## 引言

最近在读Spring源码，但是Spring代码层次嵌套太多，读起来有很大跳跃性，我有个朋友甚至开玩笑说，读Spring得拿纸笔，把方法和层次都写下来。

其实Spring我已经接触很久了，记得大学有个老师说过：“学一门技术，最好是先思考一下，如果是你，会怎么实现，再带着问题去学习它”。也有人把程序员与画家做比较，画家有门基本功叫临摹，我想程序员是不是也可以用这样的方式，学习一下世界顶级的项目的编程方法？

于是就有了[tiny-spring](https://github.com/code4craft/tiny-spring)。这个项目是从我的使用场景出发，理解Spring的功能，并且一步一步完善出来的。类和方法命名基本都是照搬Spring的，包括一些配置格式都相同。这个项目我会控制在1000行以内，但是会尽量覆盖Spring的IoC和AOP核心功能。

tiny-spring是逐步进行构建的，里程碑版本我都使用了git tag来管理。例如，最开始的tag是step-1-container-register-and-get，那么可以使用

git checkout step-1-container-register-and-get

来获得这一版本。

这次主要是学习IoC部分，以下是各版本的记录：

## 1.step1-最基本的容器

git checkout step-1-container-register-and-get

IoC最基本的角色有两个：容器(BeanFactory)和Bean本身。这里使用BeanDefinition来封装了bean对象，这样可以保存一些额外的元信息。测试代码：

// 1.初始化beanfactory

BeanFactory beanFactory = new BeanFactory();

// 2.注入bean

BeanDefinition beanDefinition = new BeanDefinition(new HelloWorldService());

beanFactory.registerBeanDefinition("helloWorldService", beanDefinition);

// 3.获取bean

HelloWorldService helloWorldService = (HelloWorldService) beanFactory.getBean("helloWorldService");

helloWorldService.helloWorld();

## 2.step2-将bean创建放入工厂

git checkout step-2-abstract-beanfactory-and-do-bean-initilizing-in-it

step1中的bean是初始化好之后再set进去的，实际使用中，我们希望容器来管理bean的创建。于是我们将bean的初始化放入BeanFactory中。为了保证扩展性，我们使用Extract Interface的方法，将BeanFactory替换成接口，而使用AbstractBeanFactory和AutowireCapableBeanFactory作为其实现。"AutowireCapable"的意思是“可自动装配的”，为我们后面注入属性做准备。

// 1.初始化beanfactory

BeanFactory beanFactory = new AutowireCapableBeanFactory();

// 2.注入bean

BeanDefinition beanDefinition = new BeanDefinition();

beanDefinition.setBeanClassName("us.codecraft.tinyioc.HelloWorldService");

beanFactory.registerBeanDefinition("helloWorldService", beanDefinition);

// 3.获取bean

HelloWorldService helloWorldService = (HelloWorldService) beanFactory.getBean("helloWorldService");

helloWorldService.helloWorld();

## 3.step3-为bean注入属性

git checkout step-3-inject-bean-with-property

这一步，我们想要为bean注入属性。我们选择将属性注入信息保存成PropertyValue对象，并且保存到BeanDefinition中。这样在初始化bean的时候，我们就可以根据PropertyValue来进行bean属性的注入。Spring本身使用了setter来进行注入，这里为了代码简洁，我们使用Field的形式来注入。

// 1.初始化beanfactory

BeanFactory beanFactory = new AutowireCapableBeanFactory();

// 2.bean定义

BeanDefinition beanDefinition = new BeanDefinition();

beanDefinition.setBeanClassName("us.codecraft.tinyioc.HelloWorldService");

// 3.设置属性

PropertyValues propertyValues = new PropertyValues();

propertyValues.addPropertyValue(new PropertyValue("text", "Hello World!"));

beanDefinition.setPropertyValues(propertyValues);

// 4.生成bean

beanFactory.registerBeanDefinition("helloWorldService", beanDefinition);

