Spring系列-2 Bean的生命周期

原创 Ewen Seong 已于 2024-04-05 16:42:16 修改 阅读量1.6k 收藏 6 点赞数 2
分类专栏： Spring系列 文章标签： spring java

背景：

作为Spring系列的第二篇，本文结合容器的启动流程介绍**单例Bean的生命周期**，包括Bean对象的创建、 属性设置 、初始化、使用、销毁等阶段；
会介绍Spring用于操作Bean或者BeanDefinition的相关扩展接口。
文章重心在于介绍整个Bean生命周期，不拘泥于每个阶段的细节；因此本文中会常见到“主线逻辑”这个关键词，请读者不要对此反感。

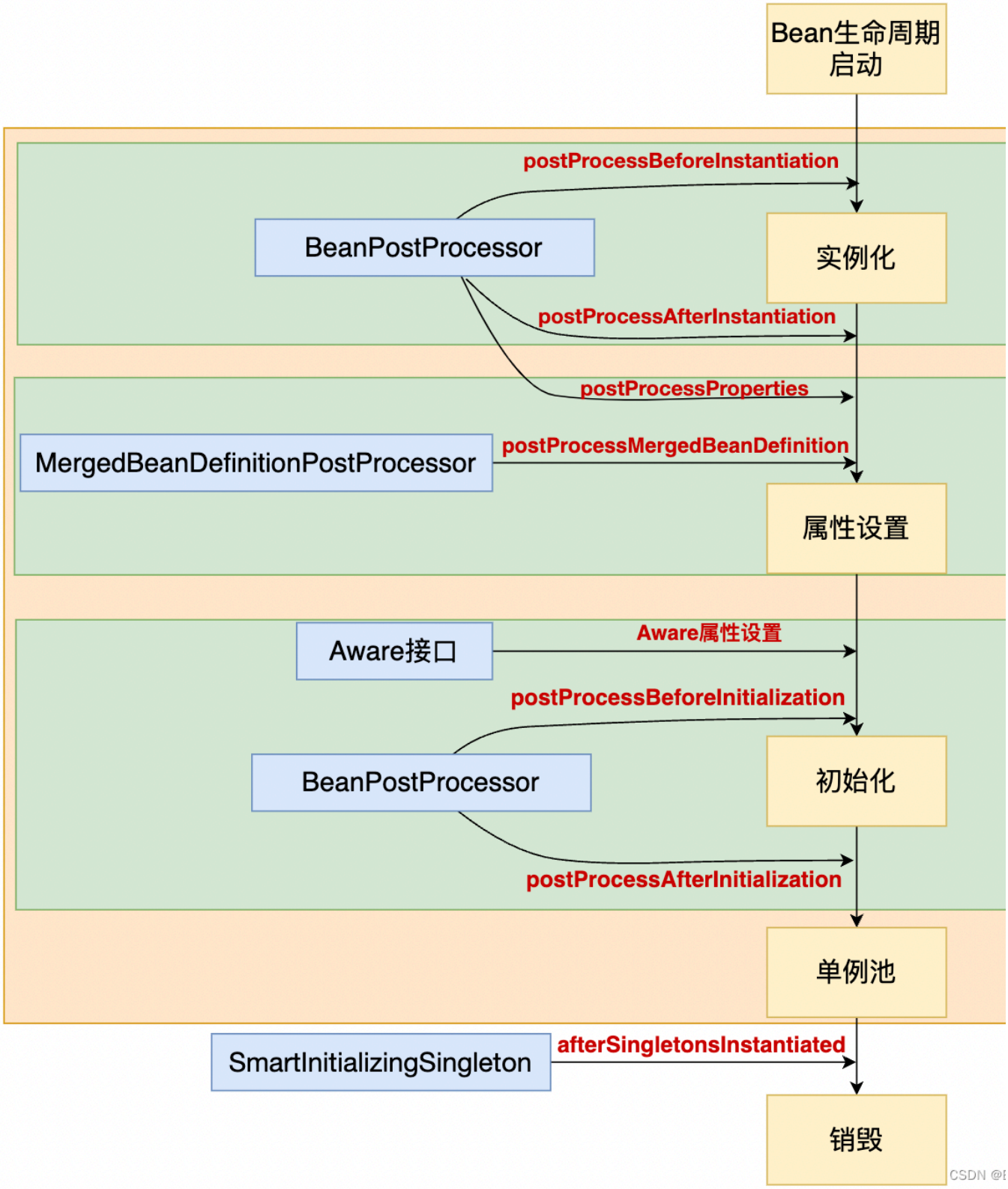
> 容器的启动流程请参考：Spring系列-1 启动流程

1. Bean的生命周期

本文介绍单例Bean的生命周期，本文后续提到中Bean对象默认指代单例Bean。

1.1 流程图

每个Bean的生命周期都包括：实例化、属性设置、初始化、入单例池、销毁等阶段，整个流程可表示为：



上图中Bean的生命周期主流程可以分为4个部分：

[1] 实例化阶段

实例化阶段的核心目的是生成Bean对象，过程包括构造函数的推断与选择、通过反射调用目标构造函数实例化Bean对象。

Spring在框架中引入了InstantiationAwareBeanPostProcessor接口，提供了操作对象实例化的切入点；用户可以继承InstantiationAwareBeanPostPro自定义实例化对象过程。

[2] 属性设置阶段

属性设置阶段的核心任务是对已实例化的Bean对象根据配置进行属性设置。

InstantiationAwareBeanPostProcessor接口的postProcessProperties方法也提供给了用户干扰属性设置的能力。

MergedBeanDefinitionPostProcessor接口提供了操作Beandefinition信息的能力，也可以基于Beandefinition提取相关信息；如AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor在实现依赖注入时使用MergedBeanDefinitionPostProcessor进行的数据准备工作。

[3] 初始化阶段
初始化阶段是Spring框架为Bean对象自定义的一个阶段，目的在于当Bea对象完成实例化和属性注入后可以执行一些扩展方法。
该阶段按执行顺序包括：Aware接口的执行与相关Aware属性的设置、执行BeanPostProcessor的前置方法、执行初始化方法、执行BeanPostProcess
法。
其中，BeanPostProcessor的后置方法是依赖注入、AOP等实现原理(*后续Spring系列文章中会反复见到BeanPostProcessor的后置方法*)。
该阶段的核心逻辑是执行初始化方法，按照调用顺序包括：InitializingBean接口的afterPropertiesSet()方法 和
xml配置文件中通过Init-method属性指定的初始化方法。

[4] 销毁阶段
当容器被注销时，会销毁单例池中的Bean对象，此时进入Bean对象生命周期的销毁阶段，该阶段按照执行顺序包括：DisposableBean的destroy()方
文件中通过destory-method属性指定的方法。

1.2 Bean生命周期方法

Bean生命周期方法是Bean对象级别的方法，即每种类型的Bean有自己的方法(仅作用于自己)，区别于Spring系统级别的方法(如BeanpostProcessor)
Bean对象。

1.2.1 Aware接口

