芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2019-06-07
Spring

【死磕 Spring】—— IoC 之加载 Bean：创建 Bean（三）之实例化 Bean 对象(2)

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。

`#createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args)` 方法，用于实例化 Bean 对象。它会根据不同情况，选择不同的实例化策略来完成 Bean 的初始化，主要包括：

> Supplier 回调：`#obtainFromSupplier(final String beanName, final RootBeanDefinition mbd)` 方法。
> 工厂方法初始化：`#instantiateUsingFactoryMethod(String beanName, RootBeanDefinition mbd, @Nullable Object[] explicitArgs)` 方法。
> 构造函数自动注入初始化：`#autowireConstructor(final String beanName, final RootBeanDefinition mbd, Constructor<?>[] chosenCtors, final Object[] explicitArgs)` 方法。
> 默认构造函数注入：`#instantiateBean(final String beanName, final RootBeanDefinition mbd)` 方法。

在上篇博客(《【死磕 Spring】—— IoC 之加载 bean：创建 bean（二）》) 中，分析了前两种 Supplier 回调和工厂方法初始化。这篇博文，分析后两种构造函数注入。

1. autowireConstructor

这个初始化方法，我们可以简单理解为是带有参数的构造方法，来初始化 Bean 对象。代码段如下：

```
// AbstractAutowireCapableBeanFactory.java

protected BeanWrapper autowireConstructor(String beanName, RootBeanDefinition mbd, @Nullable Constructor<?>[] ctors,
    return new ConstructorResolver(this).autowireConstructor(beanName, mbd, ctors, explicitArgs);
}

// ConstructorResolver.java

public BeanWrapper autowireConstructor(String beanName, RootBeanDefinition mbd,
```

```
    @Nullable Constructor<?>[] chosenCtors, @Nullable Object[] explicitArgs) {
// 封装 BeanWrapperImpl 对象，并完成初始化
BeanWrapperImpl bw = new BeanWrapperImpl();
this.beanFactory.initBeanWrapper(bw);

// 获得 constructorToUse、argsHolderToUse、argsToUse
Constructor<?> constructorToUse = null; // 构造函数
ArgumentsHolder argsHolderToUse = null; // 构造参数
Object[] argsToUse = null; // 构造参数

// 确定构造参数
// 如果 getBean() 已经传递，则直接使用
if (explicitArgs != null) {
    argsToUse = explicitArgs;
} else {
    // 尝试从缓存中获取
    Object[] argsToResolve = null;
    synchronized (mbd.constructorArgumentLock) {
        // 缓存中的构造函数或者工厂方法
        constructorToUse = (Constructor<?>) mbd.resolvedConstructorOrFactoryMethod;
        if (constructorToUse != null && mbd.constructorArgumentsResolved) {
            // Found a cached constructor...
            // 缓存中的构造参数
            argsToUse = mbd.resolvedConstructorArguments;
            if (argsToUse == null) {
                argsToResolve = mbd.preparedConstructorArguments;
            }
        }
    }
    // 缓存中存在,则解析存储在 BeanDefinition 中的参数
    // 如给定方法的构造函数 A(int ,int )，则通过此方法后就会把配置文件中的("1","1")转换为 (1,1)
    // 缓存中的值可能是原始值也有可能是最终值
    if (argsToResolve != null) {
        argsToUse = resolvePreparedArguments(beanName, mbd, bw, constructorToUse, argsToResolve, true);
    }
}

// 没有缓存，则尝试从配置文件中获取参数
if (constructorToUse == null || argsToUse == null) {
    // Take specified constructors, if any.
    // 如果 chosenCtors 未传入，则获取构造方法们
    Constructor<?>[] candidates = chosenCtors;
    if (candidates == null) {
        Class<?> beanClass = mbd.getBeanClass();
        try {
            candidates = (mbd.isNonPublicAccessAllowed() ?
                    beanClass.getDeclaredConstructors() : beanClass.getConstructors());
        } catch (Throwable ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Resolution of declared constructors on bean Class [" + beanClass.getName() +
                    "] from ClassLoader [" + beanClass.getClassLoader() + "] failed", ex);
        }
    }

    // TODO 芋艿 创建 Bean
    if (candidates.length == 1 && explicitArgs == null && !mbd.hasConstructorArgumentValues()) {
        Constructor<?> uniqueCandidate = candidates[0];
        if (uniqueCandidate.getParameterCount() == 0) {
            synchronized (mbd.constructorArgumentLock) {
                mbd.resolvedConstructorOrFactoryMethod = uniqueCandidate;
```

```
            mbd.constructorArgumentsResolved = true;
            mbd.resolvedConstructorArguments = EMPTY_ARGS;
        }
        bw.setBeanInstance(instantiate(beanName, mbd, uniqueCandidate, EMPTY_ARGS));
        return bw;
    }
}

// 是否需要解析构造器
// Need to resolve the constructor.
boolean autowiring = (chosenCtors != null ||
        mbd.getResolvedAutowireMode() == AutowireCapableBeanFactory.AUTOWIRE_CONSTRUCTOR);
// 用于承载解析后的构造函数参数的值
ConstructorArgumentValues resolvedValues = null;
int minNrOfArgs;
if (explicitArgs != null) {
    minNrOfArgs = explicitArgs.length;
} else {
    // 从 BeanDefinition 中获取构造参数，也就是从配置文件中提取构造参数
    ConstructorArgumentValues cargs = mbd.getConstructorArgumentValues();
    resolvedValues = new ConstructorArgumentValues();
    // 解析构造函数的参数
    // 将该 bean 的构造函数参数解析为 resolvedValues 对象，其中会涉及到其他 bean
    minNrOfArgs = resolveConstructorArguments(beanName, mbd, bw, cargs, resolvedValues);
}

// 对构造函数进行排序处理
// public 构造函数优先参数数量降序，非public 构造函数参数数量降序
AutowireUtils.sortConstructors(candidates);

// 最小参数类型权重
