芋道源码 —— 知识星球

2019-09-18
Spring

【死磕 Spring】—— IoC 之 Bean 的实例化策略：InstantiationStrategy

本文主要基于 Spring 5.0.6.RELEASE

摘要: 原创出处 http://cmsblogs.com/?p=todo 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老艿艿」略作修改，记录在理解过程中，参考的资料。

在开始分析 InstantiationStrategy 之前，我们先来简单回顾下 Bean 的实例化过程：

1. Bean 的创建，主要是 `AbstractAutowireCapableBeanFactory#doCreateBean(...)` 方法。在这个方法中有 Bean 的实例化、属性注入和初始化过程，对于 Bean 的实例化过程这是根据 Bean 的类型来判断的，如果是单例模式，则直接从 `factoryBeanInstanceCache` 缓存中获取，否则调用 `#createBeanInstance(...)` 方法来创建。
2. 在 `#createBeanInstance(...)` 方法中，如果 Supplier 不为空，则调用 `#obtainFromSupplier(...)` 实例化 bean。如果 `factory` 不为空，则调用 `#instantiateUsingFactoryMethod(...)` 方法来实例化 Bean 。如果都不是，则调用 `#instantiateBean(...)` 方法来实例化 Bean 。但是无论是 `#instantiateUsingFactoryMethod(...)` 方法，还是 `#instantiateBean()` 方法，最后都一定会调用到 InstantiationStrategy 接口的 `#instantiate(...)` 方法。

1. InstantiationStrategy

InstantiationStrategy 接口定义了 Spring Bean 实例化的策略，根据创建对象情况的不同，提供了三种策略：无参构造方法、有参构造方法、工厂方法。代码如下：

```
public interface InstantiationStrategy {

    /**
    * 默认构造方法
    */
    Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner)
    throws BeansException;

    /**
    * 指定构造方法
    */
    Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner,
    Constructor<?> ctor, @Nullable Object... args) throws BeansException;

    /**
    * 工厂方法
    */
    Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner,
    @Nullable Object factoryBean, Method factoryMethod, @Nullable Object... args)
    throws BeansException;

}
```

2. SimpleInstantiationStrategy

InstantiationStrategy 接口有两个实现类：SimpleInstantiationStrategy 和 CglibSubclassingInstantiationStrategy。

SimpleInstantiationStrategy 对以上三个方法都做了简单的实现。

① 如果是工厂方法实例化，则直接使用反射创建对象，如下：

```
// SimpleInstantiationStrategy.java

@Override
public Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner,
        @Nullable Object factoryBean, final Method factoryMethod, Object... args) {
    try {
        // 设置 Method 可访问
        if (System.getSecurityManager() != null) {
            AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
                ReflectionUtils.makeAccessible(factoryMethod);
                return null;
            });
        } else {
            ReflectionUtils.makeAccessible(factoryMethod);
        }

        // 获得原 Method 对象
        Method priorInvokedFactoryMethod = currentlyInvokedFactoryMethod.get();
        try {
            // 设置新的 Method 对象，到 currentlyInvokedFactoryMethod 中
            currentlyInvokedFactoryMethod.set(factoryMethod);
            // 创建 Bean 对象
            Object result = factoryMethod.invoke(factoryBean, args);
            // 未创建，则创建 NullBean 对象
            if (result == null) {
                result = new NullBean();
            }
            return result;
        } finally {
            // 设置老的 Method 对象，到 currentlyInvokedFactoryMethod 中
            if (priorInvokedFactoryMethod != null) {
                currentlyInvokedFactoryMethod.set(priorInvokedFactoryMethod);
            } else {
                currentlyInvokedFactoryMethod.remove();
            }
        }
    // 一大堆 catch 异常
    } catch (IllegalArgumentException ex) {
        throw new BeanInstantiationException(factoryMethod,
                "Illegal arguments to factory method '" + factoryMethod.getName() + "'; " +
                "args: " + StringUtils.arrayToCommaDelimitedString(args), ex);
    } catch (IllegalAccessException ex) {
        throw new BeanInstantiationException(factoryMethod,
                "Cannot access factory method '" + factoryMethod.getName() + "'; is it public?", ex);
    } catch (InvocationTargetException ex) {
        String msg = "Factory method '" + factoryMethod.getName() + "' threw exception";
        if (bd.getFactoryBeanName() != null && owner instanceof ConfigurableBeanFactory &&
                ((ConfigurableBeanFactory) owner).isCurrentlyInCreation(bd.getFactoryBeanName())) {
            msg = "Circular reference involving containing bean '" + bd.getFactoryBeanName() + "' - consider " +
                    "declaring the factory method as static for independence from its containing instance. " + msg;
        }
        throw new BeanInstantiationException(factoryMethod, msg, ex.getTargetException());
    }
}
```