Aware接口逻辑较为简单，用于向Bean对象种设置某种类型的属性，如下是BeanFactoryAware接口定义：

```
public interface BeanFactoryAware extends Aware {
    void setBeanFactory(BeanFactory beanFactory) throws BeansException;
}
```

案例如下：

```
public class ComponentA implements BeanFactoryAware {
    private BeanFactory beanFactory;

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        this.beanFactory = beanFactory;
    }

    public Object getBean(String beanName) {
        return beanFactory.getBean(beanName);
    }
}
```

1.2.2 InitializingBean与DisposableBean

InitializingBean接口用于指定初始化逻辑，DisposableBean用于指定销毁逻辑：

```
public interface InitializingBean {
    void afterPropertiesSet() throws Exception;
}

public interface DisposableBean {
    void destroy() throws Exception;
}
```

1.2.3 SmartInitializingSingleton

```
public interface SmartInitializingSingleton {
    void afterSingletonsInstantiated();
}
```

SmartInitializingSingleton在Spring容器完成所有非延迟单例Bean的注入到IOC容器后执行。
用户可通过SmartInitializingSingleton定义Bean注入到IOC后的自定义过程。

1.3 钩子函数

Spring提供这些钩子函数是为了提高Spring框架的扩展能力，基于Spring构造Bean对象的主流程衍生出新功能
如依赖注入或者AOP代理等。

钩子函数作为Spring系统级别的方法作用于所有Bean对象。

1.3.1 MergedBeanDefinitionPostProcessor

```
public interface MergedBeanDefinitionPostProcessor extends BeanPostProcessor {
    void postProcessMergedBeanDefinition(RootBeanDefinition beanDefinition, Class<?> beanType, String beanName);
}
```

MergedBeanDefinitionPostProcessor接口除去因继承BeanPostProcessor引入的接口外，只有一个postProcessMergedBeanDefinition方法：通过beanDefinition，可修改beanDefinition以影响后续的属性设置；也可以从beanDefinition中提取信息做前置准备。
值得注意的是该接口不具备直接操作Bean对象的能力，原因如下：(1) 入参中没有携带Bean对象; (2) 此时Bean对象虽然已经被实例化，但是尚未加入中, 即无法通过BeanFactory从IOC中获取该Bean对象。

