



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

【死磕 Spring】—— IOC 之 Factory 实例化 bean (<http://cmsblogs.com/?p=2848>)

2018-10-25 分类: [死磕 Spring \(http://cmsblogs.com/?cat=206\)](http://cmsblogs.com/?cat=206) 阅读(7877) 评论(6)

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

这篇我们关注创建 bean 过程中的第一个步骤: 实例化 bean, 对应的方法为: `createBeanInstance()`, 如下:

```

protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {
    // 解析 bean, 将 bean 类名解析为 class 引用
    Class<?> beanClass = resolveBeanClass(mbd, beanName);

    if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed()) {
        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
            "Bean class isn't public, and non-public access not allowed: " + beanClass.getName());
    }

    // 如果存在 Supplier 回调, 则使用给定的回调方法初始化策略
    Supplier<?> instanceSupplier = mbd.getInstanceSupplier();
    if (instanceSupplier != null) {
        return obtainFromSupplier(instanceSupplier, beanName);
    }

    // 如果工厂方法不为空, 则使用工厂方法初始化策略
    if (mbd.getFactoryMethodName() != null) {
        return instantiateUsingFactoryMethod(beanName, mbd, args);
    }

    boolean resolved = false;
    boolean autowireNecessary = false;
    if (args == null) {
        // constructorArgumentLock 构造函数的常用锁
        synchronized (mbd.constructorArgumentLock) {
            // 如果已缓存的解析的构造函数或者工厂方法不为空, 则可以利用构造函数解析
            // 因为需要根据参数确认到底使用哪个构造函数, 该过程比较消耗性能, 所有采用缓存机制
            if (mbd.resolvedConstructorOrFactoryMethod != null) {
                resolved = true;
                autowireNecessary = mbd.constructorArgumentsResolved;
            }
        }
    }

    // 已经解析好了, 直接注入即可
    if (resolved) {
        // 自动注入, 调用构造函数自动注入
        if (autowireNecessary) {
            return autowireConstructor(beanName, mbd, null, null);
        }
        else {
            // 使用默认构造函数构造
            return instantiateBean(beanName, mbd);
        }
    }

    // 确定解析的构造函数
    // 主要是检查已经注册的 SmartInstantiationAwareBeanPostProcessor
    Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);

```



```

if (ctors != null ||
    mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_CONSTRUCTOR ||
    mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {
    // 构造函数自动注入
    return autowireConstructor(beanName, mbd, ctors, args);
}

//使用默认构造函数注入
return instantiateBean(beanName, mbd);
}

```

实例化 bean 是一个复杂的过程，其主要的逻辑为：

- 如果存在 Supplier 回调，则调用 obtainFromSupplier() 进行初始化
- 如果存在工厂方法，则使用工厂方法进行初始化
- 首先判断缓存，如果缓存中存在，即已经解析过了，则直接使用已经解析了的，根据 constructorArgumentsResolved 参数来判断是使用构造函数自动注入还是默认构造函数
- 如果缓存中没有，则需要先确定到底使用哪个构造函数来完成解析工作，因为一个类有多个构造函数，每个构造函数都有不同的构造参数，所以需要根据参数来锁定构造函数并完成初始化，如果存在参数则使用相应的带有参数的构造函数，否则使用默认构造函数。

下面就上面四种情况做分别说明。

obtainFromSupplier()

```

Supplier<?> instanceSupplier = mbd.getInstanceSupplier();
if (instanceSupplier != null) {
    return obtainFromSupplier(instanceSupplier, beanName);
}

```

首先从 BeanDefinition 中获取 Supplier，如果不为空，则调用 obtainFromSupplier()。那么 Supplier 是什么呢？在这之前也没有提到过这个字段。

```

public interface Supplier<T> {
    T get();
}

```


Supplier 接口仅有一个功能性的 get()，该方法会返回一个 T 类型的对象，有点儿类似工厂方法。这个接口有什么作用？用于指定创建 bean 的回调，如果我们设置了这样的回调，那么其他的构造器或者工厂方法都没有用。在什么设置该参数呢？Spring 提供了相应的 setter 方法，如下：

```

public void setInstanceSupplier(@Nullable Supplier<?> instanceSupplier) {
    this.instanceSupplier = instanceSupplier;
}