② 如果是构造方法实例化，则是先判断是否有 MethodOverrides，如果没有则是直接使用反射，如果有则就需要 CGLIB 实例化对象。如下：

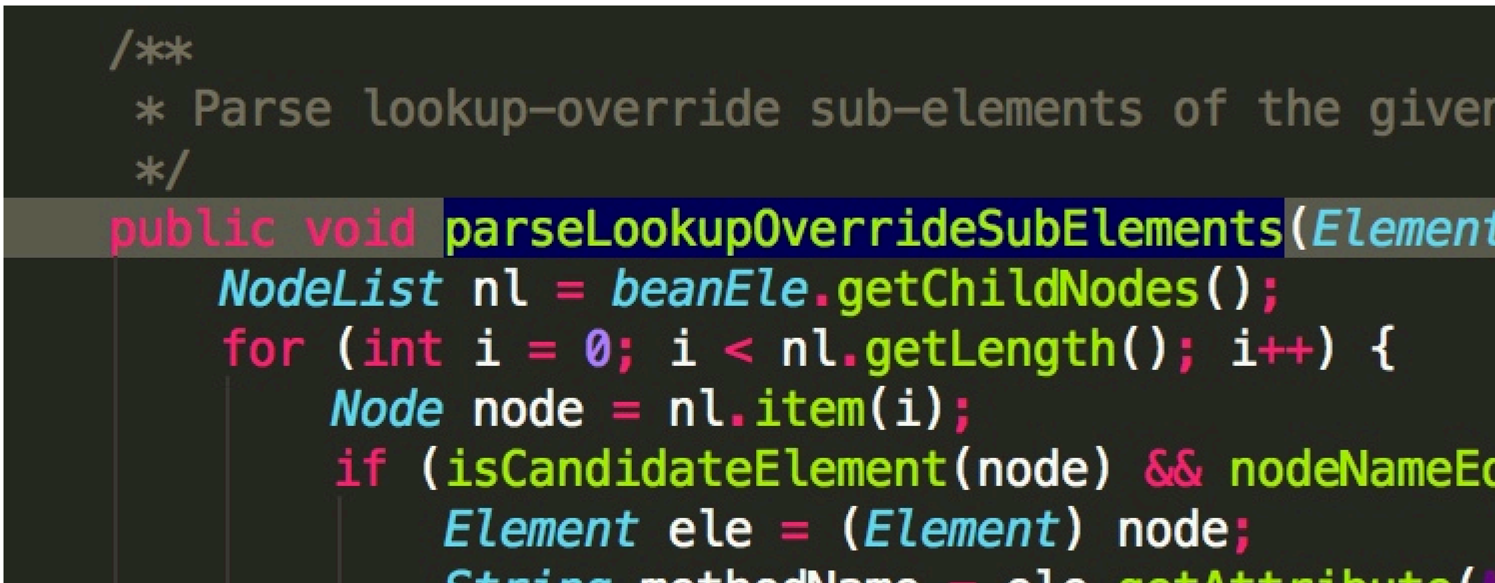
```
    // SimpleInstantiationStrategy.java

    // 默认构造方法
    @Override
    public Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner) {
     // Don't override the class with CGLIB if no overrides.
        // 没有覆盖，直接使用反射实例化即可
        if (!bd.hasMethodOverrides()) {
            Constructor<?> constructorToUse;
         synchronized (bd.constructorArgumentLock) {
                // 获得构造方法 constructorToUse
                constructorToUse = (Constructor<?>) bd.resolvedConstructorOrFactoryMethod;
             if (constructorToUse == null) {
                 final Class<?> clazz = bd.getBeanClass();
                 // 如果是接口，抛出 BeanInstantiationException 异常
                 if (clazz.isInterface()) {
                     throw new BeanInstantiationException(clazz, "Specified class is an interface");
                    }
                 try {
                        // 从 clazz 中，获得构造方法
                     if (System.getSecurityManager() != null) { // 安全模式
                            constructorToUse = AccessController.doPrivileged(
                                    (PrivilegedExceptionAction<Constructor<?>>) clazz::getDeclaredConstructor);
                        } else {
                            constructorToUse =    clazz.getDeclaredConstructor();
                        }
                     // 标记 resolvedConstructorOrFactoryMethod 属性
                        bd.resolvedConstructorOrFactoryMethod = constructorToUse;
                    } catch (Throwable ex) {
                     throw new BeanInstantiationException(clazz, "No default constructor found", ex);
                    }
                }
            }
            // 通过 BeanUtils 直接使用构造器对象实例化 Bean 对象
            return BeanUtils.instantiateClass(constructorToUse);
        } else {
         // Must generate CGLIB subclass.
            // 生成 CGLIB 创建的子类对象
            return instantiateWithMethodInjection(bd, beanName, owner);
        }
    }

    // 指定构造方法
    @Override
    public Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner,
         final Constructor<?> ctor, Object... args) {
        // 没有覆盖，直接使用反射实例化即可
     if (!bd.hasMethodOverrides()) {
         if (System.getSecurityManager() != null) {
                // 设置构造方法，可访问
            // use own privileged to change accessibility (when security is on)
                AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
                    ReflectionUtils.makeAccessible(ctor);
                 return null;
                });
            }
            // 通过 BeanUtils 直接使用构造器对象实例化 Bean 对象
         return BeanUtils.instantiateClass(ctor, args);
        } else {
            // 生成 CGLIB 创建的子类对象
         return instantiateWithMethodInjection(bd, beanName, owner, ctor, args);
        }
    }
```