1.3.2 BeanPostProcessor(BPP)

BeanPostProcessor中存在两个方法，postProcessBeforeInitialization和postProcessAfterInitialization：

```
public interface BeanPostProcessor {
    @Nullable
    default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }

    @Nullable
    default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }
}
```

从名称可以看出postProcessBeforeInitialization在初始化方法之前执行，postProcessAfterInitialization在初始化方法之后执行。
postProcessBeforeInitialization和postProcessAfterInitialization均提高了直接修改Bean对象的能力：方法的入参上Bean对象和beanName, 返回值为修Bean对象。需要注意：二者的执行时机都在Bean对象已完成实例化与属性赋值后。
其中，postProcessAfterInitialization方法的扩展能力被Spring使用得淋漓尽致；依赖注入、AOP等功能都是基于该方法的扩展能力而实现。

1.3.3 InstantiationAwareBeanPostProcessor(IABPP)

InstantiationAwareBeanPostProcessor接口的定义如下：

```
public interface InstantiationAwareBeanPostProcessor extends BeanPostProcessor {
    @Nullable
    default Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException {
        return null;
    }

    default boolean postProcessAfterInstantiation(Object bean, String beanName) throws BeansException {
        return true;
    }

    @Nullable
    default PropertyValues postProcessProperties(PropertyValues pvs, Object bean, String beanName)
            throws BeansException {
        return null;
    }
}
// 除此之外，还有个postProcessPropertyValues方法，作用同postProcessProperties。
// 因被@Deprecated注解，本文选择进行忽略
```

InstantiationAwareBeanPostProcessor为扩展**实例化过程**提供了postProcessBeforeInstantiation和postProcessAfterInstantiation两个方法，为扩展**属性**提供了postProcessProperties方法。
基于InstantiationAwareBeanPostProcessor接口的设计目的，对该接口的说明需要结合Bean对象构建过程进行介绍，否则会失去其上下文意义(如同单句)。

postProcessBeforeInstantiation

postProcessBeforeInstantiation方法的入参为beanName和Bean对象的类型，用户可基于此构建Bean对象并返回，也可返回null(默认返回null)；当有时，将走**扩展流程**而不是按照Bean生命周期的主体流程来创建Bean对象。
创建Bean对象的createBean方法中主线逻辑如下：

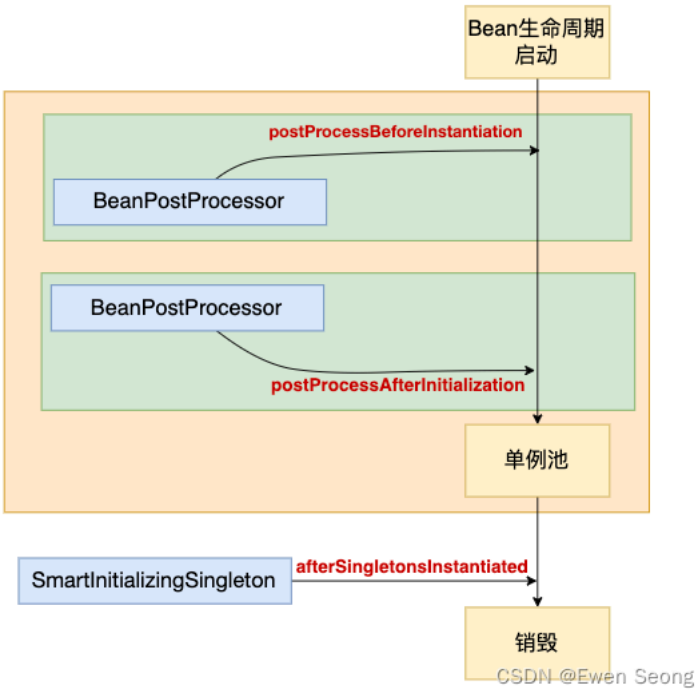
```
protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args)
    throws BeanCreationException {
    // ⚠：  省略无关逻辑与try-catch等代码...
    // ⚠：步骤1:调用resolveBeforeInstantiation方法构造Bean对象
    Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
    // 如果bean对象不为null，直接返回
    if (bean != null) {
        return bean;
    }
    // ⚠：步骤2:调用doCreateBean方法构造Bean对象并返回
    Object beanInstance = doCreateBean(beanName, mbdToUse, args);
    return beanInstance;
}
```

