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
2019-06-10
Spring

【死磕 Spring】—— IoC 之加载 Bean：创建 Bean（四）之属性填充

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，「老芳芳」略作修改，记录在理解过程中，参考的资料。

`#doCreateBean(...)` 方法，主要用于完成 bean 的创建和初始化工作，我们可以将其分为四个过程：

- `#createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args)` 方法，实例化 bean 。
  循环依赖的处理。
- `#populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw)` 方法，进行属性填充。
- `#initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd)` 方法，初始化 Bean 。

第一个过程，实例化 bean 已经在前面两篇博客分析完毕了。
这篇博客开始，分析属性填充，也就是 `#populateBean(...)` 方法。该函数的作用是将 BeanDefinition 中的属性值赋值给 BeanWrapper 实例对象(对于 BeanWrapper ，我们后续专门写文分析)。

1. populateBean

```
// AbstractAutowireCapableBeanFactory.java

protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {
    // 没有实例化对象
    if (bw == null) {
        // 有属性，则抛出 BeanCreationException 异常
        if (mbd.hasPropertyValues()) {
            throw new BeanCreationException(
                    mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");
            // 没有属性，直接 return 返回
        } else {
            // Skip property population phase for null instance.
            return;
        }
```

```
}

// <1> 在设置属性之前给 InstantiationAwareBeanPostProcessors 最后一次改变 bean 的机会
// Give any InstantiationAwareBeanPostProcessors the opportunity to modify the
// state of the bean before properties are set. This can be used, for example,
// to support styles of field injection.
boolean continueWithPropertyPopulation = true;
if (!mbd.isSynthetic()  // bean 不是"合成"的，即未由应用程序本身定义
        && hasInstantiationAwareBeanPostProcessors()) { // 是否持有 InstantiationAwareBeanPostProcessor
    // 迭代所有的 BeanPostProcessors
    for (BeanPostProcessor bp : getBeanPostProcessors()) {
        if (bp instanceof InstantiationAwareBeanPostProcessor) { // 如果为 InstantiationAwareBeanPostProcessor
            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
            // 返回值为是否继续填充 bean
            // postProcessAfterInstantiation：如果应该在 bean上面设置属性则返回 true，否则返回 false
            // 一般情况下，应该是返回true 。
            // 返回 false 的话，将会阻止在此 Bean 实例上调用任何后续的 InstantiationAwareBeanPostProcessor 实例。
            if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
                continueWithPropertyPopulation = false;
                break;
            }
        }
    }
}
// 如果后续处理器发出停止填充命令，则终止后续操作
if (!continueWithPropertyPopulation) {
    return;
}

// bean 的属性值
PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null);

// <2> 自动注入
if (mbd.getResolvedAutowireMode() == AUTOWIRE_BY_NAME || mbd.getResolvedAutowireMode() == AUTOWIRE_BY_TYPE) {
    // 将 PropertyValues 封装成 MutablePropertyValues 对象
    // MutablePropertyValues 允许对属性进行简单的操作，并提供构造函数以支持Map的深度复制和构造。
    MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
    // Add property values based on autowire by name if applicable.
    // 根据名称自动注入
    if (mbd.getResolvedAutowireMode() == AUTOWIRE_BY_NAME) {
        autowireByName(beanName, mbd, bw, newPvs);
    }
    // Add property values based on autowire by type if applicable.
    // 根据类型自动注入
    if (mbd.getResolvedAutowireMode() == AUTOWIRE_BY_TYPE) {
        autowireByType(beanName, mbd, bw, newPvs);
    }
    pvs = newPvs;
}

// 是否已经注册了 InstantiationAwareBeanPostProcessors
boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
// 是否需要进行【依赖检查】
boolean needsDepCheck = (mbd.getDependencyCheck() != AbstractBeanDefinition.DEPENDENCY_CHECK_NONE);

// <3> BeanPostProcessor 处理
PropertyDescriptor[] filteredPds = null;
if (hasInstAwareBpps) {
    if (pvs == null) {
        pvs = mbd.getPropertyValues();
```

```
            }
            // 遍历 BeanPostProcessor 数组
            for (BeanPostProcessor bp : getBeanPostProcessors()) {
                if (bp instanceof InstantiationAwareBeanPostProcessor) {
                    InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
                    // 对所有需要依赖检查的属性进行后处理
                    PropertyValues pvsToUse = ibp.postProcessProperties(pvs, bw.getWrappedInstance(), beanName);
                    if (pvsToUse == null) {
                        // 从 bw 对象中提取 PropertyDescriptor 结果集
                        // PropertyDescriptor：可以通过一对存取方法提取一个属性
                        if (filteredPds == null) {
                            filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
                        }
                        pvsToUse = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);
                        if (pvsToUse == null) {
                            return;
                        }
                    }
                    pvs = pvsToUse;
                }
            }
        }

        // <4> 依赖检查
        if (needsDepCheck) {
            if (filteredPds == null) {
                filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
            }
            // 依赖检查，对应 depends-on 属性
            checkDependencies(beanName, mbd, filteredPds, pvs);
        }

        // <5> 将属性应用到 bean 中
        if (pvs != null) {
            applyPropertyValues(beanName, mbd, bw, pvs);
        }
    }
```

