# 【死磕 Spring】—— IOC 之分析 BeanWrapper (http://cmsblogs.com/?p=4020)

2019-01-28 分类：死磕 Spring (http://cmsblogs.com/?cat=206) 阅读(10322) 评论(2)

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

在实例化 bean 阶段，我们从 BeanDefinition 得到的并不是我们最终想要的 Bean 实例，而是 BeanWrapper 实例，如下：

```
*/
protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullable Object[] args)
        throws BeanCreationException {

    // Instantiate the bean.
    BeanWrapper instanceWrapper = null;
    if (mbd.isSingleton()) {
        instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
    }
    if (instanceWrapper == null) {
        instanceWrapper = createBeanInstance(beanName, mbd, args);
    }
    final Object bean = instanceWrapper.getWrappedInstance();
    Class<?> beanType = instanceWrapper.getWrappedClass();
    if (beanType != NullBean.class) {
        mbd.resolvedTargetType = beanType;
    }
```

(https://gitee.com/chenssy/blog-home/raw/master/image/201811/15393305208552.jpg)

所以这里 BeanWrapper 是一个从 BeanDefinition 到 Bean 直接的中间产物，我们可以称它为”低级 bean“，在一般情况下，我们不会在实际项目中用到它。BeanWrapper 是 Spring 框架中重要的组件类，它就相当于一个代理类，Spring 委托 BeanWrapper 完成 Bean 属性的填充工作。在 bean 实例被 InstantiatioonStrategy 创建出来后，Spring 容器会将 Bean 实例通过 BeanWrapper 包裹起来，是通过 `BeanWrapper.setWrappedInstance()` 完成的，如下：

```
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);
    }
}
```

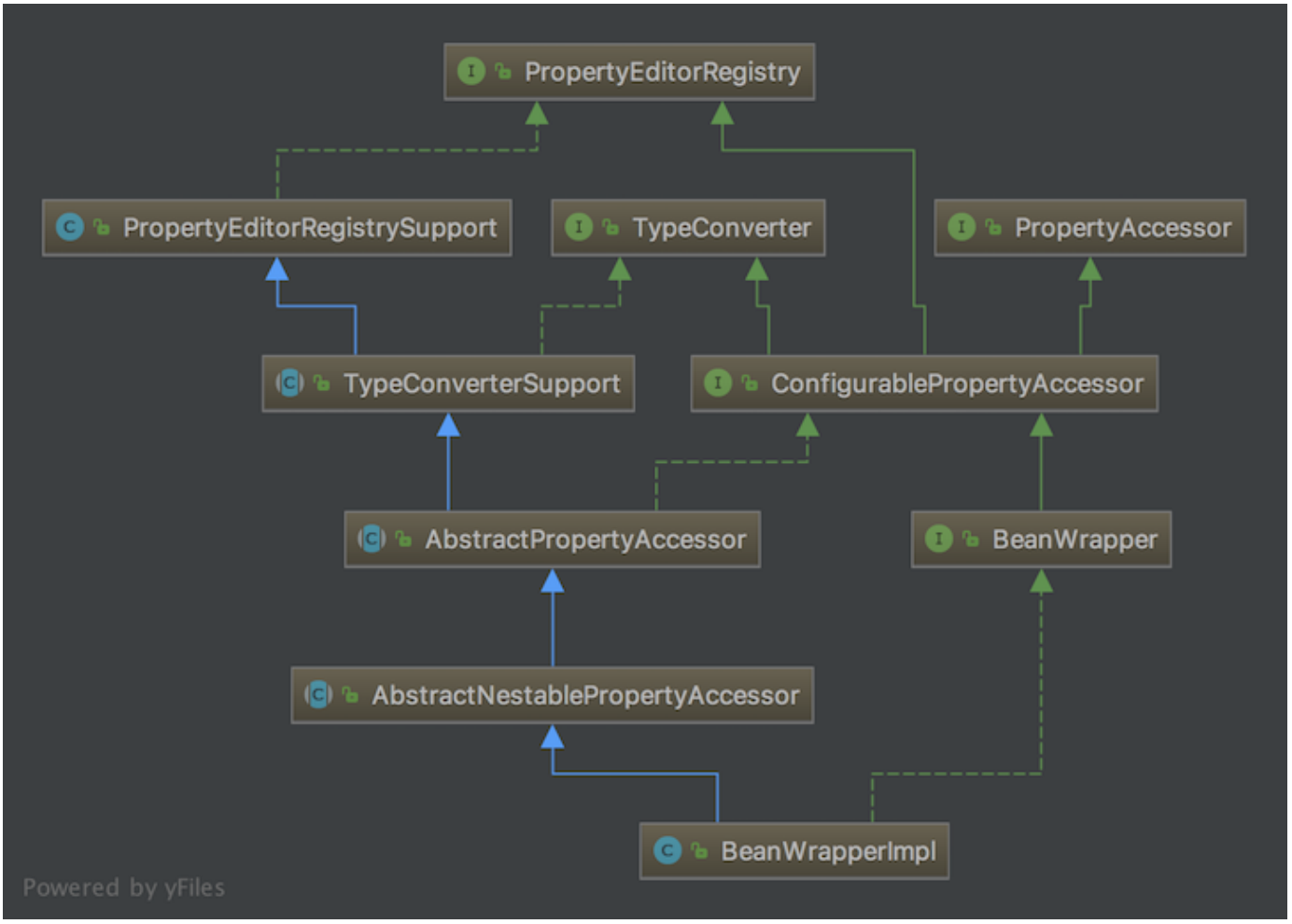
(https://gitee.com/chenssy/blog-home/raw/master/image/201811/15393328767959.jpg)

