



当前位置: Java 技术驿站 (<http://cmsblogs.com>) > 死磕Java (<http://cmsblogs.com/?cat=189>) > 死磕 Spring (<http://cmsblogs.com/?cat=206>) > 正文

【死磕 Spring】—— IOC 之构造函数实例化 bean (<http://cmsblogs.com/?p=2850>)

2018-10-25 分类: 死磕 Spring (<http://cmsblogs.com/?cat=206>) 阅读(6850) 评论(0)

原文出自: <http://cmsblogs.com> (<http://cmsblogs.com>)

`createBeanInstance()` 用于实例化 bean, 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()`
- 工厂方法初始化: `instantiateUsingFactoryMethod()`
- 构造函数自动注入初始化: `autowireConstructor()`
- 默认构造函数注入: `instantiateBean()`

在上篇博客(【死磕 Spring】----- IOC 之 Factory 实例化 bean (<http://cmsblogs.com/?p=2848>)) 中分析了 Supplier 回调和工厂方法初始化, 这篇分析两个构造函数注入。

autowireConstructor()

这个初始化方法我们可以简单理解为是带有参数的初始化 bean 。代码段如下:



public BeanWrapper autowireConstructor(final String beanName, final RootBeanDefinition mbd,
 @Nullable Constructor<?>[] chosenCtors, @Nullable final Object



```

[] explicitArgs) {
    // 封装 BeanWrapperImpl 并完成初始化
    BeanWrapperImpl bw = new BeanWrapperImpl();
    this.beanFactory.initBeanWrapper(bw);

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

    /*
     * 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) {
                // 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);
        }
    }

    /*
     * 没有缓存, 则尝试从配置文件中获取
     */
    if (constructorToUse == null) {
        // 是否需要解析构造器
    }
  }
}

```



```

boolean autowiring = (chosenCtors != null ||
    mbd.getResolvedAutowireMode() == RootBeanDefinition.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);
}

/*
 * 获取指定的构造函数
 */
// 根据前面的判断，chosenCtors 应该为 null
Constructor<?>[] candidates = chosenCtors;
if (candidates == null) {
    // 获取 bean 的 class
    Class<?> beanClass = mbd.getBeanClass();
    try {
        // 根据 class 获取所有的构造函数
        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
        );
    }
}

// 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) {
    break;
}
// 参数个数不等，继续
if (paramTypes.length < minNrOfArgs) {
    continue;
}

// 参数持有者
ArgumentsHolder argsHolder;
// 有参数
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
        argsHolder = createArgumentArray(beanName, mbd, resolvedValues, bw, paramTypes, p
aramNames,

            getUserDeclaredConstructor(candidate), autowiring);
    }
    catch (UnsatisfiedDependencyException ex) {
        if (this.beanFactory.logger.isTraceEnabled()) {
            this.beanFactory.logger.trace(
                "Ignoring constructor [" + candidate + "] of bean '" + beanName + "':
" + ex);
        }
        // Swallow and try next constructor.
        if (causes == null) {
            causes = new LinkedList<>();
        }
        causes.add(ex);
        continue;
    }
}
else {
    // 构造函数没有参数
    if (paramTypes.length != explicitArgs.length) {

```



```

        continue;
    }
    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));

// 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);
}

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);
}

// 将构造函数、构造参数保存到缓存中

```



```

        if (explicitArgs == null) {
            argsHolderToUse.storeCache(mbd, constructorToUse);
        }
    }

    try {
        // 获取创建 bean 的策略
        final InstantiationStrategy strategy = beanFactory.getInstantiationStrategy();
        Object beanInstance;

        if (System.getSecurityManager() != null) {
            final Constructor<?> ctorToUse = constructorToUse;
            final Object[] argumentsToUse = argsToUse;
            // 实例化 bean
            beanInstance = AccessController.doPrivileged((PrivilegedAction<Object>) () ->
                strategy.instantiate(mbd, beanName, beanFactory, ctorToUse, argumentsToUs
e),
                beanFactory.getAccessControlContext());
        }
        else {
            // 实例化bean
            beanInstance = strategy.instantiate(mbd, beanName, this.beanFactory, constructorToUse, ar
gsToUse);
        }

        // 将构造的 bean 加入到 BeanWrapper 实例中
        bw.setBeanInstance(beanInstance);
        return bw;
    }
    catch (Throwable ex) {
        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
            "Bean instantiation via constructor failed", ex);
    }
}

```

代码与 `instantiateUsingFactoryMethod()` 一样，又长又难懂，但是如果理解了 `instantiateUsingFactoryMethod()` 初始化 bean 的过程，那么 `autowireConstructor()` 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()` 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 Factory 实例化 bean (<http://cmsblogs.com/?p=2848>)，这里我们重点分析初始化策略。对于初始化策略，首先是获取实例化 bean 的策略，如下：

```
final InstantiationStrategy strategy = beanFactory.getInstantiationStrategy();
```