SimpleInstantiationStrategy 对 #instantiateWithMethodInjection(RootBeanDefinition bd, String beanName, BeanFactory owner, Constructor<?> ctor, Object... args) 的实现任务交给了子类 CglibSubclassingInstantiationStrategy 。

3. MethodOverrides

对于 MethodOverrides，如果读者是跟着小编文章一路跟过来的话一定不会陌生，在 BeanDefinitionParserDelegate 类解析 <bean/> 的时候是否还记得这两个方法：#parseLookupOverrideSubElements(...) 和 #parseReplacedMethodSubElements(...) 这两个方法分别用于解析 lookup-method 和 replaced-method 属性。

其中，#parseLookupOverrideSubElements(...) 源码如下：


```
    /**
     * Parse lookup-override sub-elements of the giver
     */
    public void parseLookupOverrideSubElements(Element
        NodeList nl = beanEle.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (isCandidateElement(node) && nodeNameEq
                Element ele = (Element) node;
```

更多关于 lookup-method 和 replaced-method 请看：[【死磕 Spring】—— IoC 之解析 bean 标签：meta、lookup-method、replace-method]

4. CglibSubclassingInstantiationStrategy

类 CglibSubclassingInstantiationStrategy 为 Spring 实例化 Bean 的默认实例化策略，其主要功能还是对父类功能进行补充：其父类将 CGLIB 的实例化策略委托其实现。

```
// SimpleInstantiationStrategy.java

protected Object instantiateWithMethodInjection(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner)
 throw new UnsupportedOperationException("Method Injection not supported in SimpleInstantiationStrategy");
}

// CglibSubclassingInstantiationStrategy.java

@Override
protected Object instantiateWithMethodInjection(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner)
 return instantiateWithMethodInjection(bd, beanName, owner, null);
}
```

CglibSubclassingInstantiationStrategy 实例化 Bean 策略，是通过其内部类 CglibSubclassCreator 来实现的。代码如下：

```
// CglibSubclassingInstantiationStrategy.java

@Override
protected Object instantiateWithMethodInjection(RootBeanDefinition bd, @Nullable String beanName, BeanFactory o
 // Must generate CGLIB subclass...
    // 通过CGLIB生成一个子类对象
 return new CglibSubclassCreator(bd, owner).instantiate(ctor, args);
}
```