beanInstance 就是我们实例出来的 bean 实例，通过构造一个 BeanWrapper 实例对象进行包裹，如下：

```
public BeanWrapperImpl(Object object) {
 super(object);
}

protected AbstractNestablePropertyAccessor(Object object) {
 registerDefaultEditors();
 setWrappedInstance(object);
}
```

下面小编就 BeanWrapper 来进行分析说明，先看整体的结构：




(https://gitee.com/chenssy/blog-home/raw/master/image/201811/20181012100001.png)

从上图可以看出 BeanWrapper 主要继承三个核心接口：PropertyAccessor、PropertyEditorRegistry、TypeConverter。

**PropertyAccessor**

> 可以访问属性的通用型接口（例如对象的 bean 属性或者对象中的字段），作为 BeanWrapper 的基础接口。

```
public interface PropertyAccessor {
  String NESTED_PROPERTY_SEPARATOR = ".";
  char NESTED_PROPERTY_SEPARATOR_CHAR = '.';

  String PROPERTY_KEY_PREFIX = "[";
  char PROPERTY_KEY_PREFIX_CHAR = '[';

  String PROPERTY_KEY_SUFFIX = "]";
  char PROPERTY_KEY_SUFFIX_CHAR = ']';

  boolean isReadableProperty(String propertyName);

  boolean isWritableProperty(String propertyName);

  Class<?> getPropertyType(String propertyName) throws BeansException;

  TypeDescriptor getPropertyTypeDescriptor(String propertyName) throws BeansException;

  Object getPropertyValue(String propertyName) throws BeansException;

  void setPropertyValue(String propertyName, @Nullable Object value) throws BeansException;

  void setPropertyValue(PropertyValue pv) throws BeansException;

  void setPropertyValues(Map<?, ?> map) throws BeansException;

  void setPropertyValues(PropertyValues pvs) throws BeansException;

  void setPropertyValues(PropertyValues pvs, boolean ignoreUnknown)
    throws BeansException;

  void setPropertyValues(PropertyValues pvs, boolean ignoreUnknown, boolean ignoreInvalid)
    throws BeansException;

}
```

就上面的源码我们可以分解为四类方法：

- `isReadableProperty()`：判断指定 property 是否可读，是否包含 getter 方法
- `isWritableProperty()`：判断指定 property 是否可写,是否包含 setter 方法
- `getPropertyType()`：获取指定 propertyName 的类型
- `setPropertyValue()`：设置指定 propertyValue

**PropertyEditorRegistry**

> 用于注册 JavaBean 的 PropertyEditors，对 PropertyEditorRegistrar 起核心作用的中心接口。由 BeanWrapper 扩展，BeanWrapperImpl 和 DataBinder 实现。

```
public interface PropertyEditorRegistry {

    void registerCustomEditor(Class<?> requiredType, PropertyEditor propertyEditor);

    void registerCustomEditor(@Nullable Class<?> requiredType, @Nullable String propertyPath, PropertyEditor
propertyEditor);

    @Nullable
    PropertyEditor findCustomEditor(@Nullable Class<?> requiredType, @Nullable String propertyPath);

}
```

根据接口提供的方法，PropertyEditorRegistry 就是用于 PropertyEditor 的注册和发现，而 PropertyEditor 是 Java 内省里面的接口，用于改变指定 property 属性的类型。