这里我们关注的重点在于 `resolveBeforeInstantiation(beanName, mbdToUse)` 方法，该方法的主线逻辑如下：

```
// ... 省略if分支，突出主线逻辑
protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {
    Object bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
    if (bean != null) {
        bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
    }
    return bean;
}
```

该方法逻辑较为简单，通过applyBeanPostProcessorsBeforeInstantiation调用IABPP的postProcessBeforeInstantiation方法构建Bean对象(**遍历IOC中象，并调用其postProcessBeforeInstantiation方法；返回一个非空的对象，否则返回空**)，如果为空则直接放回；否则通过applyBeanPostProcessorsAfterInitialization调用BPP的postProcessAfterInitialization方法操作Bean对象。

此时，Bean的构建过程如下所示：



因为，AOP和依赖注入的实现逻辑都在BPP的postProcessAfterInitialization方法，上述设计使得自定义的Bean对象仍然可以拥有Spring框架的依赖注功能。

postProcessAfterInstantiation和postProcessProperties

对postProcessAfterInstantiation和postProcessProperties二者的介绍放在一起是因为二者都发生在populate()方法中，可以从代码层面认为二者隶属于过程，populateBean方法的主线逻辑如下：

```
protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {
    // ⚠：1.调用postProcessAfterInstantiation
    for (InstantiationAwareBeanPostProcessor bp : getBeanPostProcessorCache().inst
        if (!bp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName))
            return;
```

```
        }
    }

    PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null);

    // ... pvs对象修改

    // ⚠️: 2.调用postProcessProperties
    for (InstantiationAwareBeanPostProcessor bp : getBeanPostProcessorCache().instantiationAware) {
        pvs = bp.postProcessProperties(pvs, bw.getWrappedInstance(), beanName);
    }

    // ... check

    // ⚠️: 3.根据pvs的信息对Bean对象设置属性
    if (pvs != null) {
        applyPropertyValues(beanName, mbd, bw, pvs);
    }
}
```

populateBean方法的核心功能是进行属性设置，步骤可分为如下三步：
[1] 调用postProcessAfterInstantiation;
[2] 根据配置信息构造pvs对象，调用IABPP对象的postProcessProperties方法修改pvs对象;
[3] 根据pvs的信息对Bean对象设置属性.

其中：步骤[1]中遍历IOC容器中的IABPP对象，并调用其postProcessAfterInstantiation方法，当所有的调用结果都返回true时，继续执行步骤[2]和步骤
性的设置；否则跳过步骤[2]和[3]，即不会执行属性设置流程.

2.案例介绍

略

3.原理

本章节基于Spring源码介绍Bean对象的构建过程。

3.1 触发时机

如Spring系列-1 启动流程文中介绍：在Spring容器启动之初先向IOC容器中注入Spring框架内置的组件对象、BeanFactory、BeanPostProcessor等Be
之后初始化所有的非懒加载的单例Bean对象。过程中，当遇到依赖的Bean对象时，优先构造被依赖的Bean对象(无论是否时懒加载Bean).
AbstractBeanFactory.refresh()方法：

```
public void refresh() throws BeansException, IllegalStateException {
    //...
    // Instantiate all remaining (non-lazy-init) singletons.
    finishBeanFactoryInitialization(beanFactory);
    //...
}

protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
    //...
    // Instantiate all remaining (non-lazy-init) singletons.
    beanFactory.preInstantiateSingletons();
}
```

preInstantiateSingletons方法：
DefaultListableBeanFactory类提供的preInstantiateSingletons方法会加载所有的非lazy单例Bean，代码如下所示：

```
public void preInstantiateSingletons() throws BeansException {
    //⚠️: 说明：为突出Bean的主线逻辑，省去了FactoryBean逻辑以及一些包装的内容。
    List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);

    // Trigger initialization of all non-lazy singleton beans...
    for (String beanName : beanNames) {
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
            getBean(beanName);
        }
    }
```

```
10
11
12
13      // Trigger post-initialization callback for all applicable beans...
14      for (String beanName : beanNames) {
15          Object singletonInstance = getSingleton(beanName);
16          if (singletonInstance instanceof SmartInitializingSingleton) {
17              SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton) singletonInstance;
18              smartSingleton.afterSingletonsInstantiated();
19          }
20      }
21  }
```

该方法对beanDefinitionNames进行了两次遍历：第一次遍历时对于非延迟的单例Bean调用 `getBean(beanName)` 方法构造Bean对象；第二次对于实现了SmartInitializingSingleton接口的Bean对象调用其afterSingletonsInstantiated方法，执行用户的自定义操作。