```



在构造 BeanDefinition 的时候设置了该值，如下（以 RootBeanDefinition 为例）：



```

public <T> RootBeanDefinition(@Nullable Class<T> beanClass, String scope, @Nullable Supplier<T> instanceSupplier) {
    super();
    setBeanClass(beanClass);
    setScope(scope);
    setInstanceSupplier(instanceSupplier);
}

```

如果设置了 instanceSupplier 则调用 obtainFromSupplier() 完成 bean 的初始化，如下：

```

protected BeanWrapper obtainFromSupplier(Supplier<?> instanceSupplier, String beanName) {
    String outerBean = this.currentlyCreatedBean.get();
    this.currentlyCreatedBean.set(beanName);
    Object instance;
    try {
        // 调用 Supplier 的 get(), 返回一个对象
        instance = instanceSupplier.get();
    }
    finally {
        if (outerBean != null) {
            this.currentlyCreatedBean.set(outerBean);
        }
        else {
            this.currentlyCreatedBean.remove();
        }
    }
    // 根据对象构造 BeanWrapper 对象
    BeanWrapper bw = new BeanWrapperImpl(instance);
    // 初始化 BeanWrapper
    initBeanWrapper(bw);
    return bw;
}

```

代码很简单，调用 Supplier 的 get() 方法，获得一个 bean 实例对象，然后根据该实例对象构造一个 BeanWrapper 对象 bw，最后初始化该对象。有关于 BeanWrapper 后面专门出文讲解。

instantiateUsingFactoryMethod()

如果存在工厂方法，则调用 instantiateUsingFactoryMethod() 完成 bean 的初始化工作（方法实现比较长，细节比较复杂，各位就硬着头皮看吧）。

```

protected BeanWrapper instantiateUsingFactoryMethod(
    String beanName, RootBeanDefinition mbd, @Nullable Object[] explicitArgs) {
    return new ConstructorResolver(this).instantiateUsingFactoryMethod(beanName, mbd, explicitArgs);
}

```

构造一个 ConstructorResolver 对象，然后调用其 instantiateUsingFactoryMethod() 方法。ConstructorResolver 是构造方法或者工厂类初始化 bean 的委托类。



```

public BeanWrapper instantiateUsingFactoryMethod(
    final String beanName, final RootBeanDefinition mbd, @Nullable final Object[] explicitArgs) {

    // 构造 BeanWrapperImpl 对象
    BeanWrapperImpl bw = new BeanWrapperImpl();
    // 初始化 BeanWrapperImpl
    // 向BeanWrapper对象中添加 ConversionService 对象和属性编辑器 PropertyEditor 对象
    //
    this.beanFactory.initBeanWrapper(bw);

    Object factoryBean;
    Class<?> factoryClass;
    boolean isStatic;

    // 工厂名不为空
    String factoryBeanName = mbd.getFactoryBeanName();
    if (factoryBeanName != null) {
        if (factoryBeanName.equals(beanName)) {
            throw new BeanDefinitionStoreException(mbd.getResourceDescription(), beanName,
                "factory-bean reference points back to the same bean definition");
        }
        // 获取工厂实例
        factoryBean = this.beanFactory.getBean(factoryBeanName);
        if (mbd.isSingleton() && this.beanFactory.containsSingleton(beanName)) {
            throw new ImplicitlyAppearedSingletonException();
        }
        factoryClass = factoryBean.getClass();
        isStatic = false;
    }
    else {
        // 工厂名为空，则其可能是一个静态工厂

        // 静态工厂创建bean，必须要提供工厂的全类名
        if (!mbd.hasBeanClass()) {
            throw new BeanDefinitionStoreException(mbd.getResourceDescription(), beanName,
                "bean definition declares neither a bean class nor a factory-bean reference");
        }
        factoryBean = null;
        factoryClass = mbd.getBeanClass();
        isStatic = true;
    }

    // 工厂方法
    Method factoryMethodToUse = null;
    ConstructorResolver.ArgumentsHolder argsHolderToUse = null;
    // 参数
    Object[] argsToUse = null;

    // 工厂方法的参数
    // 如果指定了构造参数则直接使用
    // 在调用 getBean 方法的时候指定了方法参数