**TypeConverter**

> 定义类型转换的接口，通常与 PropertyEditorRegistry 接口一起实现（但不是必须），但由于 TypeConverter 是基于线程不安全的 PropertyEditors，因此 TypeConverters 本身也不被视为线程安全。这里小编解释下，在 Spring 3 后，不在采用 PropertyEditors 类作为 Spring 默认的类型转换接口，而是采用 ConversionService 体系，但 ConversionService 是线程安全的，所以在 Spring 3 后，如果你所选择的类型转换器是 ConversionService 而不是 PropertyEditors 那么 TypeConverters 则是线程安全的。

```
public interface TypeConverter {

    <T> T convertIfNecessary(Object value, Class<T> requiredType) throws TypeMismatchException;

    <T> T convertIfNecessary(Object value, Class<T> requiredType, MethodParameter methodParam)
            throws TypeMismatchException;

    <T> T convertIfNecessary(Object value, Class<T> requiredType, Field field)
            throws TypeMismatchException;
}
```

BeanWrapper 继承上述三个接口，那么它就具有三重身份：

- 属性编辑器
- 属性编辑器注册表
- 类型转换器

BeanWrapper 继承 ConfigurablePropertyAccessor 接口，该接口除了继承上面介绍的三个接口外还集成了 Spring 的 ConversionService 类型转换体系。

```
public interface ConfigurablePropertyAccessor extends PropertyAccessor, PropertyEditorRegistry, TypeConverter {

    void setConversionService(@Nullable ConversionService conversionService);

    @Nullable
    ConversionService getConversionService();

    void setExtractOldValueForEditor(boolean extractOldValueForEditor);

    boolean isExtractOldValueForEditor();

    void setAutoGrowNestedPaths(boolean autoGrowNestedPaths);

    boolean isAutoGrowNestedPaths();

}
```

`setConversionService()` 和 `getConversionService()` 则是用于集成 Spring 的 ConversionService 类型转换体系。

**BeanWrapper**

> Spring 的 低级 JavaBean 基础结构的接口，一般不会直接使用，而是通过 BeanFactory 或者 DataBinder 隐式使用。它提供分析和操作标准 JavaBeans 的操作：获取和设置属性值、获取属性描述符以及查询属性的可读性/可写性的能力。

```
public interface BeanWrapper extends ConfigurablePropertyAccessor {

    void setAutoGrowCollectionLimit(int autoGrowCollectionLimit);

    int getAutoGrowCollectionLimit();

    Object getWrappedInstance();

    Class<?> getWrappedClass();

    PropertyDescriptor[] getPropertyDescriptors();

    PropertyDescriptor getPropertyDescriptor(String propertyName) throws InvalidPropertyException;
}
```

下面几个方法比较重要：

个对象有4个方法比较重要:

- `getWrappedInstance()`：获取包装对象的实例。

- `getWrappedClass()`：获取包装对象的类型。
- `getPropertyDescriptors()`：获取包装对象所有属性的 PropertyDescriptor 就是这个属性的上下文。
- `getPropertyDescriptor()`：获取包装对象指定属性的上下文。

**BeanWrapperImpl**

> BeanWrapper 接口的默认实现，用于对Bean的包装，实现上面接口所定义的功能很简单包括设置获取被包装的对象，获取被包装bean的属性描述器

BeanWrapper 体系相比于 Spring 中其他体系是比较简单的，它作为 BeanDefinition 向 Bean 转换过程中的中间产物，承载了 bean 实例的包装、类型转换、属性的设置以及访问等重要作用。



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)

**chenssy (http://cmsblogs.com/?author=1)**
不想当厨师的程序员不是好的架构师....

上一篇
深入理解 Java 内存模型文集 (http://cmsblogs.com/?p=4012)

下一篇
【死磕 Spring】—— IOC 之 bean 的实例化策略：InstantiationStrategy (http://cmsblogs.com/?p=4022)

- 【死磕 Redis】—— 如何排查 Redis 中的慢查询 (http://cmsblogs.com/?p=18352)
- 【死磕 Redis】—— 发布与订阅 (http://cmsblogs.com/?p=18348)
- 【死磕 Redis】—— 布隆过滤器 (http://cmsblogs.com/?p=18346)
- 【死磕 Redis】—— 理解 pipeline 管道 (http://cmsblogs.com/?p=18344)
- 【死磕 Redis】—— 事务 (http://cmsblogs.com/?p=18340)
- 【死磕 Redis】—— Redis 的线程模型 (http://cmsblogs.com/?p=18337)
- 【死磕 Redis】—— Redis 通信协议 RESP (http://cmsblogs.com/?p=18334)
- 【死磕 Redis】—— 开篇 (http://cmsblogs.com/?p=18332)
- spring boot 源码解析11-ConfigurationClassPostProcessor类加载解析 (http://cmsblogs.com/?p=9522)