什么是BeanDefinition?

什么是BeanDefinition?

* **什么是BeanDefinition? **

* + **一句话概括：spring bean的建模对象;**

* BeanDefiniton就是一个bean实例化出来的模型对象?

* + 有人会问把一个bean实例化出来有Class就行了啊，Class也就是我们通常说的类对象，就是一个普通对象的建模对象；

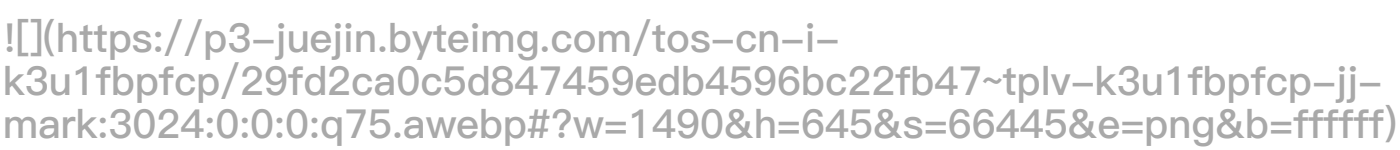
* 那么为什么spring不能用Class来建立bean呢?

* + 很简单，因为Class无法完成bean的抽象，比如bean的作用域，bean的注入模型，bean是否是懒加载等等信息，Class是无法抽象出来的，故而需要一个BeanDefinition类来抽象这些信息（当然除了抽象信息还提供了许多的api也就是方法），**以便于spring能够完美的实例化一个bean；**

BeanDefinition在spring源码当中的设计

首先需要说明的beanDefinition只是一个总称，spring当中为了各种类型不同的bean设计了不同的beanDefinition来对应；比如xml定义的bean和加了@Service来定义的bean，spring不是用同一个beandefinition来描述的；

下面可以通过idea的工具来分析一下关于beandefinition的设计以及各个beandefinition的关系：



BeanMetadataElement
==========

bean元数据，返回该bean的来源。

```
package org.springframework.beans;

import org.springframework.lang.Nullable;

/**
 * Interface to be implemented by bean metadata elements
 * that carry a configuration source object.
 *
 * 由携带配置源对象的 Bean 元数据元素实现的接口。
 *
 * @author Juergen Hoeller
 * @since 2.0
 */
public interface BeanMetadataElement {

/**
 * Return the configuration source {@code Object} for this metadata
element
 * 返回此元数据元素的配置源 {@code 对象}
 * (may be {@code null}).
 */
@Nullable
default Object getSource() {
return null;
}

}
```

AttributeAccessor
==========

Spring定义的属性访问器，对BeanDefinition的属性进行操作的API，例如：设置属性、获取属性、判断是否存在该属性，返回bean所有的属性名称等；当然这里的属性指的是程序员额外提供的，相当于一个map，之前咱们在说配置类

的时候，判断什么类是全配置类，并且解析该类的时候，会有一个attribute属性，相当于一个k，v结果存储着额外扩展的属性，例如bean加载的排序。

```
//设置一个属性
void setAttribute(String name, @Nullable Object value);

//获取一个属性
@Nullable
Object getAttribute(String name);

//删除一个属性
@Nullable
Object removeAttribute(String name);

//判断某个属性是否存在
boolean hasAttribute(String name);

//返回所有的属性名字
String[] attributeNames();
```

BeanDefinition
==========

首先beandefinition这个接口是继承了上面两个接口的：下面对BeanDefinition接口中较为重要的方法进行解析。

```
public interface BeanDefinition extends AttributeAccessor,
BeanMetadataElement {

// 单例、原型标识符
//String SCOPE_SINGLETON = ”singleton”;
String SCOPE_SINGLETON =
ConfigurableBeanFactory.SCOPE_SINGLETON;
//String SCOPE_SINGLETON = ”prototype”;
String SCOPE_PROTOTYPE =
ConfigurableBeanFactory.SCOPE_PROTOTYPE;

// 标识 Bean 的类别，分别对应 用户定义的 Bean、来源于配置文件的
Bean、Spring 内部的 Bean
```

```
int ROLE_APPLICATION = 0;
int ROLE_SUPPORT = 1;
int ROLE_INFRASTRUCTURE = 2;

//设置、返回 Bean 的父类beandefinition名称
void setParentName(@Nullable String parentName);
@Nullable
String getParentName();

//设置、返回beandefinition的类的名字
void setBeanClassName(@Nullable String beanClassName);
@Nullable
String getBeanClassName();

//设置和得到bean的作用域
void setScope(@Nullable String scope);
@Nullable
String getScope();

//设置、获取是否懒加载
void setLazyInit(boolean lazyInit);
boolean isLazyInit();

//设置和返回DependsOn的那些bean的名字
void setDependsOn(@Nullable String... dependsOn);
@Nullable
String[] getDependsOn();

// 设置、返回 Bean 是否可以自动注入。只对 @Autowired 注解有效
void setAutowireCandidate(boolean autowireCandidate);
boolean isAutowireCandidate();

// 设置、返回当前 Bean 是否为主要候选 Bean 。
// 当同一个接口有多个实现类时，通过该属性来配置某个 Bean 为主候选 Bean。
void setPrimary(boolean primary);
boolean isPrimary();

// 设置、返回创建该 Bean 的工厂类的名字
void setFactoryBeanName(@Nullable String factoryBeanName);
@Nullable
String getFactoryBeanName();

//设置、返回创建该 Bean 的工厂方法
void setFactoryMethodName(@Nullable String factoryMethodName);
@Nullable
String getFactoryMethodName();
```

```
//得到一个数据结构，该结构用来存储实例化该bean的时候如果是通过构造方法注入
//或者有特殊构造方法就从改结构当中获取值
ConstructorArgumentValues getConstructorArgumentValues();
//不解释
default boolean hasConstru
ctorArgumentValues() {
return !getConstructorArgumentValues().isEmpty();
}

//等同于上面的getConstructorArgumentValues 只不过存的是属性的值
MutablePropertyValues getPropertyValues();
default boolean hasPropertyValues() {
return !getPropertyValues().isEmpty();
}

//设置、获取bean的初始化方法名字
void setInitMethodName(@Nullable String initMethodName);
@Nullable
String getInitMethodName();

//设置、获取bean的销毁方法
void setDestroyMethodName(@Nullable String destroyMethodName);
@Nullable
String getDestroyMethodName();

//设置角色 对应上面的三个值
void setRole(int role);
int getRole();

//设置获取bean的描述信息
void setDescription(@Nullable String description);
@Nullable
String getDescription();

// 这个以后再说
ResolvableType getResolvableType();

//是否单例
boolean isSingleton();

//是否原型
boolean isPrototype();

//是否抽象
boolean isAbstract();