3.2 getBean和doGetBean

getBean方法：

该方法可以根据beanName获取Bean对象，如果对象不存在则创建该Bean对象。

```
1  public Object getBean(String name) throws BeansException {
2      // 以do开头的方法，一般是实际干活的
3      return doGetBean(name, null, null, false);
4  }
```

注意第二个参数是null，即调用doGetBean方法时，Class requiredType参数为null。

doGetBean方法：

doGetBean主线逻辑如下所示：

```
1  protected <T> T doGetBean( String name, @Nullable Class<T> requiredType, @Nullable Object[] args, boolean typeCheckOnly) {
2      String beanName = transformedBeanName(name);
3      Object beanInstance;
4
5      // ⚠1：parentBeanFactory.getBean(nameToLookup)
6
7      // ⚠2：checkMergedBeanDefinition(mbd, beanName, args);
8
9      // ⚠3：优先构造被依赖的Bean对象
10     String[] dependsOn = mbd.getDependsOn();
11     if (dependsOn != null) {
12         for (String dep : dependsOn) {
13             if (isDependent(beanName, dep)) {
14                 throw new BeanCreationException(mbd.getResourceDescription(), beanName,
15                         "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");
16             }
17             registerDependentBean(dep, beanName);
18             getBean(dep);
19         }
20     }
21
22     //  ⚠4：构造Bean对象
23     if (mbd.isSingleton()) {
24         sharedInstance = getSingleton(beanName, () -> {
25             return createBean(beanName, mbd, args);
26         });
27         beanInstance = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
28     } else if (mbd.isPrototype()) {
29         //...
30     } else {
31         //...
32     }
33
34     // ⚠5：类型校验&&适配
35     return adaptBeanInstance(name, beanInstance, requiredType);
36 }
```

doGetBean的主线逻辑可以分为如下五个步骤：

[1] 从父亲容器中获取Bean对象

如果当前容器中不存在beanName对应的Bean定义，Spring会尝试通过调用父容器的getBean方法获取Bean对象。

注意：父容器不能获取子容器中的Bean对象, 而子容器可以获取父容器中的Bean对象(因为子容器通过parentBeanFactory属性持有了父容器的引用)。

[2] 校验BeanDefinition

校验逻辑在checkMergedBeanDefinition方法中实现。

[3] 处理依赖关系

如果当前beanName对应的Bean对象存在依赖的Bean时，优先通过getBean方法处理被依赖的Bean对象。

如果存在相互依赖，则抛出BeanCreationException异常。

[4] 构造Bean对象

根据Bean的类型走不同的构造流程，当Bean为单例时，执行如下逻辑构造Bean对象：

```
sharedInstance = getSingleton(beanName, () -> createBean(beanName, mbd, args));
// getObjectForBeanInstance方法是为了适配FactoryBean类型的对象
beanInstance = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
```

上述lambda表达式用于传参给FactoryBean, 当执行该FactoryBean对象的getBean时会调用该表达式，因此

getSingleton方法的主线逻辑可以被整合为：

```
public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null) {
        beforeSingletonCreation(beanName);
        singletonObject = createBean(beanName, mbd, args);
        afterSingletonCreation(beanName);
        addSingleton(beanName, singletonObject);
    }
    return singletonObject;
}
```

首先尝试从单例池中根据beanName获取Bean对象，如果不为空——直接返回；否则调用 `createBean(beanName, mbd, args)` 构造Bean对象(前后环绕beforeSingletonCreation和afterSingletonCreation方法)。 `addSingleton(beanName, singletonObject)` 用于将创建好的对象加入到单例池中并清理缓存

[5] 类型校验和转换

adaptBeanInstance方法会根据requiredType确定是否执行类型校验和转换， 校验失败时抛出BeanNotOfRequiredTypeException异常：

```
<T> T adaptBeanInstance(String name, Object bean, @Nullable Class<?> requiredType) {
    if (requiredType != null && !requiredType.isInstance(bean)) {
        try {
            Object convertedBean = getTypeConverter().convertIfNecessary(bean, requiredType);
            if (convertedBean == null) {
                throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
            }
            return (T) convertedBean;
        } catch (TypeMismatchException ex) {
            throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
        }
    }
    return (T) bean;
}
```

当requiredType为null时，不经过任何处理直接返回。在上述getBean调用doGetBean方法时传递的requiredType为null, 因此在Spring容器初始化阶段对象不会经过类型校验和转换。

3.3 createBean和doCreateBean

createBean方法：

创建Bean对象的createBean方法中主线逻辑如下：

```
protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args)
    throws BeanCreationException {
    // ⚠：  省略无关逻辑与try-catch等代码...
    // ⚠：步骤1:调用resolveBeforeInstantiation方法构造Bean对象