处理流程如下：

<1> ，根据 hasInstantiationAwareBeanPostProcessors 属性来判断，是否需要在注入属性之前给 InstantiationAwareBeanPostProcessors 最后一次改变 bean 的机会。此过程可以控制 Spring 是否继续进行属性填充。
统一存入到 PropertyValues 中，PropertyValues 用于描述 bean 的属性。
- <2> ，根据注入类型（ AbstractBeanDefinition#getResolvedAutowireMode() 方法的返回值 ）的不同来判断：
  - 是根据名称来自动注入（#autowireByName(...)）
  - 还是根据类型来自动注入（#autowireByType(...)）
  - 详细解析，见 「1.1 自动注入」 。
- <3> ，进行 BeanPostProcessor 处理。
- <4> ，依赖检测。

<5> ，将所有 PropertyValues 中的属性，填充到 BeanWrapper 中。

1.1 自动注入

Spring 会根据注入类型（ byName / byType ）的不同，调用不同的方法来注入属性值。代码如下：

```
// AbstractBeanDefinition.java

/**
 * 注入模式
 */
private int autowireMode = AUTOWIRE_NO;

public int getResolvedAutowireMode() {
 if (this.autowireMode == AUTOWIRE_AUTODETECT) { // 自动检测模式，获得对应的检测模式
     // Work out whether to apply setter autowiring or constructor autowiring.
     // If it has a no-arg constructor it's deemed to be setter autowiring,
     // otherwise we'll try constructor autowiring.
        Constructor<?>[] constructors = getBeanClass().getConstructors();
     for (Constructor<?> constructor : constructors) {
         if (constructor.getParameterCount() == 0) {
             return AUTOWIRE_BY_TYPE;
            }
        }
     return AUTOWIRE_CONSTRUCTOR;
    } else {
     return this.autowireMode;
    }
}
```

1.1.1 autowireByName

#autowireByName(String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) 方法，是根据属性名称，完成自动依赖注入的。代码如下：

```
// AbstractAutowireCapableBeanFactory.java

protected void autowireByName(String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs)
    // <1> 对 Bean 对象中非简单属性
    String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
    // 遍历 propertyName 数组
    for (String propertyName : propertyNames) {
        // 如果容器中包含指定名称的 bean，则将该 bean 注入到 bean中
        if (containsBean(propertyName)) {
            // 递归初始化相关 bean
            Object bean = getBean(propertyName);
            // 为指定名称的属性赋予属性值
            pvs.add(propertyName, bean);
            // 属性依赖注入
            registerDependentBean(propertyName, beanName);
            if (logger.isTraceEnabled()) {
                logger.trace("Added autowiring by name from bean name '" + beanName +
                        "' via property '" + propertyName + "' to bean named '" + propertyName + "'");
            }
        } else {
            if (logger.isTraceEnabled()) {
                logger.trace("Not autowiring property '" + propertyName + "' of bean '" + beanName +
                        "' by name: no matching bean found");
            }
        }
    }
}
```

<1> 处，该方法逻辑很简单，获取该 bean 的非简单属性。什么叫做非简单属性呢？就是类型为对象类型的属性，但是这里并不是将所有的对象类型都都会找到，比如 8 个原始类型，String 类型 ，Number类型、Date类型、URL类型、URI类型等都会被忽略。代码如下：

```
// AbstractAutowireCapableBeanFactory.java

protected String[] unsatisfiedNonSimpleProperties(AbstractBeanDefinition mbd, BeanWrapper bw) {
    // 创建 result 集合
    Set<String> result = new TreeSet<>();
    PropertyValues pvs = mbd.getPropertyValues();
    // 遍历 PropertyDescriptor 数组
    PropertyDescriptor[] pds = bw.getPropertyDescriptors();
    for (PropertyDescriptor pd : pds) {
        if (pd.getWriteMethod() != null // 有可写方法
                && !isExcludedFromDependencyCheck(pd) // 依赖检测中没有被忽略
                && !pvs.contains(pd.getName()) // pvs 不包含该属性名
                && !BeanUtils.isSimpleProperty(pd.getPropertyType())) { // 不是简单属性类型
            result.add(pd.getName()); // 添加到 result 中
        }
    }
    return StringUtils.toStringArray(result);
}
```