```

下面我们来一一解析上述的代码中的点：

1. 这两个String类型的参数的作用是来帮助我们在构建一个自己想要的BeanDefinition的时候防止你写错了，来提供的枚举类的作用。

```
/**
 * Scope identifier for the standard singleton scope: {@value}.
 * <p>Note that extended bean factories might support further scopes.
 * @see #setScope
 * @see ConfigurableBeanFactory#SCOPE_SINGLETON
 */
String SCOPE_SINGLETON =
ConfigurableBeanFactory.SCOPE_SINGLETON;

/**
 * Scope identifier for the standard prototype scope: {@value}.
 * <p>Note that extended bean factories might support further scopes.
 * @see #setScope
 * @see ConfigurableBeanFactory#SCOPE_PROTOTYPE
 */
String SCOPE_PROTOTYPE =
ConfigurableBeanFactory.SCOPE_PROTOTYPE;
```

```
GenericBeanDefinition genericBeanDefinition = new
GenericBeanDefinition();
genericBeanDefinition.setScope(BeanDefinition.SCOPE_SINGLETON);
genericBeanDefinition.setScope(BeanDefinition.SCOPE_PROTOTYPE);
```

2. 下面的三个常量：这三个常量的作用是为了标识Bean的加载等级。

```
/**
 * Role hint indicating that a {@code BeanDefinition} is a major part
 * of the application. Typically corresponds to a user-defined bean.
 */
```

```
int ROLE_APPLICATION = 0;

/**
 * Role hint indicating that a {@code BeanDefinition} is a supporting
 * part of some larger configuration, typically an outer
 * {@link
org.springframework.beans.factory.parsing.ComponentDefinition}.
 * {@code SUPPORT} beans are considered important enough to be
aware
 * of when looking more closely at a particular
 * {@link
org.springframework.beans.factory.parsing.ComponentDefinition},
 * but not when looking at the overall configuration of an application.
 */
int ROLE_SUPPORT = 1;

/**
 * Role hint indicating that a {@code BeanDefinition} is providing an
 * entirely background role and has no relevance to the end-user. This
hint is
 * used when registering beans that are completely part of the internal
workings
 * of a {@link
org.springframework.beans.factory.parsing.ComponentDefinition}.
 */
int ROLE_INFRASTRUCTURE = 2;

```


```
GenericBeanDefinition genericBeanDefinition = new
GenericBeanDefinition();
genericBeanDefinition.setScope(BeanDefinition.SCOPE_SINGLETON);
genericBeanDefinition.setScope(BeanDefinition.SCOPE_PROTOTYPE);
int role = genericBeanDefinition.getRole();

```


```
public int getRole() {
    return this.role;
}

```

```
private int role = BeanDefinition.ROLE_APPLICATION;
// 发现它是AbstractBeanDefinition类的一个私有属性，并且已经被赋值了一个初始值 BeanDefinition.ROLE_APPLICATION
```

```
int ROLE_APPLICATION = 0;
```

这个role字段的唯一作用就是为了排好执行BeanDefinition的先后的等级，在DefaultLisbleBeanFactory的registryBeanDefinition()方法中：

```
//-------------------------------------------------------------------
// Implementation of BeanDefinitionRegistry
interface:BeanDefinitionRegistry 接口的实现
//-------------------------------------------------------------------
@Override
public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition) throws BeanDefinitionStoreException {
    Assert.hasText(beanName, "Bean name must not be empty");
    Assert.notNull(beanDefinition, "BeanDefinition must not be null");
    if (beanDefinition instanceof AbstractBeanDefinition) {
        try {
            ((AbstractBeanDefinition) beanDefinition).validate();
        } catch (BeanDefinitionValidationException ex) {
            throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,
                    "Validation of bean definition failed", ex);
        }
    }
    //在注册beanDefinition的时候判断该名字有没有被注册
    BeanDefinition existingDefinition = this.beanDefinitionMap.get(beanName);
    if (existingDefinition != null) {
        if (!isAllowBeanDefinitionOverriding()) {
            throw new BeanDefinitionOverrideException(beanName, beanDefinition, existingDefinition);
        } else if (existingDefinition.getRole() < beanDefinition.getRole()) {
```

```
            // e.g. was ROLE_APPLICATION, now overriding with
ROLE_SUPPORT or ROLE_INFRASTRUCTURE
            if (logger.isInfoEnabled()) {
                logger.info("Overriding user-defined bean definition for bean
'" + beanName +
                        "' with a framework-generated bean definition:
replacing [" +
                        existingDefinition + "] with [" + beanDefinition + "]");
            }
        } else if (!beanDefinition.equals(existingDefinition)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Overriding bean definition for bean '" +
beanName +
                        "' with a different definition: replacing [" +
existingDefinition +
                        "] with [" + beanDefinition + "]");
            }
        } else {
            if (logger.isTraceEnabled()) {
                logger.trace("Overriding bean definition for bean '" +
beanName +
                        "' with an equivalent definition: replacing [" +
existingDefinition +
                        "] with [" + beanDefinition + "]");
            }
        }
        this.beanDefinitionMap.put(beanName, beanDefinition);
        //如果没有就会进入到else中
    } else {
        //这个时候又要判断,当前spring容器是否开始实例化bean了?
        //判断的依据是hasBeanCreationStarted()方法是否有值?
        /*
         *protected boolean hasBeanCreationStarted() {
         * return !this.alreadyCreated.isEmpty();
         *}
         */
        //其实就是判断这个alreadyCreated set集合是否有值?
        if (hasBeanCreationStarted()) {
            // Cannot modify startup-time collection elements anymore (for
stable iteration)
            // 这句的意思:当我们开始实例某个bean的时候,一定要确保当前被实
例化的bean的beanDefinition信息是固定的,是不可修改的。
            // 这就是beanFactory.freezeConfiguration();方法主要做的一件事情
,其实讲白了就是防止出现线程安全的问题,你想想一个线程在修改
BeanDefinition一个线程在读取当前Bean的BeanDefinition是不是就很有问题
?
            synchronized (this.beanDefinitionMap) {
                this.beanDefinitionMap.put(beanName, beanDefinition);
```

```
                List<String> updatedDefinitions = new
ArrayList<>(this.beanDefinitionNames.size() + 1);
                updatedDefinitions.addAll(this.beanDefinitionNames);
                updatedDefinitions.add(beanName);
                this.beanDefinitionNames = updatedDefinitions;
                //如果你注册了beanDefinition的名称和手动注册的
BeanDefinitionMap中某个相同则删除手工注册的
                removeManualSingletonName(beanName);
            }
        }
        //如果进入到这个else中表示Bean已经开始实例化了
        else {
            // Still in startup registration phase
            // 这里的注释为什么说,当前的容器仍处于启动注册阶段就可以不像上
面的if块中加synchronized来防止并发问题,就是因为他都还没有开始实例化呢,
            // 所以你在这个时候进行修改啥的都没有问题,当前的Bean的
BeanDefinition本身就是不确定的,你修改了也只是修改了我
BeanDefinitionMap中的beanDefinition而已,
            // 这个时候spring并没有通过beanDefinition将当前的Bean进行实例
化,所以这里不加synchronized是没有问题的。
            this.beanDefinitionMap.put(beanName, beanDefinition);
            this.beanDefinitionNames.add(beanName);
            //如果你注册了beanDefinition的名称和手动注册的
BeanDefinitionMap中某个相同则删除手工注册的
            removeManualSingletonName(beanName);
        }
        this.frozenBeanDefinitionNames = null;
    }
    //判断register的beanDefinition已经存在(根据名字判断)
    if (existingDefinition != null || containsSingleton(beanName)) {
        //清除allBeanNamesByType
        //把单例池中的bean也remove
        resetBeanDefinition(beanName);
    }
    //如果被冻结了,则表示可能是有缓存
    else if (isConfigurationFrozen()) {
        //所以还是要清除一下缓存
        clearByTypeCache();
    }
}