创建 CglibSubclassCreator 实例，然后调用其 #instantiate(Constructor<?> ctor, Object... args) 方法，该方法用于动态创建子类实例，同时实现所需要的 lookups（lookup-method、replace-method）。

```
// CglibSubclassingInstantiationStrategy.java#CglibSubclassCreator

public Object instantiate(@Nullable Constructor<?> ctor, Object... args) {
    // <x> 通过 Cglib 创建一个代理类
    Class<?> subclass = createEnhancedSubclass(this.beanDefinition);
    Object instance;
    // <y> 没有构造器，通过 BeanUtils 使用默认构造器创建一个bean实例
    if (ctor == null) {
        instance = BeanUtils.instantiateClass(subclass);
    } else {
        try {
            // 获取代理类对应的构造器对象，并实例化 bean
            Constructor<?> enhancedSubclassConstructor = subclass.getConstructor(ctor.getParameterTypes());
            instance = enhancedSubclassConstructor.newInstance(args);
        } catch (Exception ex) {
            throw new BeanInstantiationException(this.beanDefinition.getBeanClass(),
                "Failed to invoke constructor for CGLIB enhanced subclass [" + subclass.getName() + "]", e
        }
    }
    // SPR-10785: set callbacks directly on the instance instead of in the
    // enhanced class (via the Enhancer) in order to avoid memory leaks.
    // 为了避免 memory leaks 异常，直接在 bean 实例上设置回调对象
    Factory factory = (Factory) instance;
    factory.setCallbacks(new Callback[] {NoOp.INSTANCE,
            new LookupOverrideMethodInterceptor(this.beanDefinition, this.owner),
            new ReplaceOverrideMethodInterceptor(this.beanDefinition, this.owner)});
    return instance;
}
```

- 在 `<x>` 处，调用 `#createEnhancedSubclass(RootBeanDefinition beanDefinition)` 方法，为提供的 BeanDefinition 创建 bean 类的增强子类。代码如下：

```
// CglibSubclassingInstantiationStrategy.java#CglibSubclassCreator

private Class<?> createEnhancedSubclass(RootBeanDefinition beanDefinition) {
    // 创建 Enhancer 对象
    Enhancer enhancer = new Enhancer();
    // 设置 Bean 类
    enhancer.setSuperclass(beanDefinition.getBeanClass());
    // 设置 Spring 的命名策略
    enhancer.setNamingPolicy(SpringNamingPolicy.INSTANCE);
    // 设置生成策略
    if (this.owner instanceof ConfigurableBeanFactory) {
        ClassLoader cl = ((ConfigurableBeanFactory) this.owner).getBeanClassLoader();
        enhancer.setStrategy(new ClassLoaderAwareGeneratorStrategy(cl));
    }
    // 过滤，自定义逻辑来指定调用的callback下标
    enhancer.setCallbackFilter(new MethodOverrideCallbackFilter(beanDefinition));
    enhancer.setCallbackTypes(CALLBACK_TYPES);
    return enhancer.createClass();
}
```

  - CGLIB 的标准 API 的使用。
- `<y>` 处，获取子类增强 `subclass` 后，如果 Constructor 实例 `ctr` 为空，则调用默认构造函数（`BeanUtils#instantiateClass(subclass)`）来实例化类，否则则根据构造函数类型获取具体的构造器，调用 `Constructor#newInstance(args)` 方法来实例化类。

4.1 MethodOverrideCallbackFilter

在 `<x>` 处调用的 `#createEnhancedSubclass(RootBeanDefinition beanDefinition)` 方法，我们注意两行代码：

```
// CglibSubclassingInstantiationStrategy.java#CglibSubclassCreator

enhancer.setCallbackFilter(new MethodOverrideCallbackFilter(beanDefinition));
enhancer.setCallbackTypes(CALLBACK_TYPES);
```

通过 MethodOverrideCallbackFilter 来定义调用 callback 类型。

MethodOverrideCallbackFilter 是用来定义 CGLIB 回调过滤方法的拦截器行为，它继承

CglibIdentitySupport 实现 CallbackFilter 接口。

CallbackFilter 是 CGLIB 的一个回调过滤器。
CglibIdentitySupport 则为 CGLIB 提供 #hashCode() 和 #equals(Object o) 方法，以确保 CGLIB 不会为每个 Bean 生成不同的类。