```



```

if (explicitArgs != null) {
    argsToUse = explicitArgs;
}
else {
    // 没有指定, 则尝试从配置文件中解析
    Object[] argsToResolve = null;
    // 首先尝试从缓存中获取
    synchronized (mbd.constructorArgumentLock) {
        // 获取缓存中的构造函数或者工厂方法
        factoryMethodToUse = (Method) mbd.resolvedConstructorOrFactoryMethod;
        if (factoryMethodToUse != null && mbd.constructorArgumentsResolved) {
            // 获取缓存中的构造参数
            argsToUse = mbd.resolvedConstructorArguments;
            if (argsToUse == null) {
                // 获取缓存中的构造函数参数的包可见字段
                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, factoryMethodToUse, argsToResolve
    );
    }
}

//
if (factoryMethodToUse == null || argsToUse == null) {
    // 获取工厂方法的类全名称
    factoryClass = ClassUtils.getUserClass(factoryClass);

    // 获取所有待定方法
    Method[] rawCandidates = getCandidateMethods(factoryClass, mbd);
    // 检索所有方法, 这里是对方法进行过滤
    List<Method> candidateSet = new ArrayList<>();
    for (Method candidate : rawCandidates) {
        // 如果有static 且为工厂方法, 则添加到 candidateSet 中
        if (Modifier.isStatic(candidate.getModifiers()) == isStatic && mbd.isFactoryMethod(candidate)) {
            candidateSet.add(candidate);
        }
    }

    Method[] candidates = candidateSet.toArray(new Method[0]);
    // 排序构造函数
    // 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.sortFactoryMethods(candidates);

    // 用于承载解析后的构造函数参数的值
    ConstructorArgumentValues resolvedValues = null;

```



```

boolean autowiring = (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_CONSTRUCTO

int minTypeDiffWeight = Integer.MAX_VALUE;
Set<Method> ambiguousFactoryMethods = null;

int minNrOfArgs;
if (explicitArgs != null) {
    minNrOfArgs = explicitArgs.length;
}
else {
    // getBean() 没有传递参数，则需要解析保存在 BeanDefinition 构造函数中指定的参数
    if (mbd.hasConstructorArgumentValues()) {
        // 构造函数的参数
        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);
    }
    else {
        minNrOfArgs = 0;
    }
}

LinkedList<UnsatisfiedDependencyException> causes = null;

for (Method candidate : candidates) {
    // 方法体的参数
    Class<?>[] paramTypes = candidate.getParameterTypes();

    if (paramTypes.length >= minNrOfArgs) {
        // 保存参数的对象
        ArgumentsHolder argsHolder;

        // getBean()传递了参数
        if (explicitArgs != null){
            // 显示给定参数，参数长度必须完全匹配
            if (paramTypes.length != explicitArgs.length) {
                continue;
            }
            // 根据参数创建参数持有者
            argsHolder = new ArgumentsHolder(explicitArgs);
        }
        else {
            // 为提供参数，解析构造参数
            try {
                String[] paramNames = null;
                // 获取 ParameterNameDiscoverer 对象
                // ParameterNameDiscoverer 是用于解析方法和构造函数的参数名称的接口，为参数名称探

                ParameterNameDiscoverer pnd = this.beanFactory.getParameterNameDiscoverer();
                if (pnd != null) {