```

可以看到它只是做了一个判断，并且输出了相对应的log。然后我们说一下三个等级分别代表了spring对应BeanDefinition的那些定义：

* 0：用户定义的 Bean。

* 1：来源于配置文件的 Bean。
* 2：Spring 内部的 Bean。

我们都知道spring是有一些内置的bean在spring容器启动的时候就会被注册到spring容器内部了，这些bean的等级应该是 2 。

```
public AnnotationConfigApplicationContext() {
this.reader = new AnnotatedBeanDefinitionReader(this);
this.scanner = new ClassPathBeanDefinitionScanner(this);
}
```

```
public AnnotatedBeanDefinitionReader(BeanDefinitionRegistry registry,
Environment environment) {
Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
Assert.notNull(environment, "Environment must not be null");
this.registry = registry;
this.conditionEvaluator = new ConditionEvaluator(registry, environment,
null);
//注册注解配置的处理器
AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
}
```

```
public static Set<BeanDefinitionHolder>
registerAnnotationConfigProcessors(BeanDefinitionRegistry registry,
@Nullable Object source) {
    DefaultListableBeanFactory beanFactory =
unwrapDefaultListableBeanFactory(registry);
    //这段代码是给beanFactory提供能力
    if (beanFactory != null) {
        if (!(beanFactory.getDependencyComparator() instanceof
AnnotationAwareOrderComparator)) {
            //解析spring当中实现了Ordered的bean、以及加了@Order注解和
@Priority注解
beanFactory.setDependencyComparator(AnnotationAwareOrderComparat
or.INSTANCE);
        }
```

```
        if (!(beanFactory.getAutowireCandidateResolver() instanceof
ContextAnnotationAutowireCandidateResolver)) {
            //用于分析特定的BeanDefinition是否符合特定的依赖项的候选者的
策略
            beanFactory.setAutowireCandidateResolver(new
ContextAnnotationAutowireCandidateResolver());
        }
    }
    Set<BeanDefinitionHolder> beanDefs = new LinkedHashSet<>(8);
    if
(!registry.containsBeanDefinition(CONFIGURATION_ANNOTATION_PROC
ESSOR_BEAN_NAME)) {
        RootBeanDefinition def = new
RootBeanDefinition(ConfigurationClassPostProcessor.class);
        def.setSource(source);
        //将创建的rootBeanDefinition注册到springContext
        BeanDefinitionHolder beanDefinitionHolder =
registerPostProcessor(registry, def,
CONFIGURATION_ANNOTATION_PROCESSOR_BEAN_NAME);
        beanDefs.add(beanDefinitionHolder);
    }
    if
(!registry.containsBeanDefinition(AUTOWIRED_ANNOTATION_PROCESS
OR_BEAN_NAME)) {
        RootBeanDefinition def = new
RootBeanDefinition(AutowiredAnnotationBeanPostProcessor.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def,
AUTOWIRED_ANNOTATION_PROCESSOR_BEAN_NAME));
    }
    // Check for JSR-250 support, and if present add the
CommonAnnotationBeanPostProcessor.
    if (jsr250Present &&
!registry.containsBeanDefinition(COMMON_ANNOTATION_PROCESSOR_
BEAN_NAME)) {
        RootBeanDefinition def = new
RootBeanDefinition(CommonAnnotationBeanPostProcessor.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def,
COMMON_ANNOTATION_PROCESSOR_BEAN_NAME));
    }
    // Check for JPA support, and if present add the
PersistenceAnnotationBeanPostProcessor.
    if (jpaPresent &&
!registry.containsBeanDefinition(PERSISTENCE_ANNOTATION_PROCESS
OR_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition();
        try {
```

```
def.setBeanClass(ClassUtils.forName(PERSISTENCE_ANNOTATION_PROCESSOR_CLASS_NAME, AnnotationConfigUtils.class.getClassLoader()));
        } catch (ClassNotFoundException ex) {
            throw new IllegalStateException("Cannot load optional framework class: " + PERSISTENCE_ANNOTATION_PROCESSOR_CLASS_NAME, ex);
        }
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, PERSISTENCE_ANNOTATION_PROCESSOR_BEAN_NAME));
    }
    if (!registry.containsBeanDefinition(EVENT_LISTENER_PROCESSOR_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition(EventListenerMethodProcessor.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, EVENT_LISTENER_PROCESSOR_BEAN_NAME));
    }
    if (!registry.containsBeanDefinition(EVENT_LISTENER_FACTORY_BEAN_NAME)) {
        RootBeanDefinition def = new RootBeanDefinition(DefaultEventListenerFactory.class);
        def.setSource(source);
        beanDefs.add(registerPostProcessor(registry, def, EVENT_LISTENER_FACTORY_BEAN_NAME));
    }
    return beanDefs;
}
```