MethodOverrideCallbackFilter 实现 CallbackFilter 的 #accept(Method method) 方法，代码如下：

```
// CglibSubclassingInstantiationStrategy.java#MethodOverrideCallbackFilter

@Override
public int accept(Method method) {
    MethodOverride methodOverride = getBeanDefinition().getMethodOverrides().getOverride(method);
 if (logger.isTraceEnabled()) {
        logger.trace("Override for '" + method.getName() + "' is [" + methodOverride + "]");
    }
 if (methodOverride == null) {
     return PASSTHROUGH;
    } else if (methodOverride instanceof LookupOverride) {
     return LOOKUP_OVERRIDE;
    } else if (methodOverride instanceof ReplaceOverride) {
     return METHOD_REPLACER;
    }
 throw new UnsupportedOperationException("Unexpected MethodOverride subclass: " +
            methodOverride.getClass().getName());
}
```

根据 BeanDefinition 中定义的 MethodOverride 不同，返回不同的值， 这里返回的 PASSTHROUGH 、LOOKUP_OVERRIDE、METHOD_REPLACER 都是 Callback 数组的下标，这里对应的数组为 CALLBACK_TYPES 数组，如下：

```
// CglibSubclassingInstantiationStrategy.java#CglibSubclassCreator

private static final Class<?>[] CALLBACK_TYPES = new Class<?>[] {
    NoOp.class,
    LookupOverrideMethodInterceptor.class,
    ReplaceOverrideMethodInterceptor.class
};
```

- 这里又定义了两个熟悉的拦截器 ：LookupOverrideMethodInterceptor 和 ReplaceOverrideMethodInterceptor，两个拦截器分别对应两个不同的 callback 业务。详细解析，见 「4.2 LookupOverrideMethodInterceptor」 和 「4.3 ReplaceOverrideMethodInterceptor」 中。

4.2 LookupOverrideMethodInterceptor

```
// CglibSubclassingInstantiationStrategy.java#LookupOverrideMethodInterceptor

private static class LookupOverrideMethodInterceptor extends CglibIdentitySupport implements MethodInterceptor {

    private final BeanFactory owner;

    public LookupOverrideMethodInterceptor(RootBeanDefinition beanDefinition, BeanFactory owner) {
        super(beanDefinition);
        this.owner = owner;
    }

    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy mp) throws Throwable {
        // Cast is safe, as CallbackFilter filters are used selectively.
        // 获得 method 对应的 LookupOverride 对象
        LookupOverride lo = (LookupOverride) getBeanDefinition().getMethodOverrides().getOverride(method);
        Assert.state(lo != null, "LookupOverride not found");
        // 获得参数
        Object[] argsToUse = (args.length > 0 ? args : null);  // if no-arg, don't insist on args at all
        // 获得 Bean
        if (StringUtils.hasText(lo.getBeanName())) { // Bean 的名字
            return (argsToUse != null ? this.owner.getBean(lo.getBeanName(), argsToUse) :
                    this.owner.getBean(lo.getBeanName()));
        } else { // Bean 的类型
            return (argsToUse != null ? this.owner.getBean(method.getReturnType(), argsToUse) :
                    this.owner.getBean(method.getReturnType()));
        }
    }
}
```

4.3 ReplaceOverrideMethodInterceptor

```
// CglibSubclassingInstantiationStrategy.java#ReplaceOverrideMethodInterceptor

private static class ReplaceOverrideMethodInterceptor extends CglibIdentitySupport implements MethodInterceptor {

    private final BeanFactory owner;

    public ReplaceOverrideMethodInterceptor(RootBeanDefinition beanDefinition, BeanFactory owner) {
        super(beanDefinition);
        this.owner = owner;
    }

    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy mp) throws Throwable {
        // 获得 method 对应的 LookupOverride 对象
        ReplaceOverride ro = (ReplaceOverride) getBeanDefinition().getMethodOverrides().getOverride(method);
        Assert.state(ro != null, "ReplaceOverride not found");
        // TODO could cache if a singleton for minor performance optimization
        // 获得 MethodReplacer 对象
        MethodReplacer mr = this.owner.getBean(ro.getMethodReplacerBeanName(), MethodReplacer.class);
        // 执行替换
        return mr.reimplement(obj, method, args);
    }
}
```

通过这两个拦截器，再加上这篇博客：【死磕 Spring】—— IoC 之解析 bean 标签：meta、lookup-method、replace-method，是不是一道绝佳的美食。

文章目录