```

测器



```

// 获取指定构造函数的参数名称
paramNames = pnd.getParameterNames(candidate);
}

// 在已经解析的构造函数参数值的情况下，创建一个参数持有者对象
argsHolder = createArgumentArray(
    beanName, mbd, resolvedValues, bw, paramTypes, paramNames, candidate,
    autowiring);
}
catch (UnsatisfiedDependencyException ex) {
    if (this.beanFactory.logger.isTraceEnabled()) {
        this.beanFactory.logger.trace("Ignoring factory method [" + candidate +
            "] of bean '" + beanName + "': " + ex);
    }
    if (causes == null) {
        causes = new LinkedList<>();
    }
    causes.add(ex);
    continue;
}
}

// 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.getAssignabilityW
eight(paramTypes));

// 代表最接近的类型匹配，则选择作为构造函数
if (typeDiffWeight < minTypeDiffWeight) {
    factoryMethodToUse = candidate;
    argsHolderToUse = argsHolder;
    argsToUse = argsHolder.arguments;
    minTypeDiffWeight = typeDiffWeight;
    ambiguousFactoryMethods = null;
}

// 如果具有相同参数数量的方法具有相同的类型差异权重，则收集此类型选项
// 但是，仅在非宽松构造函数解析模式下执行该检查，并显式忽略重写方法（具有相同的参数签名）
else if (factoryMethodToUse != null && typeDiffWeight == minTypeDiffWeight &&
    !mbd.isLenientConstructorResolution() &&
    paramTypes.length == factoryMethodToUse.getParameterCount() &&
    !Arrays.equals(paramTypes, factoryMethodToUse.getParameterTypes())) {

    // 查找到多个可匹配的方法
    if (ambiguousFactoryMethods == null) {
        ambiguousFactoryMethods = new LinkedHashSet<>();
        ambiguousFactoryMethods.add(factoryMethodToUse);
    }
    ambiguousFactoryMethods.add(candidate);
}

```





```

    }
}

// 没有可执行的工厂方法，抛出异常
if (factoryMethodToUse == null) {
    if (causes != null) {
        UnsatisfiedDependencyException ex = causes.removeLast();
        for (Exception cause : causes) {
            this.beanFactory.onSuppressedException(cause);
        }
        throw ex;
    }
    List<String> argTypes = new ArrayList<>(minNrOfArgs);
    if (explicitArgs != null) {
        for (Object arg : explicitArgs) {
            argTypes.add(arg != null ? arg.getClass().getSimpleName() : "null");
        }
    }
    else if (resolvedValues != null){
        Set<ConstructorArgumentValues.ValueHolder> valueHolders = new LinkedHashSet<>(resolvedValues.getArgumentCount());
        valueHolders.addAll(resolvedValues.getIndexedArgumentValues().values());
        valueHolders.addAll(resolvedValues.getGenericArgumentValues());
        for (ConstructorArgumentValues.ValueHolder value : valueHolders) {
            String argType = (value.getType() != null ? ClassUtils.getShortName(value.getType()) :
                (value.getValue() != null ? value.getValue().getClass().getSimpleName() :
                    "null"));
            argTypes.add(argType);
        }
    }
    String argDesc = StringUtils.collectionToCommaDelimitedString(argTypes);
    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
        "No matching factory method found: " +
            (mbd.getFactoryBeanName() != null ?
                "factory bean '" + mbd.getFactoryBeanName() + "'; " : "") +
            "factory method '" + mbd.getFactoryMethodName() + "(" + argDesc + ")'. " +
            +
            "Check that a method with the specified name " +
            (minNrOfArgs > 0 ? "and arguments " : "") +
            "exists and that it is " +
            (isStatic ? "static" : "non-static") + "." );
}
else if (void.class == factoryMethodToUse.getReturnType()) {
    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
        "Invalid factory method '" + mbd.getFactoryMethodName() +
            "': needs to have a non-void return type!");
}
else if (ambiguousFactoryMethods != null) {
    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
        "Ambiguous factory method matches found in bean '" + beanName + "' " +

```

 type ambiguities): " +

 Java技术驿站


```

        ambiguousFactoryMethods);
    }

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

try {
    // 实例化 bean
    Object beanInstance;

    if (System.getSecurityManager() != null) {
        final Object fb = factoryBean;
        final Method factoryMethod = factoryMethodToUse;
        final Object[] args = argsToUse;
        // 通过执行工厂方法来创建bean示例
        beanInstance = AccessController.doPrivileged((PrivilegedAction<Object>) () ->
            beanFactory.getInstantiationStrategy().instantiate(mbd, beanName, beanFac
tory, fb, factoryMethod, args),
            beanFactory.getAccessControlContext());
    }
    else {
        // 通过执行工厂方法来创建bean示例
        beanInstance = this.beanFactory.getInstantiationStrategy().instantiate(
            mbd, beanName, this.beanFactory, factoryBean, factoryMethodToUse, argsToUse);
    }

    // 包装为 BeanWraper 对象
    bw.setBeanInstance(beanInstance);
    return bw;
}
catch (Throwable ex) {
    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
        "Bean instantiation via factory method failed", ex);
}
}