可以看到这些高亮的行上面都有一个方法registerPostProcessor，该方法就是对这些spring内置的bean设置role的。

```
private static BeanDefinitionHolder registerPostProcessor(
BeanDefinitionRegistry registry,
RootBeanDefinition definition,
String beanName) {
// 在初始化spring read初始化的BeanDefinition的role为ROLE_INFRASTRUCTURE(2)标识该BeanDefinition为spring内部的BeanDefinition
definition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
```

```
registry.registerBeanDefinition(beanName, definition);
return new BeanDefinitionHolder(definition, beanName);
}
```

```
int ROLE_INFRASTRUCTURE = 2;
```

```
//在注册beanDefinition的时候判断该名字有没有被注册
BeanDefinition existingDefinition =
this.beanDefinitionMap.get(beanName);
if (existingDefinition != null) {
    if (!isAllowBeanDefinitionOverriding()) {
        throw new BeanDefinitionOverrideException(beanName,
beanDefinition, existingDefinition);
    } else if (existingDefinition.getRole() < beanDefinition.getRole()) {
        // e.g. was ROLE_APPLICATION, now overriding with
ROLE_SUPPORT or ROLE_INFRASTRUCTURE
        if (logger.isInfoEnabled()) {
            logger.info("Overriding user-defined bean definition for bean '"
+ beanName +
                    "' with a framework-generated bean definition: replacing
[" +
                    existingDefinition + "] with [" + beanDefinition + "]");
        }
    } else if (!beanDefinition.equals(existingDefinition)) {
        if (logger.isDebugEnabled()) {
            logger.debug("Overriding bean definition for bean '" +
beanName +
                    "' with a different definition: replacing [" +
existingDefinition +
                    "] with [" + beanDefinition + "]");
        }
    } else {
        if (logger.isTraceEnabled()) {
            logger.trace("Overriding bean definition for bean '" + beanName
+
                    "' with an equivalent definition: replacing [" +
existingDefinition +
                    "] with [" + beanDefinition + "]");
        }
    }
```

```
    this.beanDefinitionMap.put(beanName, beanDefinition);
```

上述截取部分为实现了对于beanDefinition的注册，其实可以看到，它这段代码会先用你当前的想要注册进来的BeanDefinition从beanDefinitonMap中获取一下，假如获取到了，说明已经存在，这个时候需要判断一下，当前你想要进行覆盖的beandDefinition的role等级是否和已经存在的role等级发生了冲突，并且做出相对应的提示，但是并没有做什么实质性的动作。

3. setDependsOn，这个方法简单来说是来指定Bean的加载先后顺序的，可以来看一个实例：

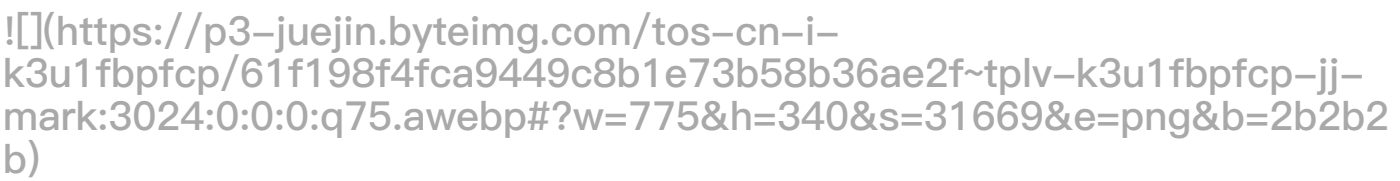
```
public class A {
public A() {
System.out.println(”I an A,init or create on ...”);
}
}
public class B {
public B(){
System.out.println(”I an B,init or create on ...”);
}
}
public class Start {
public static void main(String[] args) {
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
context.register(A.class, B.class);
context.refresh();
}
}
```

启动之后的结果：先是A的无参构造器被加载，然后再是B的无参构造器被加载。

```
@DependsOn(”b”)
public class A {
public A() {
System.out.println(”I an A,init or create on ...”);
```

```
}
}
public class B {
public B(){
System.out.println(”I an B,init or create on ...”);
}
}
public class Start {
public static void main(String[] args) {
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
context.register(A.class, B.class);
context.refresh();
}
}
```

当我们在A类上添加了注解 **@DependsOn(”b”)，** 结果如下：我们发现加载的顺序会发生变化，A类后于B类加载。



那么 **@DependsOn**注解作用就来指定某一个类会在某一个类的后面被加载。

4. void setFactoryMethodName(@Nullable String factoryMethodName);void setFactoryBeanName(@Nullable String factoryBeanName);

这两个方法放在一起说，原因是它俩的作用实际上差不多，是用来指定，我实例化一个Bean的是指定实例的方法，啥意思呢？就我们都知道spring是通过反射的方式来得到一个类的全部信息，那么反射的基础是会获取到一个类的构造器来实例化的，所以当你现在想要实例化一个类，但是并不想要是使用该类默认的构造器来实例化，这个时候你就可以指定它的实例化方法。

5. ConstructorArgumentValues getConstructorArgumentValues();

该方法的作用是推断构造方法，较为复杂，简单的来讲当一个类中有多个构造

器的时候，spring会推断你到底使用那个构造器来对当该进行实例化。下面我们通过一些简单的例子来描述该方法：

```
public class C {
public C() {
System.out.println(”init create C”);
}
}
public class Start {
public static void main(String[] args) {
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
context.register(B.class, A.class);
GenericBeanDefinition genericBeanDefinition = new
GenericBeanDefinition();
genericBeanDefinition.setBeanClass(C.class);
context.registerBeanDefinition(”c”, genericBeanDefinition);
context.refresh();
}
}
```

上述的代码非常简单就是将一个名为C的类变为一个BeanDefinition，然后将其注册到spring容器中，这个时候在控制台输出的内容肯定是

”init create C”

但是我现在将C类改造一下，并且将启动类也改造一下：

```
public class C {
public C() {
System.out.println(”init create C”);
}
public C(A a){
System.out.println(”init create c string a”);
}
}
public class Start {
public static void main(String[] args) {
```

```
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
context.register(B.class, A.class);
GenericBeanDefinition genericBeanDefinition = new
GenericBeanDefinition();
genericBeanDefinition.setBeanClass(C.class);
genericBeanDefinition.setAutowireMode(AbstractBeanDefinition.AUTOWI
RE_CONSTRUCTOR);
context.registerBeanDefinition("c", genericBeanDefinition);
context.refresh();
}
}
```

这个时候spring会推断你要使用第二个有参数的构造器进行实例化C类。

```
....
init create c string a

BUILD SUCCESSFUL in 5s
46 actionable tasks: 2 executed, 44 up-to-date
```

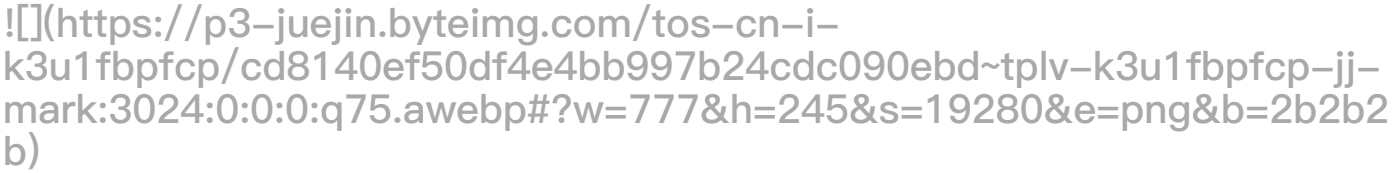
可以看到这个时候采用了第二个构造器对C类进行实例化，那么对于这个特殊的现象我们会在讲spring推断构造器的时候讲到。