- 过滤条件为：有可写方法、依赖检测中没有被忽略、不是简单属性类型。
- 过滤结果为：其实这里获取的就是需要依赖注入的属性。

获取需要依赖注入的属性后，通过迭代、递归的方式初始化相关的 bean ，然后调用 `#registerDependentBean(String beanName, String dependentBeanName)` 方法，完成注册依赖。代码如下：

```
// DefaultSingletonBeanRegistry.java

/**
 * Map between dependent bean names: bean name to Set of dependent bean names.
 *
 * 保存的是依赖 beanName 之间的映射关系：beanName - > 依赖 beanName 的集合
 */
private final Map<String, Set<String>> dependentBeanMap = new ConcurrentHashMap<>(64);

/**
 * Map between depending bean names: bean name to Set of bean names for the bean's dependencies.
 *
 * 保存的是依赖 beanName 之间的映射关系：依赖 beanName - > beanName 的集合
 */
private final Map<String, Set<String>> dependenciesForBeanMap = new ConcurrentHashMap<>(64);

public void registerDependentBean(String beanName, String dependentBeanName) {
    // 获取 beanName
    String canonicalName = canonicalName(beanName);
    // 添加 <canonicalName, <dependentBeanName>> 到 dependentBeanMap 中
    synchronized (this.dependentBeanMap) {
        Set<String> dependentBeans =
                this.dependentBeanMap.computeIfAbsent(canonicalName, k -> new LinkedHashSet<>(8));
        if (!dependentBeans.add(dependentBeanName)) {
            return;
        }
    }
    // 添加 <dependentBeanName, <canonicalName>> 到 dependenciesForBeanMap 中
```

```
            synchronized (this.dependenciesForBeanMap) {
                Set<String> dependenciesForBean =
                        this.dependenciesForBeanMap.computeIfAbsent(dependentBeanName, k -> new LinkedHashSet<>(8));
                dependenciesForBean.add(canonicalName);
            }
        }
```

1.1.2 autowireByType

#autowireByType(String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) 方法，是根据属性类型，完成自动依赖注入的。代码如下：

```
    // AbstractAutowireCapableBeanFactory.java

    protected void autowireByType(String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs)

        // 获取 TypeConverter 实例
        // 使用自定义的 TypeConverter，用于取代默认的 PropertyEditor 机制
        TypeConverter converter = getCustomTypeConverter();
        if (converter == null) {
            converter = bw;
        }

        Set<String> autowiredBeanNames = new LinkedHashSet<>(4);
        // 获取非简单属性
        String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
        // 遍历 propertyName 数组
        for (String propertyName : propertyNames) {
            try {
                // 获取 PropertyDescriptor 实例
                PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);
                // Don't try autowiring by type for type Object: never makes sense,
                // even if it technically is a unsatisfied, non-simple property.
                // 不要尝试按类型
                if (Object.class != pd.getPropertyType()) {
                    // 探测指定属性的 set 方法
                    MethodParameter methodParam = BeanUtils.getWriteMethodParameter(pd);
                    // Do not allow eager init for type matching in case of a prioritized post-processor.
                    boolean eager = !PriorityOrdered.class.isInstance(bw.getWrappedInstance());
                    DependencyDescriptor desc = new AutowireByTypeDependencyDescriptor(methodParam, eager);
                    // 解析指定 beanName 的属性所匹配的值，并把解析到的属性名称存储在 autowiredBeanNames 中
                    // 当属性存在过个封装 bean 时将会找到所有匹配的 bean 并将其注入
                    Object autowiredArgument = resolveDependency(desc, beanName, autowiredBeanNames, converter);
                    if (autowiredArgument != null) {
                        pvs.add(propertyName, autowiredArgument);
                    }
                    // 遍历 autowiredBeanName 数组
                    for (String autowiredBeanName : autowiredBeanNames) {
                        // 属性依赖注入
                        registerDependentBean(autowiredBeanName, beanName);
                        if (logger.isTraceEnabled()) {
                            logger.trace("Autowiring by type from bean name '" + beanName + "' via property '" +
                                    propertyName + "' to bean named '" + autowiredBeanName + "'");
                        }
                    }
                    // 清空 autowiredBeanName 数组
                    autowiredBeanNames.clear();
```

```
            }
        } catch (BeansException ex) {
            throw new UnsatisfiedDependencyException(mbd.getResourceDescription(), beanName, propertyName, ex);
        }
    }
}
```

