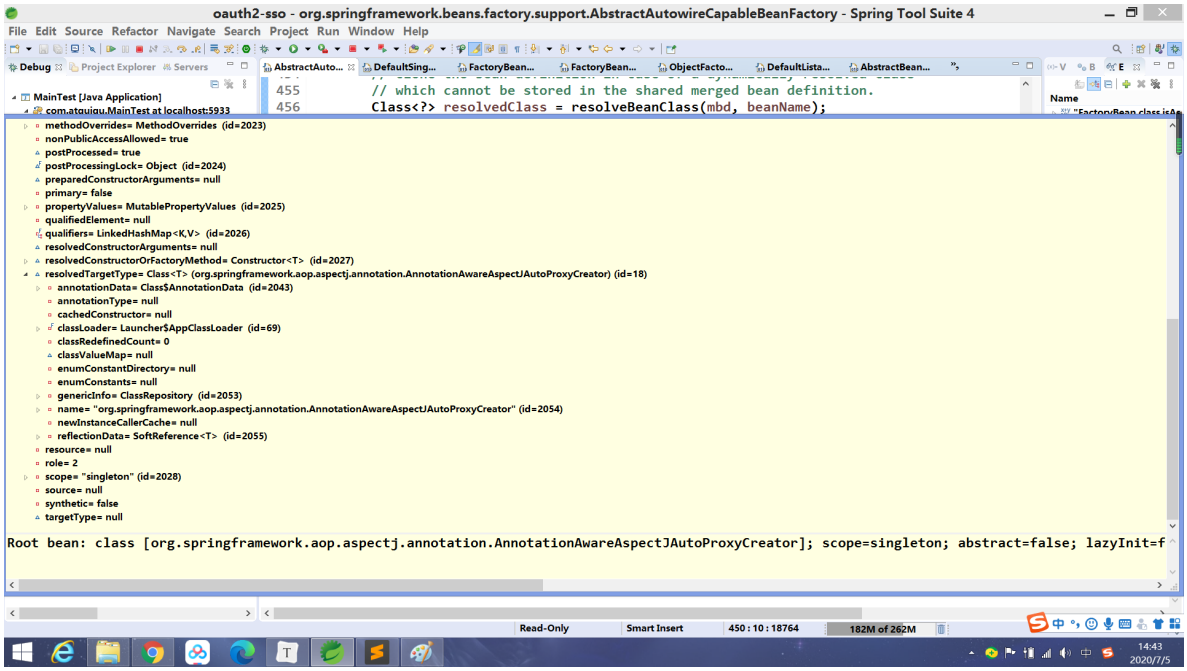


容器中的后置处理器



beanDedinition为何物





Spring 事务机制

Spring IOC容器初始化过程