【死磕 Spring】—— IOC 之 属性填充 (http://cmsblogs.com/?p=2885)

2018-11-05 分类：死磕 Spring (http://cmsblogs.com/?cat=206) 阅读(6290) 评论(2)

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

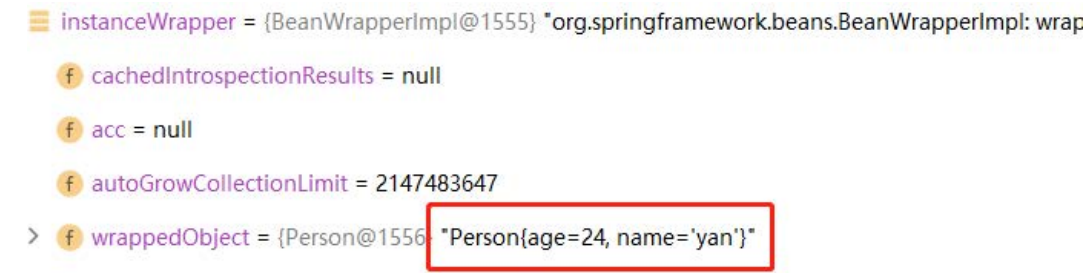
`doCreateBean()` 主要用于完成 bean 的创建和初始化工作，我们可以将其分为四个过程：

- `createBeanInstance()` 实例化 bean
- `populateBean()` 属性填充
- 循环依赖的处理
- `initializeBean()` 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()` ，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)。

把BeanDefinition封装的属性数值，set到instanceWrapper内部的实例对象中。

执行populateBean之前wrapper内部的实例化对象



执行populateBean之后wrapper内部的实例化对象

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

        // 在设置属性之前给 InstantiationAwareBeanPostProcessors 最后一次改变 bean 的机会
        boolean continueWithPropertyPopulation = true;

        // bena 不是"合成"的，即未由应用程序本身定义
        // 是否持有 InstantiationAwareBeanPostProcessor
        if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
            // 迭代所有的 BeanPostProcessors
            for (BeanPostProcessor bp : getBeanPostProcessors()) {
                // 如果为 InstantiationAwareBeanPostProcessor
                if (bp instanceof 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);

        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
                mbd.getResolvedAutowireMode() == RootBeanDefinition.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);

            // 根据名称自动注入
            if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
                autowireByName(beanName, mbd, bw, newPvs);
            }

            // 根据类型自动注入
            if (mbd.getResolvedAutowireMode() == RootBeanDefinition.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() != RootBeanDefinition.DEPENDENCY_CHECK_NONE);

        if (hasInstAwareBpps || needsDepCheck) {
            if (pvs == null) {
                pvs = mbd.getPropertyValues();
            }

            // 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
            PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowC
aching);
            if (hasInstAwareBpps) {
                for (BeanPostProcessor bp : getBeanPostProcessors()) {
                    if (bp instanceof InstantiationAwareBeanPostProcessor) {
                        InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) b
p;
                        // 对所有需要依赖检查的属性进行后处理
                        pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), be
anName);
                        if (pvs == null) {
                            return;
                        }
                    }
                }
            }
            if (needsDepCheck) {
                // 依赖检查，对应 depends-on 属性
                checkDependencies(beanName, mbd, filteredPds, pvs);
            }
        }

        if (pvs != null) {
            // 将属性应用到 bean 中
```

```
            applyPropertyValues(beanName, mbd, bw, pvs);
        }
    }
```

实际上是把beanDefinition内部的属性值取出来，set到bw内部的实例对象中。通过反射执行set属性的方法。

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。
2. 根据注入类型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()` ），统一存入到 PropertyValues 中，PropertyValues 用于描述 bean 的属性。
3. 判断是否需要进行 BeanPostProcessor 和 依赖检测。
4. 将所有 PropertyValues 中的属性填充到 BeanWrapper 中。

自动注入

Spring 会根据注入类型（ byName / byType ）的不同，调用不同的方法（ `autowireByName()` / `autowireByType()` ）来注入属性值。 **autowireByName()** 方法 `autowireByName()` 是根据属性名称完成自动依赖注入的，代码如下：

```
    protected void autowireByName(
            String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) {

        // 对 Bean 对象中非简单属性
        String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
        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.isDebugEnabled()) {
                    logger.debug("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");
                }
            }
        }
    }
```

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，如下：

```
protected String[] unsatisfiedNonSimpleProperties(AbstractBeanDefinition mbd, BeanWrapper bw) {
    Set<String> result = new TreeSet<>();
    PropertyValues pvs = mbd.getPropertyValues();
    PropertyDescriptor[] pds = bw.getPropertyDescriptors();
    for (PropertyDescriptor pd : pds) {
        if (pd.getWriteMethod() != null && !isExcludedFromDependencyCheck(pd) && !pvs.contains(pd.get
Name()) &&
                !BeanUtils.isSimpleProperty(pd.getPropertyType())) {
            result.add(pd.getName());
        }
    }
    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。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()` 完成注册依赖，如下：

```
public void registerDependentBean(String beanName, String dependentBeanName) {
    String canonicalName = canonicalName(beanName);

    synchronized (this.dependentBeanMap) {
        Set<String> dependentBeans =
                this.dependentBeanMap.computeIfAbsent(canonicalName, k -> new LinkedHashSet<>(8));
        if (!dependentBeans.add(dependentBeanName)) {
            return;
        }
    }

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

autowireByType()