其实主要过程和根据名称自动注入差不多，都是找到需要依赖注入的属性，然后通过迭代的方式寻找所匹配的 bean，最后调用 `#registerDependentBean(...)` 方法，来注册依赖。不过相对于 `#autowireByName(...)` 方法而言，根据类型寻找相匹配的 bean 过程比较复杂。

1.1.2.1 resolveDependency

下面我们就分析这个复杂的过程，代码如下：

```
// DefaultListableBeanFactory.java

@Nullable
private static Class<?> javaxInjectProviderClass;

static {
 try {
        javaxInjectProviderClass = ClassUtils.forName("javax.inject.Provider", DefaultListableBeanFactory.class.getCl
    } catch (ClassNotFoundException ex) {
     // JSR-330 API not available - Provider interface simply not supported then.
        javaxInjectProviderClass = null;
    }
}

@Override
@Nullable
public Object resolveDependency(DependencyDescriptor descriptor, @Nullable String requestingBeanName,
        @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeConverter) throws BeansException {
    // 初始化参数名称发现器，该方法并不会在这个时候尝试检索参数名称
    // getParameterNameDiscoverer 返回 parameterNameDiscoverer 实例，parameterNameDiscoverer 方法参数名称的解析器
    descriptor.initParameterNameDiscovery(getParameterNameDiscoverer());
    // 依赖类型为 Optional 类型
    if (Optional.class == descriptor.getDependencyType()) {
        return createOptionalDependency(descriptor, requestingBeanName);
    // 依赖类型为ObjectFactory、ObjectProvider
    } else if (ObjectFactory.class == descriptor.getDependencyType() ||
            ObjectProvider.class == descriptor.getDependencyType()) {
        return new DependencyObjectProvider(descriptor, requestingBeanName);
    // javaxInjectProviderClass 类注入的特殊处理
    } else if (javaxInjectProviderClass == descriptor.getDependencyType()) {
        return new Jsr330Factory().createDependencyProvider(descriptor, requestingBeanName);
    } else {
        // 为实际依赖关系目标的延迟解析构建代理
        // 默认实现返回 null
        Object result = getAutowireCandidateResolver().getLazyResolutionProxyIfNecessary(descriptor, requestingBeanNa
        if (result == null) {
            // 通用处理逻辑
            result = doResolveDependency(descriptor, requestingBeanName, autowiredBeanNames, typeConverter);
        }
        return result;
    }
}
```