// 5.获取bean

HelloWorldService helloWorldService = (HelloWorldService) beanFactory.getBean("helloWorldService");

helloWorldService.helloWorld();

## 4.step4-读取xml配置来初始化bean

git checkout step-4-config-beanfactory-with-xml

这么大一坨初始化代码让人心烦。这里的BeanDefinition只是一些配置，我们还是用xml来初始化吧。我们定义了BeanDefinitionReader初始化bean，它有一个实现是XmlBeanDefinitionReader。

// 1.读取配置

XmlBeanDefinitionReader xmlBeanDefinitionReader = new XmlBeanDefinitionReader(new ResourceLoader());

xmlBeanDefinitionReader.loadBeanDefinitions("tinyioc.xml");

// 2.初始化BeanFactory并注册bean

BeanFactory beanFactory = new AutowireCapableBeanFactory();

for (Map.Entry<String, BeanDefinition> beanDefinitionEntry : xmlBeanDefinitionReader.getRegistry().entrySet()) {

beanFactory.registerBeanDefinition(beanDefinitionEntry.getKey(), beanDefinitionEntry.getValue());

}

// 3.获取bean

HelloWorldService helloWorldService = (HelloWorldService) beanFactory.getBean("helloWorldService");

helloWorldService.helloWorld();

## 5.step5-为bean注入bean

git checkout step-5-inject-bean-to-bean

使用xml配置之后，似乎里我们熟知的Spring更近了一步！但是现在有一个大问题没有解决：我们无法处理bean之间的依赖，无法将bean注入到bean中，所以它无法称之为完整的IoC容器！如何实现呢？我们定义一个BeanReference，来表示这个属性是对另一个bean的引用。这个在读取xml的时候初始化，并在初始化bean的时候，进行解析和真实bean的注入。

for (PropertyValue propertyValue : mbd.getPropertyValues().getPropertyValues()) {

Field declaredField = bean.getClass().getDeclaredField(propertyValue.getName());

declaredField.setAccessible(true);

Object value = propertyValue.getValue();

if (value instanceof BeanReference) {

BeanReference beanReference = (BeanReference) value;

value = getBean(beanReference.getName());

}

declaredField.set(bean, value);

}

同时为了解决循环依赖的问题，我们使用lazy-init的方式，将createBean的事情放到getBean的时候才执行，是不是一下子方便很多？这样在注入bean的时候，如果该属性对应的bean找不到，那么就先创建！因为总是先创建后注入，所以不会存在两个循环依赖的bean创建死锁的问题。

// 1.读取配置

XmlBeanDefinitionReader xmlBeanDefinitionReader = new XmlBeanDefinitionReader(new ResourceLoader());

xmlBeanDefinitionReader.loadBeanDefinitions("tinyioc.xml");

// 2.初始化BeanFactory并注册bean

AbstractBeanFactory beanFactory = new AutowireCapableBeanFactory();

for (Map.Entry<String, BeanDefinition> beanDefinitionEntry : xmlBeanDefinitionReader.getRegistry().entrySet()) {

beanFactory.registerBeanDefinition(beanDefinitionEntry.getKey(), beanDefinitionEntry.getValue());

}

// 3.初始化bean

beanFactory.preInstantiateSingletons();

// 4.获取bean

HelloWorldService helloWorldService = (HelloWorldService) beanFactory.getBean("helloWorldService");

helloWorldService.helloWorld();

## 6.step6-ApplicationContext登场

git checkout step-6-invite-application-context

现在BeanFactory的功能齐全了，但是使用起来有点麻烦。于是我们引入熟悉的ApplicationContext接口，并在AbstractApplicationContext的refresh()方法中进行bean的初始化工作。

ApplicationContext applicationContext = new ClassPathXmlApplicationContext("tinyioc.xml");

HelloWorldService helloWorldService = (HelloWorldService) applicationContext.getBean("helloWorldService");

helloWorldService.helloWorld();