6. void setInitMethodName(@Nullable String initMethodName); 和 String getInitMethodName();

没啥好说的就是指定一个beanDefinition在初始化的时候的方法：

```
public class C {
public C() {
System.out.println("init create C");
}
public void initOK(){
System.out.println("init c.....");
}
```

```
}
public class Start {
public static void main(String[] args) {
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
context.register(B.class, A.class);
GenericBeanDefinition genericBeanDefinition = new
GenericBeanDefinition();
genericBeanDefinition.setBeanClass(C.class);
context.registerBeanDefinition("c", genericBeanDefinition);
genericBeanDefinition.setInitMethodName("initOK");
context.refresh();
}
}
```



7. void setDestroyMethodName(@Nullable String destroyMethodName);
和 String getDestroyMethodName();

没啥好和上面的初始化方法一样也是指定一个销毁方法。

AttributeAccessorSupport
==========

这个类是对AttributeAccessor这个接口的实现。

```
public abstract class AttributeAccessorSupport implements
AttributeAccessor, Serializable {

/** Map with String keys and Object values. */
private final Map<String, Object> attributes = new LinkedHashMap<>();

//往map当中put了一个k/v 如果传一个为null的value等同于remove的操作
public void setAttribute(String name, @Nullable Object value) {
```

```
    Assert.notNull(name, "Name must not be null");
    if (value != null) {
        this.attributes.put(name, value);
    }
    else {
        removeAttribute(name);
    }
}

//map.get（key）
@Override
@Nullable
public Object getAttribute(String name) {
    Assert.notNull(name, "Name must not be null");
    return this.attributes.get(name);
}

//map.remove（key）
@Override
@Nullable
public Object removeAttribute(String name) {
    Assert.notNull(name, "Name must not be null");
    return this.attributes.remove(name);
}

//判断一个key是否存在
@Override
public boolean hasAttribute(String name) {
    Assert.notNull(name, "Name must not be null");
    return this.attributes.containsKey(name);
}

//返回所有的key
@Override
public String[] attributeNames() {
    return StringUtils.toStringArray(this.attributes.keySet());
}

/**
 * Copy the attributes from the supplied AttributeAccessor to this
accessor.
 * @param source the AttributeAccessor to copy from
 */
protected void copyAttributesFrom(AttributeAccessor source) {
    Assert.notNull(source, "Source must not be null");
    String[] attributeNames = source.attributeNames();
    for (String attributeName : attributeNames) {
        setAttribute(attributeName, source.getAttribute(attributeName));
```

```
    }
}

@Override
public boolean equals(@Nullable Object other) {
    return (this == other || (other instanceof AttributeAccessorSupport &&
            this.attributes.equals(((AttributeAccessorSupport)
other).attributes)));
}

@Override
public int hashCode() {
    return this.attributes.hashCode();
}
```

BeanMetadataAttributeAccessor
==========

继承了AttributeAccessorSupport并且实现了BeanMetadataElement接口，主要重现了AttributeAccessorSupport当中的方法为，进行了简单的封装。

AbstractBeanDefinition
==========

AbstractBeanDefinition实现了beandefinition接口中定义的各种api，并定义了一系列的常量属性，这些常量会直接影响到 Spring 实例化 Bean 时的策略，并且提供了对beanDefinition的拷贝的能力，下面我们开始对该类中的重要的方法一个一个进行解析：

AbstracrtBeanDefinition中的字段解析

```
```
标注当前的BeanDefinition的注入模型为?

```
// 手动装配
public static final int AUTOWIRE_NO =
AutowireCapableBeanFactory.AUTOWIRE_NO;
// 设置自动装配的模型
public static final int AUTOWIRE_BY_NAME =
```

```
AutowireCapableBeanFactory.AUTOWIRE_BY_NAME;
public static final int AUTOWIRE_BY_TYPE =
AutowireCapableBeanFactory.AUTOWIRE_BY_TYPE;
public static final int AUTOWIRE_CONSTRUCTOR =
AutowireCapableBeanFactory.AUTOWIRE_CONSTRUCTOR;

//首先尝试使用constructor进行自动装配。如果失败，再尝试使用byType进行自动装配
@Deprecated
public static final int AUTOWIRE_AUTODETECT =
AutowireCapableBeanFactory.AUTOWIRE_AUTODETECT;

```

```
private int autowireMode = AUTOWIRE_NO;

```

```
标注当前的BeanDefinition的检查模型为?
/**
 * 不检查
 */
public static final int DEPENDENCY_CHECK_NONE = 0;

/**
 * 检查对象依赖
 */
public static final int DEPENDENCY_CHECK_OBJECTS = 1;

/**
 * 检查原始类型:
 public static boolean isSimpleValueType(Class<?> type) {
return (Void.class != type
&& void.class != type
&& (ClassUtils.isPrimitiveOrWrapper(type)
|| Enum.class.isAssignableFrom(type)
|| CharSequence.class.isAssignableFrom(type)
|| Number.class.isAssignableFrom(type)
|| Date.class.isAssignableFrom(type)
|| Temporal.class.isAssignableFrom(type)
|| URI.class == type
|| URL.class == type
|| Locale.class == type
|| Class.class == type));
```

```
}
 */
public static final int DEPENDENCY_CHECK_SIMPLE = 2;

/**
 * 检查所有
 */
public static final int DEPENDENCY_CHECK_ALL = 3;
```

```
private int dependencyCheck = DEPENDENCY_CHECK_NONE;
```

```
@Nullable
private volatile Object beanClass;
```

```
@Nullable
private String scope = SCOPE_DEFAULT;
```

```
private boolean abstractFlag = false;
```

```
@Nullable
private Boolean lazyInit;
```

```
@Nullable
private String[] dependsOn;
```

```

```
private boolean primary = false;
```

```

```
@Nullable
private Supplier<?> instanceSupplier;
```

```

```
@Nullable
private String factoryBeanName;
```

```

```
@Nullable
private String factoryMethodName;
```

```

```
@Nullable
private ConstructorArgumentValues constructorArgumentValues;
```

```

```
@Nullable
private MutablePropertyValues propertyValues;
```

```

```
private MethodOverrides methodOverrides = new MethodOverrides();
```
```

```