这里我们关注通用处理逻辑 #doResolveDependency(DependencyDescriptor descriptor, @Nullable String beanName, Set<String> autowiredBeanNames, TypeConverter typeConverter) 方法，代码如下：

```
// DefaultListableBeanFactory.java

@Nullable
public Object doResolveDependency(DependencyDescriptor descriptor, @Nullable String beanName,
    @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typeConverter) throws BeansException {
    // 注入点
    InjectionPoint previousInjectionPoint = ConstructorResolver.setCurrentInjectionPoint(descriptor);
    try {
        // 针对给定的工厂给定一个快捷实现的方式，例如考虑一些预先解析的信息
        // 在进入所有bean的常规类型匹配算法之前，解析算法将首先尝试通过此方法解析快捷方式。
        // 子类可以覆盖此方法
        Object shortcut = descriptor.resolveShortcut(this);
        if (shortcut != null) {
            // 返回快捷的解析信息
            return shortcut;
        }
        // 依赖的类型
        Class<?> type = descriptor.getDependencyType();
        // 支持 Spring 的注解 @value
        Object value = getAutowireCandidateResolver().getSuggestedValue(descriptor);
        if (value != null) {
            if (value instanceof String) {
                String strVal = resolveEmbeddedValue((String) value);
                BeanDefinition bd = (beanName != null && containsBean(beanName) ? getMergedBeanDefinition(beanN
                value = evaluateBeanDefinitionString(strVal, bd);
            }
            TypeConverter converter = (typeConverter != null ? typeConverter : getTypeConverter());
            return (descriptor.getField() != null ?
                    converter.convertIfNecessary(value, type, descriptor.getField()) :
                    converter.convertIfNecessary(value, type, descriptor.getMethodParameter()));
        }
        // 解析复合 bean，其实就是对 bean 的属性进行解析
        // 包括：数组、Collection 、Map 类型
        Object multipleBeans = resolveMultipleBeans(descriptor, beanName, autowiredBeanNames, typeConverter);
        if (multipleBeans != null) {
            return multipleBeans;
        }
        // 查找与类型相匹配的 bean
        // 返回值构成为：key = 匹配的 beanName，value = beanName 对应的实例化 bean
        Map<String, Object> matchingBeans = findAutowireCandidates(beanName, type, descriptor);
        // 没有找到，检验 @autowire  的 require 是否为 true
        if (matchingBeans.isEmpty()) {
            // 如果 @autowire 的 require 属性为 true ，但是没有找到相应的匹配项，则抛出异常
            if (isRequired(descriptor)) {
                raiseNoMatchingBeanFound(type, descriptor.getResolvableType(), descriptor);
            }
            return null;
        }
        String autowiredBeanName;
        Object instanceCandidate;
        if (matchingBeans.size() > 1) {
            // 确认给定 bean autowire 的候选者
            // 按照 @Primary 和 @Priority 的顺序
            autowiredBeanName = determineAutowireCandidate(matchingBeans, descriptor);
            if (autowiredBeanName == null) {
                if (isRequired(descriptor) || !indicatesMultipleBeans(type)) {
                    // 唯一性处理
```

```
                    return descriptor.resolveNotUnique(descriptor.getResolvableType(), matchingBeans);
                }
                else {
                    // In case of an optional Collection/Map, silently ignore a non-unique case:
                    // possibly it was meant to be an empty collection of multiple regular beans
                    // (before 4.3 in particular when we didn't even look for collection beans).
                    // 在可选的Collection / Map的情况下，默默地忽略一个非唯一的情况：可能它是一个多个常规bean的
                    return null;
                }
            }
            instanceCandidate = matchingBeans.get(autowiredBeanName);
        } else {
            // We have exactly one match.
            Map.Entry<String, Object> entry = matchingBeans.entrySet().iterator().next();
            autowiredBeanName = entry.getKey();
            instanceCandidate = entry.getValue();
        }
        if (autowiredBeanNames != null) {
            autowiredBeanNames.add(autowiredBeanName);
        }
        if (instanceCandidate instanceof Class) {
            instanceCandidate = descriptor.resolveCandidate(autowiredBeanName, type, this);
        }
        Object result = instanceCandidate;
        if (result instanceof NullBean) {
            if (isRequired(descriptor)) {
                raiseNoMatchingBeanFound(type, descriptor.getResolvableType(), descriptor);
            }
            result = null;
        }
        if (!ClassUtils.isAssignableValue(type, result)) {
            throw new BeanNotOfRequiredTypeException(autowiredBeanName, type, instanceCandidate.getClass());
        }
        return result;
    } finally {
        ConstructorResolver.setCurrentInjectionPoint(previousInjectionPoint);
    }
}
```

- 代码比较多，胖友调试看看。看懂大体逻辑即可。

到这里就已经完成了所有属性的注入了。`populateBean()` 该方法就已经完成了一大半工作了：

> 下一步，则是对依赖 bean 的依赖检测和 PostProcessor 处理，这个我们后面分析。
> 下面，分析该方法的最后一步：`#applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues pvs)` 方法。

