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
2019-09-15
Spring

【死磕 Spring】—— IoC 之分析 BeanWrapper

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！

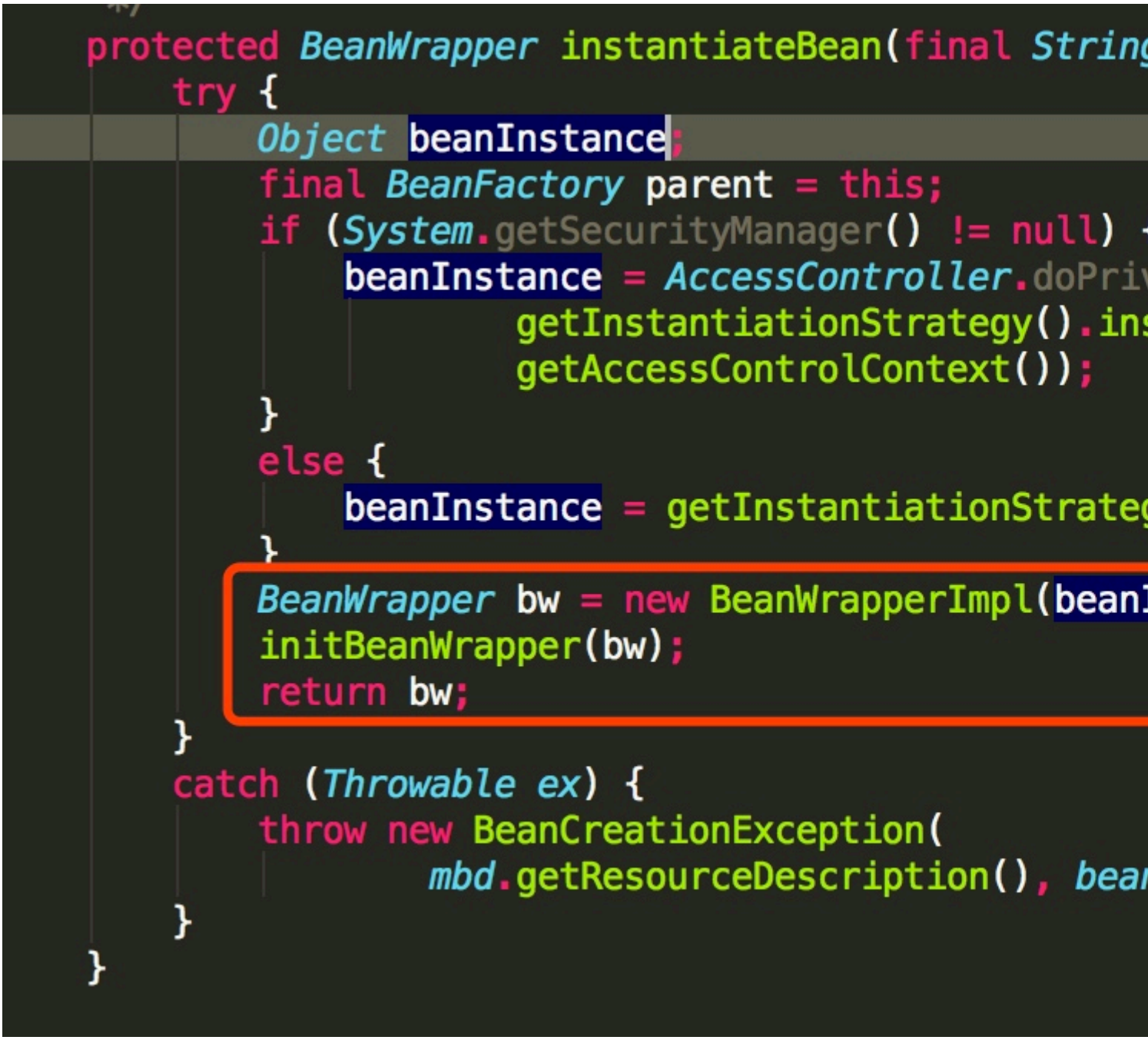
作为「小明哥」的忠实读者，「老芳芳」略作修改，记录在理解过程中，参考的资料。

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,
        throws BeanCreationException {