```

```
    Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
    // 如果bean对象不为null，直接返回
    if (bean != null) {
        return bean;
    }
    // ⚠：步骤2:调用doCreateBean方法构造Bean对象并返回
    Object beanInstance = doCreateBean(beanName, mbdToUse, args);
    return beanInstance;
}
```

主线逻辑分为两个步骤：(1) 调用resolveBeforeInstantiation方法构造Bean对象和 (2) 调用doCreateBean方法构造Bean对象。其中resolveBeforeInsta
法在请参考 1.3.3 InstantiationAwareBeanPostProcessor 节中的内容。

doCreateBean方法：

该方法的主线逻辑如下所示：

```
// 省略无关逻辑
protected Object doCreateBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {
    // ...
    // ⚠:1.实例化Bean对象
    // Instantiate the bean.
    BeanWrapper instanceWrapper = createBeanInstance(beanName, mbd, args);
    Object bean = instanceWrapper.getWrappedInstance();

    // ⚠:2.将半成品对象加入三级缓存
    // Eagerly cache singletons to be able to resolve circular references
    // even when triggered by lifecycle interfaces like BeanFactoryAware.
    boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
            isSingletonCurrentlyInCreation(beanName));
    if (earlySingletonExposure) {
        addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
    }

    // ⚠:3.Bean对象的属性设置
    populateBean(beanName, mbd, instanceWrapper);
    // ⚠:4.Bean对象的初始化
    exposedObject = initializeBean(beanName, exposedObject, mbd);

    // ⚠:5.注册DisposableBean
    registerDisposableBeanIfNecessary(beanName, bean, mbd);

    // ⚠:6.返回对象
    return exposedObject;
}
```

doCreateBean实际承担了构造Bean对象的任务，包括：

[1] 调用createBeanInstance实例化Bean对象；
[2] 将半成品的Bean对象加入三级缓存；
[3] 对Bean对象进行属性设置；
[4] 对Bean对象执行初始化；
[5] 注册实现了DisposableBean接口的Bean对象；
[6] 返回构建完成的Bean对象。

3.4 实例化

createBeanInstance方法：

通过createBeanInstance实例化Bean对象，代码如下所示：

```
protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {
    // 1.校验Bean对象的字节码对象：必须为public且存在public的构造方法

    // 2.Supplier接口或者工厂方法不为空的，使用对应的构造流程构造Bean对象

    // 3.使用构造函数构造Bean对象 {
        //resolved表示构造函数是否已经解析完成；autowireNecessary表示是否需要自动装配
        boolean resolved = false;

```

```
        boolean autowireNecessary = false;

        if (args == null) {
            if (mbd.resolvedConstructorOrFactoryMethod != null) {
                resolved = true;
                autowireNecessary = mbd.constructorArgumentsResolved;
            }
        }
        if (resolved) {
            if (autowireNecessary) {
                return autowireConstructor(beanName, mbd, null, null);
            } else {
                return instantiateBean(beanName, mbd);
            }
        }

        Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);
        if (ctors != null || mbd.getResolvedAutowireMode() == AUTOWIRE_CONSTRUCTOR ||
                mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {
            return autowireConstructor(beanName, mbd, ctors, args);
        }
        return instantiateBean(beanName, mbd);
    }
}
```

该方法主线逻辑有以下三个步骤：
[1] 校验Bean对象的字节码对象：必须为public且存在public的构造方法;
[2] Supplier接口或者工厂方法不为空的，使用对应的构造流程构造Bean对象;
[3] 使用Bean的构造函数构造Bean对象。
其中第三步为本文感兴趣的部分，整体上看：根据Bean的具体情况选择调用 `autowireConstructor` 或者 `instantiateBean` 方法进行实例化；并引入了两个变量：**resolved**表示构造函数是否已经完成过解析；**autowireNecessary**表示是否调 `autowireConstructor` 方法进行实例化。
另外，BeanDefinition对象的constructorArgumentsResolved属性用于标记是否已进行过构造函数推断(true表示使用autowireConstructor方法实例化Bean，false表示使用instantiateBean方法)，是一种优化：当同一个BeanDefinition被多次用于构造Bean对象时，不需要执行重复的推断步骤；该属性针对的是多例类型的Bean对象，对单例Bean没有优化的意义。

`determineConstructorsFromBeanPostProcessors(beanClass, beanName)` 也是一个钩子函数，用于干预构造函数的选择：
如果用户自定义了SmartInstantiationAwareBeanPostProcessor实现类，且返回了通过 `determineConstructorsFromBeanPostProcessors(beanClass, be` 对应的bean确定了构造函数，则使用该构造函数实例化Bean对象。