1.2 applyPropertyValues

其实，上面只是完成了所有注入属性的获取，将获取的属性封装在 PropertyValues 的实例对象 `pvs` 中，并没有应用到已经实例化的 bean 中。而 `#applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues pvs)` 方法，则是完成这一步骤的。代码如下：

```
// AbstractAutowireCapableBeanFactory.java
```

```
protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues pvs) {
    if (pvs.isEmpty()) {
        return;
    }

    // 设置 BeanWrapperImpl 的 SecurityContext 属性
    if (System.getSecurityManager() != null && bw instanceof BeanWrapperImpl) {
        ((BeanWrapperImpl) bw).setSecurityContext(getAccessControlContext());
    }

    // MutablePropertyValues 类型属性
    MutablePropertyValues mpvs = null;

    // 原始类型
    List<PropertyValue> original;
    // 获得 original
    if (pvs instanceof MutablePropertyValues) {
        mpvs = (MutablePropertyValues) pvs;
        // 属性值已经转换
        if (mpvs.isConverted()) {
            // Shortcut: use the pre-converted values as-is.
            try {
                // 为实例化对象设置属性值 ，依赖注入真真正正地实现在此！！！！！
                bw.setPropertyValues(mpvs);
                return;
            } catch (BeansException ex) {
                throw new BeanCreationException(
                        mbd.getResourceDescription(), beanName, "Error setting property values", ex);
            }
        }
        original = mpvs.getPropertyValueList();
    } else {
        // 如果 pvs 不是 MutablePropertyValues 类型，则直接使用原始类型
        original = Arrays.asList(pvs.getPropertyValues());
    }

    // 获取 TypeConverter = 获取用户自定义的类型转换
    TypeConverter converter = getCustomTypeConverter();
    if (converter == null) {
        converter = bw;
    }

    // 获取对应的解析器
    BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver(this, beanName, mbd, converter);

    // Create a deep copy, resolving any references for values.
    List<PropertyValue> deepCopy = new ArrayList<>(original.size());
    boolean resolveNecessary = false;
    // 遍历属性，将属性转换为对应类的对应属性的类型
    for (PropertyValue pv : original) {
        // 属性值不需要转换
        if (pv.isConverted()) {
            deepCopy.add(pv);
        // 属性值需要转换
        } else {
            String propertyName = pv.getName();
            Object originalValue = pv.getValue(); // 原始的属性值，即转换之前的属性值
            Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue); // 转换属性值，例如将引用
            Object convertedValue = resolvedValue; // 转换之后的属性值
```

```
                boolean convertible = bw.isWritableProperty(propertyName) &&
                        !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);  // 属性值是否可以转换
                // 使用用户自定义的类型转换器转换属性值
                if (convertible) {
                    convertedValue = convertForProperty(resolvedValue, propertyName, bw, converter);
                }
                // Possibly store converted value in merged bean definition,
                // in order to avoid re-conversion for every created bean instance.
                // 存储转换后的属性值，避免每次属性注入时的转换工作
                if (resolvedValue == originalValue) {
                    if (convertible) {
                        // 设置属性转换之后的值
                        pv.setConvertedValue(convertedValue);
                    }
                    deepCopy.add(pv);
                // 属性是可转换的，且属性原始值是字符串类型，且属性的原始类型值不是
                // 动态生成的字符串，且属性的原始值不是集合或者数组类型
                } else if (convertible && originalValue instanceof TypedStringValue &&
                        !((TypedStringValue) originalValue).isDynamic() &&
                        !(convertedValue instanceof Collection || ObjectUtils.isArray(convertedValue))) {
                    pv.setConvertedValue(convertedValue);
                    deepCopy.add(pv);
                } else {
                    resolveNecessary = true;
                    // 重新封装属性的值
                    deepCopy.add(new PropertyValue(pv, convertedValue));
                }
            }
        }
        // 标记属性值已经转换过
        if (mpvs != null && !resolveNecessary) {
            mpvs.setConverted();
        }

        // Set our (possibly massaged) deep copy.
        // 进行属性依赖注入，依赖注入的真真正正实现依赖的注入方法在此！！！
        try {
            bw.setPropertyValues(new MutablePropertyValues(deepCopy));
        } catch (BeansException ex) {
            throw new BeanCreationException(
                    mbd.getResourceDescription(), beanName, "Error setting property values", ex);
        }
    }
```

总结 `#applyPropertyValues(...)` 方法（完成属性转换）：

> 属性值类型不需要转换时，不需要解析属性值，直接准备进行依赖注入。
> 属性值需要进行类型转换时，如对其他对象的引用等，首先需要解析属性值，然后对解析后的属性值进行依赖注入。

而且，我们看到调用了 `#resolveValueIfNecessary(...)`方法对属性值的解析。详细解析，可见 [《Spring应用、原理以及粗读源码系列（一）－框架总述、以Bean为核心的机制（IoC容器初始化以及依赖注入）》](#) 的 [「7. 追踪 resolveValueIfNecessary ，发现是在 BeanDefinitionValueResolver 类」](#) 。

2. 小结

至此，`#doCreateBean(...)` 方法的第二个过程：属性填充已经分析完成了，下篇分析第三个过程：循环依赖的处理。其实，循环依赖并不仅仅只是在 `#doCreateBean(...)` 方法中处理，在整个加载 bean 的过程中都有涉及。所以下篇内容并不仅仅只局限于 `#doCreateBean(...)` 方法。

文章目录