```

instantiateUsingFactoryMethod() 方法体实在是太大了，处理细节感觉很复杂，LZ是硬着头皮看完的，中间断断续续的。吐槽这里的代码风格，完全不符合我们前面看的 Spring 代码风格。Spring 的一贯做法是将一个复杂逻辑进行拆分，分为多个细小的模块进行嵌套，每个模块负责一部分功能，模块与模块之间层层嵌套，上一层一般都是对下一层的总结和概括，这样就会使得每一层的逻辑变得清晰易懂。回归到上面的方法体，虽然代码体量大，但是总体我们还是可看清楚这个方法要做的事情。一句话概括就是：确定工厂对象，然后获取构造函数和构造参数，最后调用 InstantiationStrategy 对象的 instantiate() 来创建 bean 实例。下面我们就这个句概括的话进行拆分并详细说明。 **确定工厂对象** 首先获取工厂方法名，若工厂方法名不为空，则调用 beanFactory.getBean() 获取工厂对象，若为空，则可能为一个静态工厂，对于静态工厂则必须提供工厂类的全类名，同时设置 factoryBean = null **构造参数确认** 工厂对象确定后，则是确认构造参数。构造参数的确认

主要分为三种情况：explicitArgs 参数、缓存中获取、配置文件中解析。**explicitArgs 参数** explicitArgs 参数是我们调用 `getBean()` 时传递来的，一般该参数，该参数就是用于初始化 bean 时所传递的参数，如果该参数不为空，则可以确定构造函数的参数就是它了。**缓存中获取** 在该方法的最后，我们会发现这样一段代码：`argsHolderToUse.storeCache(mbd, factoryMethodToUse)`，这段代码主要是将构造函数、构造参数保存到缓存中，如下：

```
public void storeCache(RootBeanDefinition mbd, Executable constructorOrFactoryMethod) {
    synchronized (mbd.constructorArgumentLock) {
        mbd.resolvedConstructorOrFactoryMethod = constructorOrFactoryMethod;
        mbd.constructorArgumentsResolved = true;
        if (this.resolveNecessary) {
            mbd.preparedConstructorArguments = this.preparedArguments;
        }
        else {
            mbd.resolvedConstructorArguments = this.arguments;
        }
    }
}
```

其中涉及到的几个参数 `constructorArgumentLock`、`resolvedConstructorOrFactoryMethod`、`constructorArgumentsResolved`、`resolvedConstructorArguments`。这些参数都是跟构造函数、构造函数缓存有关的。

- `constructorArgumentLock`：构造函数的缓存锁
- `resolvedConstructorOrFactoryMethod`：缓存已经解析的构造函数或者工厂方法
- `constructorArgumentsResolved`：标记字段，标记构造函数、参数已经解析了。默认为false
- `resolvedConstructorArguments`：缓存已经解析的构造函数参数，包可见字段

所以从缓存中获取就是提取这几个参数的值，如下：

```
synchronized (mbd.constructorArgumentLock) {
    // 获取缓存中的构造函数或者工厂方法
    factoryMethodToUse = (Method) mbd.resolvedConstructorOrFactoryMethod;
    if (factoryMethodToUse != null && mbd.constructorArgumentsResolved) {
        // 获取缓存中的构造参数
        argsToUse = mbd.resolvedConstructorArguments;
        if (argsToUse == null) {
            // 获取缓存中的构造函数参数的包可见字段
            argsToResolve = mbd.preparedConstructorArguments;
        }
    }
}
```

如果缓存中存在构造参数，则需要调用 `resolvePreparedArguments()` 方法进行转换，因为缓存中的值有可能是最终值也有可能不是最终值，比如我们构造函数中的类型为 `Integer` 类型的 `1`，但是原始的参数类型有可能是 `String` 类型的 `1`，所以即便是从缓存中得到了构造参数也需要经过一番的类型转换确保参数类型完全对应。**配置文件中解析** 即没有通过传递参数的方式传递构造参数，缓存中也没有，那就只能通过解析配置文件获

取构造参数了。在 bean 解析类的博文中我们了解了，配置文件中的信息都会转换到 BeanDefinition 实例对象中，所以配置文件中的参数可以直接通过 BeanDefinition 对象获取。代码如下：

```
if (mbd.hasConstructorArgumentValues()) {  
    // 构造函数的参数  
    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);  
}
```