因此，对于单例Bean对象上述逻辑可以简化为：

```
protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @Nullable Object[] args) {
    boolean resolved = false;
    boolean autowireNecessary = false;

    if (args == null) {
        if (mbd.resolvedConstructorOrFactoryMethod != null) {
            resolved = true;
        }
    }
    if (resolved) {
        if (autowireNecessary) {
            return autowireConstructor(beanName, mbd, null, null);
        } else {
            return instantiateBean(beanName, mbd);
        }
    }

    if (mbd.getResolvedAutowireMode() == AUTOWIRE_CONSTRUCTOR
        || mbd.hasConstructorArgumentValues()
        || !ObjectUtils.isEmpty(args)) {
        return autowireConstructor(beanName, mbd, null, args);
    }
    return instantiateBean(beanName, mbd);
}
```

其实，不看具体实现而仅从上述代码逻辑也可以推断出来： `instantiateBean(beanName, mbd)` 表示 `autowireConstructor(beanName, mbd, ctors, args)` 使用有参构造函数。

autowireConstructor方法封装了一个重要的概念，构造函数推断：

（1）当存在使用@Autowired注解的构造函数时，使用被@Autowired注解的构造函数实例化对象；

（2）当只有一个构造函数时，使用该构造函数；

（3）当存在多个构造函数且存在默认构造函数时，使用默认构造函数；

（4）当存在多个构造函数且不存在默认构造函数时，抛出异常。

3.5 populateBean

因介绍InstantiationAwareBeanPostProcessor而前置了该部分内容，请参考 1.3.3 InstantiationAwareBeanPostProcessor 节中的内容。

3.6 初始化

initializeBean方法：

该方法的源码如下：

```
protected Object initializeBean(String beanName, Object bean, @Nullable RootBeanDefinition mbd) {

    // ⚠：1.调用Aware接口
    invokeAwareMethods(beanName, bean);

    // ⚠：2.调用BPP的postProcessBeforeInitialization方法
    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
    }

    // ⚠：3.执行初始化方法
    invokeInitMethods(beanName, wrappedBean, mbd);

    // ⚠：4.调用BPP的postProcessAfterInitialization方法
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
    }

    // ⚠：5.调用BPP的postProcessBeforeInitialization方法
    return wrappedBean;
}
```

主线逻辑分为如下五个步骤：

(1) 调用Aware接口，进行Aware相关属性的设置；

(2) 调用BPP的postProcessBeforeInitialization方法；

(3) 执行初始化方法；

(4) 调用BPP的postProcessAfterInitialization方法；

(5) 调用BPP的postProcessBeforeInitialization方法。

invokeInitMethods方法：

```
protected void invokeInitMethods(String beanName, Object bean, @Nullable RootBeanDefinition mbd) {
    // ⚠：1.调用InitializingBean的afterPropertiesSet方法
    boolean isInitializingBean = (bean instanceof InitializingBean);
    if (isInitializingBean && (mbd == null || !mbd.isExternallyManagedInitMethod("afterPropertiesSet"))) {
        ((InitializingBean) bean).afterPropertiesSet();
    }

    // ⚠：2.调用xml配置中通过init-method属性指定的初始化方法
    if (mbd != null && bean.getClass() != NullBean.class) {
        String initMethodName = mbd.getInitMethodName();
        if (StringUtils.hasLength(initMethodName) &&
```

其他步骤在上文中均已进行过介绍，此处不再赘述。

Spring中Bean的生命周期详解 fudaihb的
Spring框架作为Java开发中最重要的框架之一，受到了广泛的关注和使用。Spring的核心概念之一就是Bean，它是Spring IoC（Inversion of Control，控制反转）容器管理

Spring 的 Bean 管理 第2关：bean 的生命周期 于建章的
bean 的生命周期

详解Bean的生命周期
BeanDefinition里面里面包含了bean定义的各种信息,如:bean对应的class、scope、lazy信息、dependOn信息、autowireCandidate(是否是候选对象)、primary(是否是主要

Bean 的生命周期总结_bean的生命周期
Java 中的公共类称之为 Bean 或 Java Bean,而 Spring 中的 Bean 指的是将对象的生命周期,交个 Spring IoC 容器来管理的对象。所以 Spring 中的 Bean 对象在使用时,无

第2关：bean 的生命周期 @ZhengLan的
题解代码： 1.applicationContext.xml <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w