```
@Nullable
private String initMethodName;
@Nullable
private String destroyMethodName;
```

```

```
private boolean enforceInitMethod = true;
private boolean enforceDestroyMethod = true;
// 如下面的代码所示:
// 初始化方法:
public void initMethodClass(){
    System.out.println("invoke custom method init a");
}
// 启动类
public class Start {
public static void main(String[] args) {
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
GenericBeanDefinition genericBeanDefinition = new
GenericBeanDefinition();
genericBeanDefinition.setBeanClass(A.class);
genericBeanDefinition.setInitMethodName("initMethodClassa");
genericBeanDefinition.setEnforceInitMethod(true); // 我们这里设置了强制
执行
context.registerBeanDefinition("a", genericBeanDefinition);
context.refresh();
}
}
// 执行之后spring由于找不到你所说的初始化方法但是你又说要强制执行,那么
它就会报错，报错信息如下:
Exception in thread "main"
org.springframework.beans.factory.BeanCreationException:
Error creating bean with name 'a': Invocation of init method failed;
nested exception is
org.springframework.beans.factory.support.BeanDefinitionValidationExce
ption:
Could not find an init method named 'initMethodClassa' on bean with
name 'a'
```

```
```

```
private boolean synthetic = false;
```

```
private int role = BeanDefinition.ROLE_APPLICATION;
```

```
@Nullable
private String description;
```

```
@Nullable
private Resource resource;
```

AbstractBeanDefinition中的方法和构造器

挑一些较为重要的方法来讲，不太重要的没必要来讲。

```
/**
 * Create a new AbstractBeanDefinition with default settings.
 */
protected AbstractBeanDefinition() {
    this(null, null);
}
```

```
/**
 * Create a new AbstractBeanDefinition with the given
```

```
 * constructor argument values and property values.
 */
protected AbstractBeanDefinition(@Nullable ConstructorArgumentValues
cargs, @Nullable MutablePropertyValues pvs) {
    this.constructorArgumentValues = cargs;
    this.propertyValues = pvs;
}
```

```
/**
 * Create a new AbstractBeanDefinition as a deep copy of the given
 * bean definition.
 * @param original the original bean definition to copy from
 */
protected AbstractBeanDefinition(BeanDefinition original) {
    setParentName(original.getParentName());
    setBeanClassName(original.getBeanClassName());
    setScope(original.getScope());
    setAbstract(original.isAbstract());
    setFactoryBeanName(original.getFactoryBeanName());
    setFactoryMethodName(original.getFactoryMethodName());
    setRole(original.getRole());
    setSource(original.getSource());
    copyAttributesFrom(original);

    if (original instanceof AbstractBeanDefinition) {
        AbstractBeanDefinition originalAbd = (AbstractBeanDefinition)
original;
        if (originalAbd.hasBeanClass()) {
            setBeanClass(originalAbd.getBeanClass());
        }
        if (originalAbd.hasConstructorArgumentValues()) {
            setConstructorArgumentValues(new
ConstructorArgumentValues(original.getConstructorArgumentValues()));
        }
        if (originalAbd.hasPropertyValues()) {
            setPropertyValues(new
MutablePropertyValues(original.getPropertyValues()));
        }
        if (originalAbd.hasMethodOverrides()) {
            setMethodOverrides(new
MethodOverrides(originalAbd.getMethodOverrides()));
        }
        Boolean lazyInit = originalAbd.getLazyInit();
        if (lazyInit != null) {
```

```
            setLazyInit(lazyInit);
        }
        setAutowireMode(originalAbd.getAutowireMode());
        setDependencyCheck(originalAbd.getDependencyCheck());
        setDependsOn(originalAbd.getDependsOn());
        setAutowireCandidate(originalAbd.isAutowireCandidate());
        setPrimary(originalAbd.isPrimary());
        copyQualifiersFrom(originalAbd);
        setInstanceSupplier(originalAbd.getInstanceSupplier());
setNonPublicAccessAllowed(originalAbd.isNonPublicAccessAllowed());
setLenientConstructorResolution(originalAbd.isLenientConstructorResolu
tion());
        setInitMethodName(originalAbd.getInitMethodName());
        setEnforceInitMethod(originalAbd.isEnforceInitMethod());
        setDestroyMethodName(originalAbd.getDestroyMethodName());
        setEnforceDestroyMethod(originalAbd.isEnforceDestroyMethod());
        setSynthetic(originalAbd.isSynthetic());
        setResource(originalAbd.getResource());
    }
    else {
        setConstructorArgumentValues(new
ConstructorArgumentValues(original.getConstructorArgumentValues()));
        setPropertyValues(new
MutablePropertyValues(original.getPropertyValues()));
        setLazyInit(original.isLazyInit());
        setResourceDescription(original.getResourceDescription());
    }
}

```

1. 解释一下，为什么说上述代码中AbstractBeanDefinition的该构造器是Copy，它Copy啥呢?
2. 是这样的，我们之前说过如果两个BeanDefinition是一个父子关系，或者说一个BeanDefinition最后会变为一个RootBeanDefiniton，来描述一个最终的BeanDefinition，那么这个构造器写的这么复杂它到底是拿来干啥用的呢?
3. 还记得在对beanDefinition进行合并的时候会创建一个RootBeanDefinition吗？看代码：

```
protected RootBeanDefinition getMergedBeanDefinition(String
beanName, BeanDefinition bd, @Nullable BeanDefinition containingBd)
throws BeanDefinitionStoreException {
    synchronized (this.mergedBeanDefinitions) {
        RootBeanDefinition mbd = null;
        RootBeanDefinition previous = null;
```

```
        // Check with full lock now in order to enforce the same merged
instance.
        if (containingBd == null) {
            mbd = this.mergedBeanDefinitions.get(beanName);
        }
        if (mbd == null || mbd.stale) {
            //表示是一个普通类或者是一个父类
            previous = mbd;
            if (bd.getParentName() == null) {
                // Use copy of given root bean definition.
                if (bd instanceof RootBeanDefinition) {
                    //判断当前的bd的类型是否是RootBeanDefinition
                    mbd = ((RootBeanDefinition) bd).cloneBeanDefinition();
                } else {
                    //如果不是rootBeanDefinition创建一个rootBeanDefinition
                    //解释一下为什么不是RootBeanDefinition也会创建一个
RootBeanDefinition:
                    //因为RootBeanDefinition相当于是beanDefinition的顶级父类
不管怎么样都可以被放入到BeanDefinition或者是
MergeBeanDefinitionMap中
                    mbd = new RootBeanDefinition(bd);
                }
            }
```

4. 看到在15~22行中的两个逻辑分支了嘛，判断如果当前的BeanDefinition是一个RootBeanDefinition的话就指定调用cloneBeanDefinition()克隆beanDefinition方法对当前的BeanDefinition进行克隆：

```
@Override
public RootBeanDefinition cloneBeanDefinition() {
    //this:代表的是原始的bd(从bdMap中获取的bd)
    return new RootBeanDefinition(this);
}
```

5. 该克隆相当于自己克隆自己，我们再看RootBeanDefiniton中看看它的构造器：

```
public RootBeanDefinition(RootBeanDefinition original) {
```

```
    super(original);
    this.decoratedDefinition = original.decoratedDefinition;
    this.qualifiedElement = original.qualifiedElement;
    this.allowCaching = original.allowCaching;
    this.isFactoryMethodUnique = original.isFactoryMethodUnique;
    this.targetType = original.targetType;
    this.factoryMethodToIntrospect = original.factoryMethodToIntrospect;
}
```

6. 再看看super方法是啥?