是不是非常熟悉？至此为止，我们的tiny-spring的IoC部分可说完工了。这部分的类、方法命名和作用，都是对应Spring中相应的组件。虽然代码量只有400多行，但是已经有了基本的IoC功能！

## 项目地址

最后补充一下项目地址：<https://github.com/code4craft/tiny-spring>

## 关于AOP

AOP是Spring核心功能之一。今天就用tiny-spring来实现一个AOP。具体功能会包括：

1. 读取AspectJ格式的Pointcut描述。
2. 使用JDK动态代理以及CGLib两种方式进行AOP织入。

AOP分为配置(Pointcut，Advice)，织入(Weave)两部分工作，当然还有一部分是将AOP整合到整个容器的生命周期中。

AOP相关概念较多，我不会一一列举，但是会在每一步对概念做一点解释。

## 7.step7-使用JDK动态代理实现AOP织入

git checkout step-7-method-interceptor-by-jdk-dynamic-proxy

织入（weave）相对简单，我们先从它开始。Spring AOP的织入点是AopProxy，它包含一个方法Object getProxy()来获取代理后的对象。

在Spring AOP中，我觉得最重要的两个角色，就是我们熟悉的MethodInterceptor和MethodInvocation（这两个角色都是AOP联盟的标准），它们分别对应AOP中两个基本角色：Advice和Joinpoint。Advice定义了在切点指定的逻辑，而Joinpoint则代表切点。

public interface MethodInterceptor extends Interceptor {

Object invoke(MethodInvocation invocation) throws Throwable;

}

Spring的AOP只支持方法级别的调用，所以其实在AopProxy里，我们只需要将MethodInterceptor放入对象的方法调用即可。

我们称被代理对象为TargetSource，而AdvisedSupport就是保存TargetSource和MethodInterceptor的元数据对象。这一步我们先实现一个基于JDK动态代理的JdkDynamicAopProxy，它可以对接口进行代理。于是我们就有了基本的织入功能。

@Test

public void testInterceptor() throws Exception {

// --------- helloWorldService without AOP

ApplicationContext applicationContext = new ClassPathXmlApplicationContext("tinyioc.xml");

HelloWorldService helloWorldService = (HelloWorldService) applicationContext.getBean("helloWorldService");

helloWorldService.helloWorld();

// --------- helloWorldService with AOP

// 1. 设置被代理对象(Joinpoint)

AdvisedSupport advisedSupport = new AdvisedSupport();

TargetSource targetSource = new TargetSource(helloWorldService, HelloWorldServiceImpl.class,

HelloWorldService.class);

advisedSupport.setTargetSource(targetSource);

// 2. 设置拦截器(Advice)

TimerInterceptor timerInterceptor = new TimerInterceptor();

advisedSupport.setMethodInterceptor(timerInterceptor);

// 3. 创建代理(Proxy)

JdkDynamicAopProxy jdkDynamicAopProxy = new JdkDynamicAopProxy(advisedSupport);

HelloWorldService helloWorldServiceProxy = (HelloWorldService) jdkDynamicAopProxy.getProxy();

// 4. 基于AOP的调用

helloWorldServiceProxy.helloWorld();

}

## 8.step8-使用AspectJ管理切面

git checkout step-8-invite-pointcut-and-aspectj

完成了织入之后，我们要考虑另外一个问题：对什么类以及什么方法进行AOP？对于“在哪切”这一问题的定义，我们又叫做“Pointcut”。Spring中关于Pointcut包含两个角色：ClassFilter和MethodMatcher，分别是对类和方法做匹配。Pointcut有很多种定义方法，例如类名匹配、正则匹配等，但是应用比较广泛的应该是和AspectJ表达式的方式。

AspectJ是一个“对Java的AOP增强”。它最早是其实是一门语言，我们跟写Java代码一样写它，然后静态编译之后，就有了AOP的功能。下面是一段AspectJ代码：

aspect PointObserving {

private Vector Point.observers = new Vector();

public static void addObserver(Point p, Screen s) {

p.observers.add(s);

}

public static void removeObserver(Point p, Screen s) {

p.observers.remove(s);

}

...