然后是调用其 `instantiate()` 方法，该方法在 `SimpleInstantiationStrategy` 中实现，如下：






```

public Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner) {
    // 如果beandefinition没有override属性
    // 直接使用反射实例化即可
    if (!bd.hasMethodOverrides()) {
        // 重新检测获取下构造函数
        // 该构造函数是经过前面 N 多复杂过程确认的构造函数
        Constructor<?> constructorToUse;
        synchronized (bd.constructorArgumentLock) {
            // 获取已经解析的构造函数
            constructorToUse = (Constructor<?>) bd.resolvedConstructorOrFactoryMethod;
            // 如果为 null, 从 class 中解析获取, 并设置
            if (constructorToUse == null) {
                final Class<?> clazz = bd.getBeanClass();
                if (clazz.isInterface()) {
                    throw new BeanInstantiationException(clazz, "Specified class is an interface");
                }
                try {
                    if (System.getSecurityManager() != null) {
                        constructorToUse = AccessController.doPrivileged(
                            (PrivilegedExceptionAction<Constructor<?>>) clazz::getDeclaredConstructor);
                    }
                    else {
                        constructorToUse = clazz.getDeclaredConstructor();
                    }
                    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
        return BeanUtils.instantiateClass(constructorToUse);
    }
    else {
        // 生成CGLIB创建的子类对象, 如果配置了override则cglib
        return instantiateWithMethodInjection(bd, beanName, owner);
    }
}

```

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 即可, 方便快捷, 但是如果存在需要覆盖的方法或者动态替换的方法则需要使用 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。反射调用工具类 BeanUtils 的 instantiateClass() 方法完成反射工作:

```

public static <T> T instantiateClass(Constructor<T> ctor, Object... args) throws BeanInstantiationException {
    Assert.notNull(ctor, "Constructor must not be null");
    try {
        ReflectionUtils.makeAccessible(ctor);
        return (KotlinDetector.isKotlinType(ctor.getDeclaringClass()) ?
            KotlinDelegate.instantiateClass(ctor, args) : ctor.newInstance(args));
    }
    // 省略一些 catch
}

```

CGLIB

```

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 CglibSubclassingInstantiationStrategy 实现：

```

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

protected Object instantiateWithMethodInjection(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner,
    @Nullable Constructor<?> ctor, @Nullable Object... args) {

    // 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，调用其 instantiate() 方法生成其子类对象：



```

public Object instantiate(@Nullable Constructor<?> ctor, @Nullable 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()
                + "]", ex);
        }
    }

    // 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 CglibSubclassingInstantiationStrategy.LookupOverrideMethodInterceptor(this.beanDefinition, this.owner),
        new CglibSubclassingInstantiationStrategy.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。

instantiateBean()



```
protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefinition mbd) {  
    try {  
        Object beanInstance;  
        final BeanFactory parent = this;  
        if (System.getSecurityManager() != null) {  
            beanInstance = AccessController.doPrivileged((PrivilegedAction<Object>) () ->  
                getInstantiationStrategy().instantiate(mbd, beanName, parent),  
                getAccessControlContext());  
        }  
        else {  
            beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);  
        }  
        BeanWrapper bw = new BeanWrapperImpl(beanInstance);  
        initBeanWrapper(bw);  
        return bw;  
    }  
    catch (Throwable ex) {  
        throw new BeanCreationException(  
            mbd.getResourceDescription(), beanName, "Instantiation of bean failed", ex);  
    }  
}
```



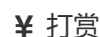
这个方法相比于 `instantiateUsingFactoryMethod()`、`autowireConstructor()` 方法实在是太简单了，因为它没有参数，所以不需要确认经过复杂的过来来确定构造器、构造参数，所以这里就不过多阐述了。对于 `createBeanInstance()` 而言，他就是选择合适实例化策略来为 bean 创建实例对象，具体的策略有：**Supplier 回调方式、工厂方法初始化、构造函数自动注入初始化、默认构造函数注入**。其中工厂方法初始化和构造函数自动注入初始化两种方式最为复杂，主要是因为构造函数和构造参数的不确定性，Spring 需要花大量的精力来确定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：属性填充。

更多阅读

- 【死磕 Spring】----- IOC 之开启 bean 的实例化进程 (<http://cmsblogs.com/?p=2846>)
- 【死磕 Spring】----- IOC 之 Factory 实例化 bean (<http://cmsblogs.com/?p=2848>)



赞(7)



打赏

【公告】版权声明 (http://cmsblogs.com/?page_id=1908)

标签： Spring源码解析 (<http://cmsblogs.com/?tag=spring%e6%ba%90%e7%a0%81%e8%a7%a3%e6%9e%90>)

死磕Java (<http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95java>)

死磕Spring (<http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95spring>)