这里就不粘过来了，其实就是AbstractBeanDefinition中当前咱们聊的构造器，它会把当前的beanDefinition中的信息构建出一个BeanDefintion

7. 然后对于else中的：

```
mbd = new RootBeanDefinition(bd);
```

8. 其实也一样到最后都会去调用AbstractBeanDefinition的构造器完成BeanDefinition的拷贝和创建。

```
/**
 * Override settings in this bean definition (presumably a copied parent
 * from a parent-child inheritance relationship) from the given bean
 * definition (presumably the child).
 * <ul>
 * <li>Will override beanClass if specified in the given bean definition.
 * <li>Will always take {@code abstract}, {@code scope},
 * {@code lazyInit}, {@code autowireMode}, {@code dependencyCheck},
 * and {@code dependsOn} from the given bean definition.
 * <li>Will add {@code constructorArgumentValues}, {@code
propertyValues},
 * {@code methodOverrides} from the given bean definition to existing
ones.
 * <li>Will override {@code factoryBeanName}, {@code
factoryMethodName},
```

```
 * {@code initMethodName}, and {@code destroyMethodName} if
specified
 * in the given bean definition.
 * </ul>
 */
public void overrideFrom(BeanDefinition other) {
    if (StringUtils.hasLength(other.getBeanClassName())) {
        setBeanClassName(other.getBeanClassName());
    }
    if (StringUtils.hasLength(other.getScope())) {
        setScope(other.getScope());
    }
    setAbstract(other.isAbstract());
    if (StringUtils.hasLength(other.getFactoryBeanName())) {
        setFactoryBeanName(other.getFactoryBeanName());
    }
    if (StringUtils.hasLength(other.getFactoryMethodName())) {
        setFactoryMethodName(other.getFactoryMethodName());
    }
    setRole(other.getRole());
    setSource(other.getSource());
    copyAttributesFrom(other);

    if (other instanceof AbstractBeanDefinition) {
        AbstractBeanDefinition otherAbd = (AbstractBeanDefinition) other;
        if (otherAbd.hasBeanClass()) {
            setBeanClass(otherAbd.getBeanClass());
        }
        if (otherAbd.hasConstructorArgumentValues()) {
getConstructorArgumentValues().addArgumentValues(other.getConstruct
orArgumentValues());
        }
        if (otherAbd.hasPropertyValues()) {
getPropertyValues().addPropertyValues(other.getPropertyValues());
}
        if (otherAbd.hasMethodOverrides()) {
getMethodOverrides().addOverrides(otherAbd.getMethodOverrides());
  }
        Boolean lazyInit = otherAbd.getLazyInit();
        if (lazyInit != null) {
            setLazyInit(lazyInit);
        }
        setAutowireMode(otherAbd.getAutowireMode());
        setDependencyCheck(otherAbd.getDependencyCheck());
        setDependsOn(otherAbd.getDependsOn());
        setAutowireCandidate(otherAbd.isAutowireCandidate());
        setPrimary(otherAbd.isPrimary());
        copyQualifiersFrom(otherAbd);
```

```
        setInstanceSupplier(otherAbd.getInstanceSupplier());
setNonPublicAccessAllowed(otherAbd.isNonPublicAccessAllowed());
setLenientConstructorResolution(otherAbd.isLenientConstructorResolutio
n());
        if (otherAbd.getInitMethodName() != null) {
            setInitMethodName(otherAbd.getInitMethodName());
            setEnforceInitMethod(otherAbd.isEnforceInitMethod());
        }
        if (otherAbd.getDestroyMethodName() != null) {
            setDestroyMethodName(otherAbd.getDestroyMethodName());
            setEnforceDestroyMethod(otherAbd.isEnforceDestroyMethod());
        }
        setSynthetic(otherAbd.isSynthetic());
        setResource(otherAbd.getResource());
    }
    else {
getConstructorArgumentValues().addArgumentValues(other.getConstruct
orArgumentValues());
        getPropertyValues().addPropertyValues(other.getPropertyValues());
        setLazyInit(other.isLazyInit());
        setResourceDescription(other.getResourceDescription());
    }
}

```


```
/**
 * Apply the provided default values to this bean.
 * @param defaults the default settings to apply
 * @since 2.5
 */
public void applyDefaults(BeanDefinitionDefaults defaults) {
    Boolean lazyInit = defaults.getLazyInit();
    if (lazyInit != null) {
        setLazyInit(lazyInit);
    }
    setAutowireMode(defaults.getAutowireMode());
    setDependencyCheck(defaults.getDependencyCheck());
    setInitMethodName(defaults.getInitMethodName());
    setEnforceInitMethod(false);
    setDestroyMethodName(defaults.getDestroyMethodName());
    setEnforceDestroyMethod(false);
}

```

```
/**
 * Specify the bean class name of this bean definition.
 */
@Override
public void setBeanClassName(@Nullable String beanClassName) {
    this.beanClass = beanClassName;
}

```

```
/**
 * Return the current bean class name of this bean definition.
 */
@Override
@Nullable
public String getBeanClassName() {
    Object beanClassObject = this.beanClass;
    if (beanClassObject instanceof Class) {
        return ((Class<?>) beanClassObject).getName();
    }
    else {
        return (String) beanClassObject;
    }
}

```

```
/**
 * Specify the class for this bean.
 * @see #setBeanClassName(String)
 */
public void setBeanClass(@Nullable Class<?> beanClass) {
    this.beanClass = beanClass;
}

```

```
public Class<?> getBeanClass() throws IllegalStateException {
    Object beanClassObject = this.beanClass;
    if (beanClassObject == null) {
```

```
        throw new IllegalStateException("No bean class specified on bean
definition");
    }
    if (!(beanClassObject instanceof Class)) {
        throw new IllegalStateException(
                "Bean class name [" + beanClassObject + "] has not been
resolved into an actual Class");
    }
    return (Class<?>) beanClassObject;
}
```