}

这种方式无疑太重了，为了AOP，还要适应一种语言？所以现在使用也不多，但是它的Pointcut表达式被Spring借鉴了过来。于是我们实现了一个AspectJExpressionPointcut：

@Test

public void testMethodInterceptor() throws Exception {

String expression = "execution(\* us.codecraft.tinyioc.\*.\*(..))";

AspectJExpressionPointcut aspectJExpressionPointcut = new AspectJExpressionPointcut();

aspectJExpressionPointcut.setExpression(expression);

boolean matches = aspectJExpressionPointcut.getMethodMatcher().matches(HelloWorldServiceImpl.class.getDeclaredMethod("helloWorld"),HelloWorldServiceImpl.class);

Assert.assertTrue(matches);

}

## 9.step9-将AOP融入Bean的创建过程

git checkout step-9-auto-create-aop-proxy

万事俱备，只欠东风！现在我们有了Pointcut和Weave技术，一个AOP已经算是完成了，但是它还没有结合到Spring中去。怎么进行结合呢？Spring给了一个巧妙的答案：使用BeanPostProcessor。

BeanPostProcessor是BeanFactory提供的，在Bean初始化过程中进行扩展的接口。只要你的Bean实现了BeanPostProcessor接口，那么Spring在初始化时，会优先找到它们，并且在Bean的初始化过程中，调用这个接口，从而实现对BeanFactory核心无侵入的扩展。

那么我们的AOP是怎么实现的呢？我们知道，在AOP的xml配置中，我们会写这样一句话：

<aop:aspectj-autoproxy/>

它其实相当于：

<bean id="autoProxyCreator" class="org.springframework.aop.aspectj.autoproxy.AspectJAwareAdvisorAutoProxyCreator"></bean>

AspectJAwareAdvisorAutoProxyCreator就是AspectJ方式实现织入的核心。它其实是一个BeanPostProcessor。在这里它会扫描所有Pointcut，并对bean做织入。

为了简化xml配置，我在tiny-spring中直接使用Bean的方式，而不是用aop前缀进行配置：

<bean id="autoProxyCreator" class="us.codecraft.tinyioc.aop.AspectJAwareAdvisorAutoProxyCreator"></bean>

<bean id="timeInterceptor" class="us.codecraft.tinyioc.aop.TimerInterceptor"></bean>

<bean id="aspectjAspect" class="us.codecraft.tinyioc.aop.AspectJExpressionPointcutAdvisor">

<property name="advice" ref="timeInterceptor"></property>

<property name="expression" value="execution(\* us.codecraft.tinyioc.\*.\*(..))"></property>

</bean>

TimerInterceptor实现了MethodInterceptor（实际上Spring中还有Advice这样一个角色，为了简单，就直接用MethodInterceptor了）。

至此，一个AOP基本完工。

## 10.step10-使用CGLib进行类的织入

git checkout step-10-invite-cglib-and-aopproxy-factory

前面的JDK动态代理只能对接口进行代理，对于类则无能为力。这里我们需要一些字节码操作技术。这方面大概有几种选择：ASM，CGLib和javassist，后两者是对ASM的封装。Spring中使用了CGLib。

在这一步，我们还要定义一个工厂类ProxyFactory，用于根据TargetSource类型自动创建代理，这样就需要在调用者代码中去进行判断。

另外我们实现了Cglib2AopProxy，使用方式和JdkDynamicAopProxy是完全相同的。

有一个细节是CGLib创建的代理是没有注入属性的， Spring的解决方式是：CGLib仅作代理，任何属性都保存在TargetSource中，使用MethodInterceptor=>TargetSource的方式进行调用。

至此，AOP部分完工，Spring的核心也基本成型。除去import语句，main下面一共是1026行。下篇博文会对Spring进行一个整体的分析。

## 项目地址

依然附上项目地址：<https://github.com/code4craft/tiny-spring>