int minTypeDiffWeight = Integer.MAX_VALUE;
Set<Constructor<?>> ambiguousConstructors = null;
LinkedList<UnsatisfiedDependencyException> causes = null;

// 迭代所有构造函数
for (Constructor<?> candidate : candidates) {
    // 获取该构造函数的参数类型
    Class<?>[] paramTypes = candidate.getParameterTypes();

    // 如果已经找到选用的构造函数或者需要的参数个数小于当前的构造函数参数个数，则终止。
    // 因为，已经按照参数个数降序排列了
    if (constructorToUse != null && argsToUse.length > paramTypes.length) {
        // Already found greedy constructor that can be satisfied ->
        // do not look any further, there are only less greedy constructors left.
        break;
    }
    // 参数个数不等，继续
    if (paramTypes.length < minNrOfArgs) {
        continue;
    }

    // 参数持有者 ArgumentsHolder 对象
    ArgumentsHolder argsHolder;
    if (resolvedValues != null) {
        try {
            // 注释上获取参数名称
            String[] paramNames = ConstructorPropertiesChecker.evaluate(candidate, paramTypes.length);
            if (paramNames == null) {
                // 获取构造函数、方法参数的探测器
```

```
                    ParameterNameDiscoverer pnd = this.beanFactory.getParameterNameDiscoverer();
                    if (pnd != null) {
                        // 通过探测器获取构造函数的参数名称
                        paramNames = pnd.getParameterNames(candidate);
                    }
                }
                // 根据构造函数和构造参数，创建参数持有者 ArgumentsHolder 对象
                argsHolder = createArgumentArray(beanName, mbd, resolvedValues, bw, paramTypes, paramNames,
                        getUserDeclaredConstructor(candidate), autowiring, candidates.length == 1);
            } catch (UnsatisfiedDependencyException ex) {
                // 若发生 UnsatisfiedDependencyException 异常，添加到 causes 中。
                if (logger.isTraceEnabled()) {
                    logger.trace("Ignoring constructor [" + candidate + "] of bean '" + beanName + "': " + ex);
                }
                // Swallow and try next constructor.
                if (causes == null) {
                    causes = new LinkedList<>();
                }
                causes.add(ex);
                continue; // continue ，继续执行
            }
        } else {
            // continue 构造函数没有参数
            // Explicit arguments given -> arguments length must match exactly.
            if (paramTypes.length != explicitArgs.length) {
                continue;
            }
            // 根据 explicitArgs ，创建 ArgumentsHolder 对象
            argsHolder = new ArgumentsHolder(explicitArgs);
        }

        // isLenientConstructorResolution 判断解析构造函数的时候是否以宽松模式还是严格模式
        // 严格模式：解析构造函数时，必须所有的都需要匹配，否则抛出异常
        // 宽松模式：使用具有"最接近的模式"进行匹配
        // typeDiffWeight：类型差异权重
        int typeDiffWeight = (mbd.isLenientConstructorResolution() ?
                argsHolder.getTypeDifferenceWeight(paramTypes) : argsHolder.getAssignabilityWeight(paramTypes));
        // Choose this constructor if it represents the closest match.
        // 如果它代表着当前最接近的匹配则选择其作为构造函数
        if (typeDiffWeight < minTypeDiffWeight) {
            constructorToUse = candidate;
            argsHolderToUse = argsHolder;
            argsToUse = argsHolder.arguments;
            minTypeDiffWeight = typeDiffWeight;
            ambiguousConstructors = null;
        } else if (constructorToUse != null && typeDiffWeight == minTypeDiffWeight) {
            if (ambiguousConstructors == null) {
                ambiguousConstructors = new LinkedHashSet<>();
                ambiguousConstructors.add(constructorToUse);
            }
            ambiguousConstructors.add(candidate);
        }
    }

    // 没有可执行的工厂方法，抛出异常
    if (constructorToUse == null) {
        if (causes != null) {
            UnsatisfiedDependencyException ex = causes.removeLast();
            for (Exception cause : causes) {
                this.beanFactory.onSuppressedException(cause);
```

```
                }
                throw ex;
            }
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Could not resolve matching constructor " +
                    "(hint: specify index/type/name arguments for simple parameters to avoid type ambiguities)");
        } else if (ambiguousConstructors != null && !mbd.isLenientConstructorResolution()) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Ambiguous constructor matches found in bean '" + beanName + "' " +
                    "(hint: specify index/type/name arguments for simple parameters to avoid type ambiguities): " +
                    ambiguousConstructors);
        }

        if (explicitArgs == null) {
            // 将解析的构造函数加入缓存
            argsHolderToUse.storeCache(mbd, constructorToUse);
        }
    }

    // 创建 Bean 对象，并设置到 bw 中
    bw.setBeanInstance(instantiate(beanName, mbd, constructorToUse, argsToUse));
    return bw;
}
```