关于bean指定初始化方法注意的点
---

1. bean有三种初始化方法：

* 一：通过注解@PostConstruct注解来指定当前 Bean的初始化方法，这里稍微注意一下该注解的提供方不是spring，而是java原生的注解，来自javax.anntotation包下。

```
@PostConstruct
public void annotationMethod(){
    System.out.println("invoke annotation init a");
}
```

* 二：是通过实现spring提供的InitializingBean接口，来重写该接口的afterPropertiesSet()方法来描述你想要指定该的类的初始化方法。

```
public class A implements InitializingBean {
    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("interface initMethod init A");
    }
}
```

* 三：通过指定初始方法来创建：通过在需要指定初始化方法的类中提供一个自己指定的初始化方法，并且在注册这个Bean之前创建该bean的BeanDefinition的时候指定初始化方法，比如下面的案例中我们指定了initMethodClass()方法来作为A类的初始化方法。

```
// A类中自定义的初始化方法
public void initMethodClass(){
System.out.println(”invoke custom method init a”);
}
public class Start {
public static void main(String[] args) {
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
GenericBeanDefinition genericBeanDefinition = new
GenericBeanDefinition();
genericBeanDefinition.setBeanClass(A.class);
genericBeanDefinition.setInitMethodName(”initMethodClass”);
context.registerBeanDefinition(”a”, genericBeanDefinition);
context.refresh();
}
}
```

* 这里说一下，上述介绍的三种对于一个bean指定初始化方法顺序(也就是三种初始化bean的方式同时存在)，他们的执行方式按照上述的介绍顺序来：

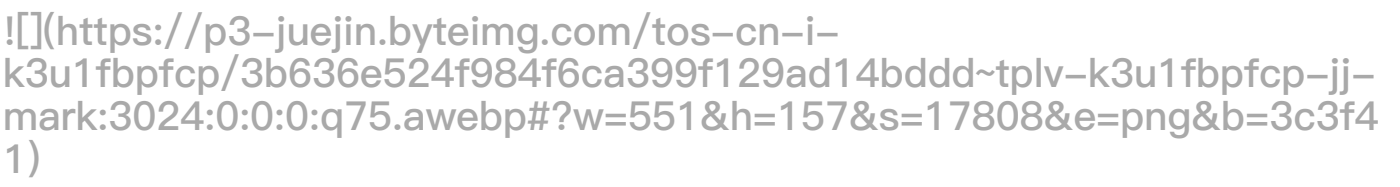
2. 下面我们看看对于上述三种bean的初始化方法spring会放在BeanDefinition的哪里：

* @PostConstruct：注解来指定某个类的初始化方法是有些特殊的，他会单独的存放在一个set集合中：

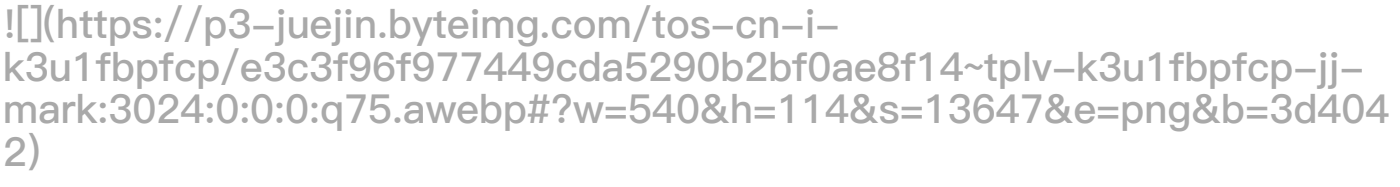
```
```

```
@Nullable
private Set<String> externallyManagedInitMethods;
```

* InitializingBean接口：对于实现InitializingBean接口的方法来说spring是不会进行记录的，原因很简单，因为这是spring内置的对于bean自定义初始化方法的实现，并且对于单个bean来说只会有一个初始化方法，也只能有一个初始化方法。所以spring在执行Bean生命周期中的初始化回调的时候就会发现当前的类，比如说A类就实现InitializingBean接口，然后就会指定你写的逻辑来初始化该bean。



* 指定初始方法：这个时候我们可以看到initMethodName字段上面就是咱们说的自己实现该类的初始化方法并且在注册该Bean的时候指定初始化方法，和上面实现了InitialzingBean接口不一样吧。



RootBeanDefinition
==========

```
@Nullable
private BeanDefinitionHolder decoratedDefinition;
```

```
@Nullable
private AnnotatedElement qualifiedElement;
```

```
boolean allowCaching = true;
```

```
boolean isFactoryMethodUnique = false;
```

```
volatile Class<?> resolvedTargetType;
```

```
final Object constructorArgumentLock = new Object();
```

```
Executable resolvedConstructorOrFactoryMethod;
```

```
boolean constructorArgumentsResolved = false;
```

```
Object[] resolvedConstructorArguments;
```

```
Object[] preparedConstructorArguments;
```

```
final Object postProcessingLock = new Object();
```

```
boolean postProcessed = false;
```

```
volatile Boolean beforeInstantiationResolved;
```

```
private Set<Member> externallyManagedConfigMembers;
```

```
private Set<String> externallyManagedInitMethods;
```

```
private Set<String> externallyManagedDestroyMethods;
```

ChildBeanDefinition
==========

继承 AbstractBeanDefinition,用来描述一个子beandefinition，**不可以单独存在**，必须依赖一个父 BeanDetintion，构造 ChildBeanDefinition 时，通

过构造方法传入父 BeanDetintion 的名称或通过 setParentName 设置父名称。它可以从父类继承方法参数、属性值，并可以重写父类的方法，同时也可以增加新的属性或者方法。若重新定义 init 方法，destroy 方法或者静态工厂方法，ChildBeanDefinition 会重写父类的设置。

GenericBeanDefinition
==========

从ChildBeanDefinition的javadoc当中可以知道；这个beandefinition是2.5版本后用来替代ChilBeanDenitiontion的；可以单独存在，也可以用来作为子类beandefinition。

AnnotatedBeanDefinition
==========

AnnotatedBeanDefinition 是 BeanDefinition 子接口之一，该接口扩展了 BeanDefinition 的功能，其用来操作注解元数据。一般情况下，通过注解方式得到的 Bean（@Component、@Bean），其 BeanDefinition 类型都是该接口的实现类。

```
// 获得当前 Bean 的注解元数据
AnnotationMetadata getMetadata();

// 获得当前 Bean 的工厂方法上的元数据
@Nullable
MethodMetadata getFactoryMethodMetadata();
```

原文链接: https://juejin.cn/post/7383029698115158067