```
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);

        for (String propertyName : propertyNames) {
            try {
                // 获取 PropertyDescriptor 实例
                PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);

                // 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);

                    boolean eager = !PriorityOrdered.class.isInstance(bw.getWrappedInstance());
                    DependencyDescriptor desc = new AbstractAutowireCapableBeanFactory.AutowireByTypeDepe
ndencyDescriptor(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, conv
erter);

                    if (autowiredArgument != null) {
                        pvs.add(propertyName, autowiredArgument);
                    }

                    // 迭代方式注入 bean
                    for (String autowiredBeanName : autowibeanredBeanNames) {
                        registerDependentBean(autowiredBeanName, beanName);
                        if (logger.isDebugEnabled()) {
                            logger.debug("Autowiring by type from bean name '" + beanName + "' via proper
ty '" +
                                    propertyName + "' to bean named '" + autowiredBeanName + "'");
                        }
                    }
                    autowiredBeanNames.clear();
                }
            }
            catch (BeansException ex) {
                throw new UnsatisfiedDependencyException(mbd.getResourceDescription(), beanName, property
```

```
Name, 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()` 注册依赖。不过相对于 `autowireByName()` 而言，根据类型寻找相匹配的 bean 过程比较复杂，下面我们就分析这个复杂的过程，如下：

```
    public Object resolveDependency(DependencyDescriptor descriptor, @Nullable String requestingBeanName,
                                    @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter typ
eConverter) 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()) {
            // 创建 Optional 实例依赖类型
            return createOptionalDependency(descriptor, requestingBeanName);
        }

        // 依赖类型为ObjectFactory、ObjectProvider
        else if (ObjectFactory.class == descriptor.getDependencyType() ||
                ObjectProvider.class == descriptor.getDependencyType()) {
            // ObjectFactory / ObjectProvider 用于 用于延迟解析依赖项
            return new DefaultListableBeanFactory.DependencyObjectProvider(descriptor, requestingBeanName
);
        }

        else if (javaxInjectProviderClass == descriptor.getDependencyType()) {
            // javaxInjectProviderClass 类注入的特殊处理
            return new DefaultListableBeanFactory.Jsr330ProviderFactory().createDependencyProvider(descri
ptor, 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, requestingBeanName);
            if (result == null) {
                // 通用处理逻辑
                result = doResolveDependency(descriptor, requestingBeanName, autowiredBeanNames, typeConv
erter);
            }
            return result;
        }
    }
```

这里我们关注通用处理逻辑：`doResolveDependency()`，如下：

```
    public Object doResolveDependency(DependencyDescriptor descriptor, @Nullable String beanName,
                                      @Nullable Set<String> autowiredBeanNames, @Nullable TypeConverter t
ypeConverter) 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) ? getMergedBeanDefini
tion(beanName) : null);
                    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, typeCon
verter);
            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(type, matchingBeans);
                    } else {
                        // 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be
an的空集合

                        return null;
                    }
                }
                instanceCandidate = matchingBeans.get(autowiredBeanName);
            } else {
                // We have exactly one match.
                Map.Entry<Staring, 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.getCl
ass());
            }
            return result;
        } finally {
            ConstructorResolver.setCurrentInjectionPoint(previousInjectionPoint);
        }
    }
```

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()`

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()` 则是完成这一步骤的。

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

        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;


        if (pvs instanceof MutablePropertyValues) {
            mpvs = (MutablePropertyValues) pvs;
            if (mpvs.isConverted()) {
                try {
                    // 设置到 BeanWrapper 中去
                    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
        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) {
```

```
            if (pv.isConverted()) {
                deepCopy.add(pv);
            }
            else {
                String propertyName = pv.getName();
                Object originalValue = pv.getValue();
                Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);
                Object convertedValue = resolvedValue;
                boolean convertible = bw.isWritableProperty(propertyName) &&
                        !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);
                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.
                if (resolvedValue == originalValue) {
                    if (convertible) {
                        pv.setConvertedValue(convertedValue);
                    }
                    deepCopy.add(pv);
                }
                else if (convertible && originalValue instanceof TypedStringValue &&
                        !((TypedStringValue) originalValue).isDynamic() &&
                        !(convertedValue instanceof Collection || ObjectUtils.isArray(convertedValue))) {
                    pv.setConvertedValue(convertedValue);
                    deepCopy.add(pv);
                }
                else {
                    resolveNecessary = true;
                    deepCopy.add(new PropertyValue(pv, convertedValue));
                }
            }
        }
        if (mpvs != null && !resolveNecessary) {
            mpvs.setConverted();
        }

        // Set our (possibly massaged) deep copy.
        try {
            bw.setPropertyValues(new MutablePropertyValues(deepCopy));
        }
        catch (BeansException ex) {
            throw new BeanCreationException(
                    mbd.getResourceDescription(), beanName, "Error setting property values", ex);
        }
    }
```

至此， `doCreateBean()` 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()` 中处理，其实在整个加载 bean 的过程中都有涉及，所以下篇内容并不仅仅只局限于 `doCreateBean()`。 **更多阅读**

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