代码与 `#instantiateUsingFactoryMethod(String beanName, RootBeanDefinition mbd, @Nullable Object[] explicitArgs)` 方法，一样，又长又难懂。但是如果理解了 `#instantiateUsingFactoryMethod(...)` 方法的初始化 bean 的过程，那么 `#autowireConstructor(...)` 方法，也不存在什么难的地方了。
一句话概括：首先确定构造函数参数、构造函数，然后调用相应的初始化策略进行 bean 的初始化。关于如何确定构造函数、构造参数，该部分逻辑和 `#instantiateUsingFactoryMethod(...)` 方法，基本一致。所以这里不再重复阐述了，具体过程请移步【死磕 Spring】—— IoC 之加载 bean：创建 bean（二），这里我们重点分析初始化策略。

1.1 instantiate

```
// BeanUtils.java

public static <T> T instantiateClass(Constructor<T> ctor, Object... args) throws BeanInstantiationException {
    Assert.notNull(ctor, "Constructor must not be null");
    try {
        // 设置构造方法，可访问
        ReflectionUtils.makeAccessible(ctor);
        // 使用构造方法，创建对象
        return (KotlinDetector.isKotlinReflectPresent() && KotlinDetector.isKotlinType(ctor.getDeclaringClass()) ?
                KotlinDelegate.instantiateClass(ctor, args) : ctor.newInstance(args));
    // 各种异常的翻译，最终统一抛出 BeanInstantiationException 异常
    } catch (InstantiationException ex) {
        throw new BeanInstantiationException(ctor, "Is it an abstract class?", ex);
    } catch (IllegalAccessException ex) {
        throw new BeanInstantiationException(ctor, "Is the constructor accessible?", ex);
    } catch (IllegalArgumentException ex) {
        throw new BeanInstantiationException(ctor, "Illegal arguments for constructor", ex);
    } catch (InvocationTargetException ex) {
        throw new BeanInstantiationException(ctor, "Constructor threw exception", ex.getTargetException());
    }
}
```

<1> 首先，是获取实例化 Bean 的策略 InstantiationStrategy 对象。

<2> 然后，调用其 #instantiate(RootBeanDefinition bd, String beanName, BeanFactory owner, Constructor<?> ctor, Object... args) 方法，该方法在 SimpleInstantiationStrategy 中实现。代码如下：

```
// SimpleInstantiationStrategy.java

@Override
public Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner,
        final Constructor<?> ctor, Object... args) {
    // <x1> 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 {
        // <x2> 生成 CGLIB 创建的子类对象
        return instantiateWithMethodInjection(bd, beanName, owner, ctor, args);
    }
}
```

- <x1> 如果该 bean 没有配置 lookup-method、replaced-method 标签或者 @Lookup 注解，则直接通过反射的方式实例化 Bean 对象即可，方便快捷。详细解析，见 「1.1.1 反射创建 Bean 对象」 中。
- <x2> 但是，如果存在需要覆盖的方法或者动态替换的方法时，则需要使用 CGLIB 进行动态代理，因为可以在创建代理的同时将动态方法织入类中。详细解析，见 「1.1.2 CGLIB 创建 Bean 对象」 中。