通过 BeanDefinition 的 getConstructorArgumentValues() 就可以获取构造信息了，有了构造信息就可以获取相关的参数值信息了，获取的参数信息包括直接值和引用，这一步骤的处理交由 resolveConstructorArguments() 完成，该方法会将构造参数信息解析为 resolvedValues 对象并返回解析到的参数个数。**构造函数** 确定构造参数后，下一步则是确定构造函数。第一步则是通过 getCandidateMethods() 获取所有的构造方法，同时对构造方法进行刷选，然后在对其进行排序处理 (AutowireUtils.sortFactoryMethods(candidates))，排序的主要目的是为了能够更加方便的找到匹配的构造函数，因为构造函数的确认是根据参数个数确认的。排序的规则是：public 构造函数优先参数数量降序、非 public 构造参数数量降序。通过迭代 candidates (包含了所有要匹配的构造函数) 的方式，一次比较其参数，如果显示提供了参数 (explicitArgs != null)，则直接比较两者是否相等，如果相等则表示找到了，否则继续比较。如果没有显示提供参数，则需要获取 ParameterNameDiscoverer 对象，该对象为参数名称探测器，主要用于发现方法和构造函数的参数名称。将参数包装成 ArgumentsHolder 对象，该对象用于保存参数，我们称之为参数持有者。当将对象包装成 ArgumentsHolder 对象后，我们就可以通过它来进行构造函数匹配，匹配分为严格模式和宽松模式。

- 严格模式：解析构造函数时，必须所有参数都需要匹配，否则抛出异常
- 宽松模式：使用具有"最接近的模式"进行匹配

判断的依据是根据 BeanDefinition 的 isLenientConstructorResolution 属性 (该参数是我们在构造 AbstractBeanDefinition 对象是传递的) 来获取类型差异权重 (typeDiffWeight) 的。如果 typeDiffWeight < minTypeDiffWeight，则代表 "最接近的模式"，选择其作为构造函数，否则只有两者具有相同的参数数量且类型差异权重相等才会纳入考虑范围。至此，构造函数已经确认了。**创建 bean 实例** 工厂对象、构造函数、构造参数都已经确认了，则最后一步就是调用 InstantiationStrategy 对象的 instantiate() 来创建 bean 实例，如下：

```

public Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFactory owner,
    @Nullable Object factoryBean, final Method factoryMethod, @Nullable Object... args) {

    try {
        if (System.getSecurityManager() != null) {
            AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
                ReflectionUtils.makeAccessible(factoryMethod);
                return null;
            });
        }
        else {
            ReflectionUtils.makeAccessible(factoryMethod);
        }

        Method priorInvokedFactoryMethod = currentlyInvokedFactoryMethod.get();
        try {
            currentlyInvokedFactoryMethod.set(factoryMethod);
            // 执行工厂方法，并返回实例
            Object result = factoryMethod.invoke(factoryBean, args);
            if (result == null) {
                result = new NullBean();
            }
            return result;
        }
        finally {
            if (priorInvokedFactoryMethod != null) {
                currentlyInvokedFactoryMethod.set(priorInvokedFactoryMethod);
            }
            else {
                currentlyInvokedFactoryMethod.remove();
            }
        }
    }
    // 省略一波 catch
}

```

instantiate() 最核心的部分就是利用 Java 反射执行工厂方法并返回创建好的实例，也就是这段代码：

```
Object result = factoryMethod.invoke(factoryBean, args);
```

到这里 instantiateUsingFactoryMethod() 已经分析完毕了，这里 LZ 有些题外话需要说下，看源码真心是一个痛苦的过程，尤其是复杂的源码，比如这个方法我看了三天才弄清楚点皮毛，当然这里跟 LZ 的智商有些关系（智商捉急 T_T），写这篇博客也花了五天时间才写完（最后截稿日为：2018.08.10 01:23:49），所以每一个坚持写博客的都是折翼的天使，值得各位尊敬 createBeanInstance() 还有两个重要方法 autowireConstructor() 和 instantiateBean()，由于篇幅问题，所以将这两个方法放在下篇博客分析。敬请期待!!!

更多阅读

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