    // Instantiate the bean.
    BeanWrapper instanceWrapper = null;
    if (mbd.isSingleton()) {
        instanceWrapper = this.factoryBeanInstanceCa
    }
    if (instanceWrapper == null) {
        instanceWrapper = createBeanInstance(beanNam
    }
    final Object bean = instanceWrapper.getWrappedIn
    Class<?> beanType = instanceWrapper.getWrappedCl
    if (beanType != NullBean.class) {
        mbd.resolvedTargetType = beanType;
    }
```

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 InstantiationStrategy 创建出来后，Spring 容器会将 Bean 实例通过 BeanWrapper 包裹起来，是通过如如下代码实现：



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：

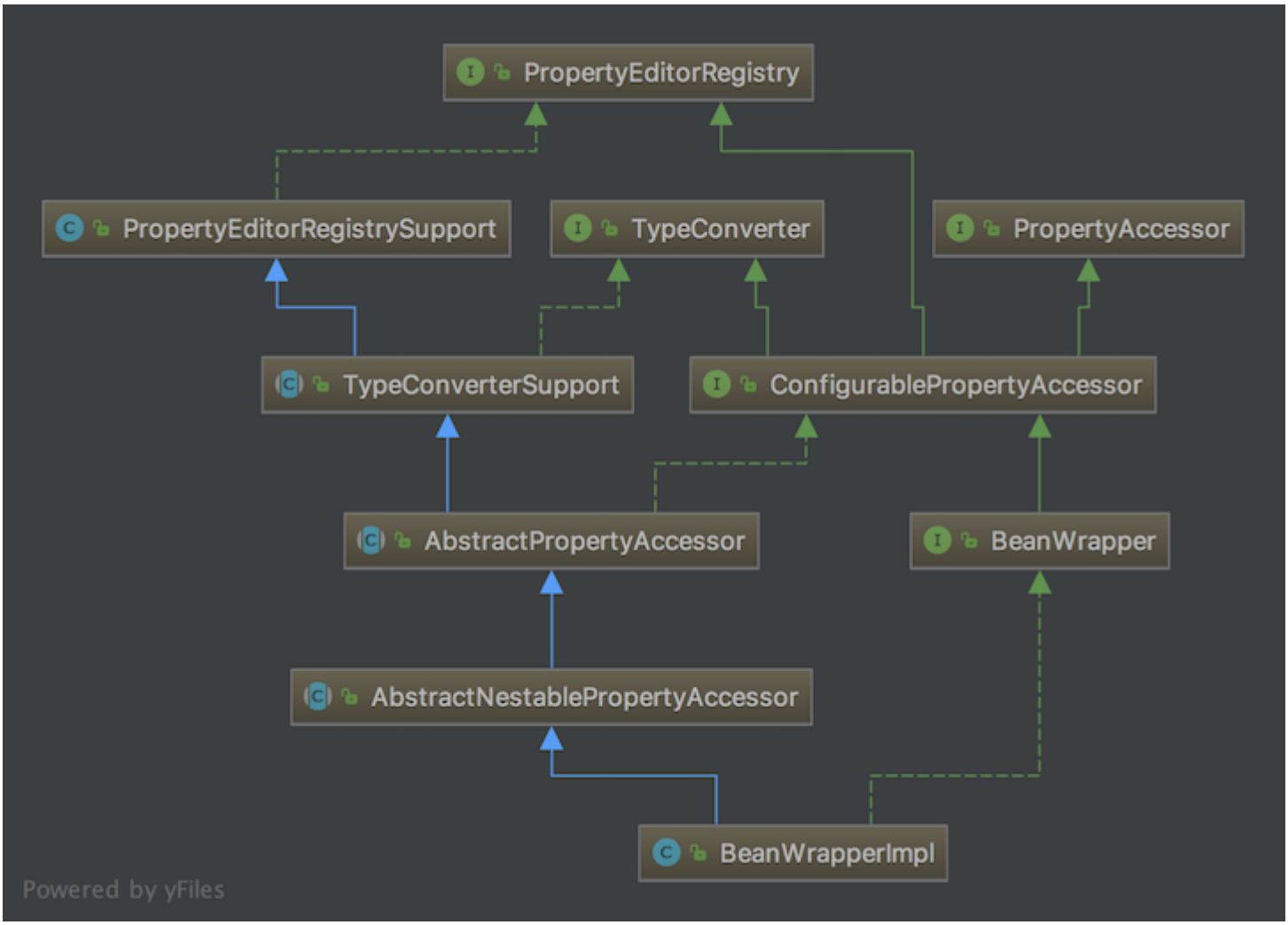
```
// BeanWrapperImpl.java

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：



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。

```
// PropertyAccessor.java

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(String propertyName)` 方法：判断指定 property 是否可读，是否包含 getter 方法。
`#isWritableProperty(String propertyName)` 方法：判断指定 property 是否可写,是否包含 setter 方法。
`#getPropertyType(...)` 方法：获取指定 propertyName 的类型
`#setPropertyValue(...)` 方法：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。

```
// PropertyEditorRegistry.java

public interface PropertyEditorRegistry {

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

    void registerCustomEditor(@Nullable Class<?> requiredType, @Nullable String propertyPath, PropertyEditor property

    @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。

```
// ConfigurablePropertyAccessor.java

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(ConversionService conversionService)` 和 `#getConversionService()` 方法，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。

```
// BeanWrapper.java
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：

#getWrappedInstance() 方法：获取包装对象的实例。
#getWrappedClass() 方法：获取包装对象的类型。
#getPropertyDescriptors() 方法：获取包装对象所有属性的 PropertyDescriptor 就是这个属性的上下文。
#getPropertyDescriptor(String propertyName) 方法：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

小结

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。