1.1.1 反射创建 Bean 对象

调用工具类 BeanUtils 的 #instantiateClass(Constructor<T> ctor, Object... args) 方法，完成反射工作，创建对象。代码如下：

```
// BeanUtils.java

public static <T> T instantiateClass(Constructor<T> ctor, Object... args) throws BeanInstantiationException {
    Assert.notNull(ctor, "Constructor must not be null");
    try {
        // 设置构造方法，可访问
        ReflectionUtils.makeAccessible(ctor);
        // 使用构造方法，创建对象
        return (KotlinDetector.isKotlinReflectPresent() && KotlinDetector.isKotlinType(ctor.getDeclaringClass()) ?
                KotlinDelegate.instantiateClass(ctor, args) : ctor.newInstance(args));
    // 各种异常的翻译，最终统一抛出 BeanInstantiationException 异常
    } catch (InstantiationException ex) {
        throw new BeanInstantiationException(ctor, "Is it an abstract class?", ex);
    } catch (IllegalAccessException ex) {
        throw new BeanInstantiationException(ctor, "Is the constructor accessible?", ex);
    } catch (IllegalArgumentException ex) {
```

```
            throw new BeanInstantiationException(ctor, "Illegal arguments for constructor", ex);
        } catch (InvocationTargetException ex) {
            throw new BeanInstantiationException(ctor, "Constructor threw exception", ex.getTargetException());
        }
    }
```

1.1.2 CGLIB 创建 Bean 对象

```
// SimpleInstantiationStrategy.java

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

方法默认是没有实现的，具体过程由其子类 `org.springframework.beans.factory.support.CglibSubclassingInstantiationStrategy` 来实现。代码如下：

```
// CglibSubclassingInstantiationStrategy.java

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

- 创建一个 CglibSubclassCreator 对象，后调用其 `#instantiate(Constructor<?> ctor, Object... args)` 方法，生成其子类对象。代码如下：

```
// CglibSubclassingInstantiationStrategy.java

public Object instantiate(@Nullable Constructor<?> ctor, Object... args) {
    // 通过 Cglib 创建一个代理类
    Class<?> subclass = createEnhancedSubclass(this.beanDefinition);
    Object instance;
    // 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() +
        }
    }
    // 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;
        }
```

- 到这类 CGLIB 的方式分析完毕了，当然这里还没有具体分析 CGLIB 生成子类的详细过程，具体的过程等后续分析 AOP 的时候再详细地介绍。

2. instantiateBean

```
// AbstractAutowireCapableBeanFactory.java

protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefinition mbd) {
    try {
        Object beanInstance;
        final BeanFactory parent = this;
        // 安全模式
        if (System.getSecurityManager() != null) {
            beanInstance = AccessController.doPrivileged((PrivilegedAction<Object>) () ->
                    // 获得 InstantiationStrategy 对象，并使用它，创建 Bean 对象
                    getInstantiationStrategy().instantiate(mbd, beanName, parent),
                    getAccessControlContext());
        } else {
            // 获得 InstantiationStrategy 对象，并使用它，创建 Bean 对象
            beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);
        }
        // 封装 BeanWrapperImpl  并完成初始化
        BeanWrapper bw = new BeanWrapperImpl(beanInstance);
        initBeanWrapper(bw);
        return bw;
    } catch (Throwable ex) {
        throw new BeanCreationException(
                mbd.getResourceDescription(), beanName, "Instantiation of bean failed", ex);
    }
}
```

这个方法,相比于 #instantiateUsingFactoryMethod(...) 、 #autowireConstructor(...) 方法，实在是太简单了，因为，它没有参数，所以不需要确认经过复杂的过来来确定构造器、构造参数，所以这里就不过多阐述了。

2.1 instantiate

```
// SimpleInstantiationStrategy.java

@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);
        }
    }
```

3. 小结

对于 #createBeanInstance(...) 方法而言，他就是选择合适实例化策略来为 bean 创建实例对象，具体的策略有：

Supplier 回调方式
工厂方法初始化
构造函数自动注入初始化
默认构造函数注入。

其中，工厂方法初始化和构造函数自动注入初始化两种方式最为复杂，主要是因为构造函数和构造参数的不确定性，Spring 需要花大量的精力来确定构造函数和构造参数，如果确定了则好办，直接选择实例化策略即可。

当然，在实例化的时候会根据是否有需要覆盖或者动态替换掉的方法，因为存在覆盖或者织入的话需要创建动态代理将方法织入，这个时候就只能选择 CGLIB 的方式来实例化，否则直接利用反射的方式即可，方便快捷。

到这里 #createBeanInstance(...) 的过程就已经分析完毕了，下篇介绍 #doCreateBean(...) 方法中的第二个过程：循环依赖的处理。其实，在整个 bean 的加载过程中都涉及到了循环依赖的处理，所以下

面分析并不是仅仅只针对 `#doCreateBean(...)` 方法中的循环依赖处理，而是 Spring 在整个 IoC 体系中是如何解决循环依赖的。

文章目录

回到首页