Spring注解开发（二）——Bean的生命周期 幽灵 逐梦--
Spring注解驱动开发（二） Bean的生命周期 bean创建—初始化—销毁的过程 容器管理bean的生命周期 可以自定义初始化和销毁方法；容器在bean进行到当前生命周期的

【Spring6】| Bean的生命周期(五步、七步、十步法剖析)
(1)Spring其实就是一个管理Bean对象的工厂,它负责对象的创建,对象的销毁等。 (2)所谓的生命周期就是:对象从创建开始到最终销毁的整个过程。 (3)为什么要知道Bean的

一文读懂 Bean的生命周期
bean的生命周期指的是:bean创建-->初始化-->销毁 的过程,bean的生命周期由容器进行管理,我们可以自定义bean的初始化和销毁方法来满足我们的需求,当容器在bean进

Spring-02-Bean的生命周期 程序yuar
bean的生命周期：指 bean创建-----初始化----销毁 的过程 bean的生命周期是由容器进行管理的。 我们可以自定义 bean初始化和销毁 方法: 容器在bean进行到当前生命周

谈谈我对Spring Bean 生命周期的理解
Spring Bean 生命周期是 Spring 框架中的一个核心概念，了解 Spring Bean 的生命周期对我们了解整个 Spring 框架会有很大的帮助。本文将详细介绍 Spring Bean 生命

带你彻底掌握 Bean 的生命周期_bean的生命周期
调用销毁方法:如果 Bean 配置了销毁方法,Spring 会在所有 Bean 都已经使用完毕,且 IOC 容器关闭之前调用它,可以在销毁方法里面做一些资源释放的工作,比如关闭连接、

Spring之Bean的生命周期_spring bean的生命周期
2. Bean生命周期的流程步骤(四个阶段讲解) 2.1 实例化 通过XML、Java annotation(注解)以及Java Configuration(配置类)等方式加载Spring Bean。程序启动后,Spring把注

spring bean的生命周期
2. **容器管理的生命周期回调** - **Singleton Beans的懒加载**：如果Bean的scope为singleton，并且在XML配置中没有设置`lazy-init="true"`，那么Spring容器在启动时就

Spring学习笔记之bean生命周期
《Spring学习笔记之bean生命周期》 在Spring框架中，Bean是核心组件，它们构成了应用程序的主要结构。理解Spring Bean的生命周期对于有效地管理和优化Spring应

Spring系列(三)之Bean的生命周期以及Bean的单例与多例模式
一.Bean的生命周期 bean的生命周期可以表达为:bean的定义➡bean的初始化➡bean的使用➡bean的销毁 Bean的初始化过程 1)通过XML、Java annotation(注解)以及Jav

Spring Bean生命周期和重要接口之概述_bean三级缓存
1 Bean生命周期 1.1 概述 Spring Bean的生命周期对Spring框架原理理解的重要性,所以接下来我们就来分析一下Bean生命周期的整体流程。首先Bean就是一些Java对象,

浅谈Spring bean 生命周期验证
浅谈Spring bean 生命周期验证 Spring bean 生命周期验证是 Spring 框架中一个非常重要的概念，它描述了 bean 从创建到销毁的整个生命周期。了解 Spring bean 生命

spring中bean的生命周期详解
"Spring中Bean的生命周期详解" Spring框架是当前Java EE开发中最流行的框架之一，Spring框架中Bean的生命周期是指从创建到销毁的整个过程。在这个过程中，Sprin

Spring——bean的生命周期(Springboot --- 2.6.1)_springboot bean的...
Spring的生命周期 0.1 扫描 bean 去扫描XML/注解/JavaConfig 0.2 BeanDefinition 初始化时的bean BeanDefinition表示了一个实例,它具有属性值、构造函数参数值以及具

bean的生命周期详解_bean生命周期
在学习 Bean 的生命周期之前,你至少应该知道,或者至少了解 Spring IOC 和 DI 以及他们的详细流程、Spring 容器的初始化流程、AOP 的代码织入过程等,有兴趣的还

【俯瞰Spring】二、Bean的生命周期 温柔一
文章目录一、前言二、Bean的生命周期三、Bean生命周期剖析3.1 Bean定义生成、注册3.2 实例化3.2.1实例化是什么？3.2.2 扩展点3.3 依赖注入3.4 初始化3.5 销毁四、

Spring学习（二）—— bean的生命周期 weixin_45716265的
目录（一）构造（对象创建）（1）单实例（2）多实例（二）初始化\销毁（1）指定初始化和销毁方法（2）通过让bean实现InitializingBean（定义初始化逻辑），Dispos

Spring中单例模式下Bean的生命周期 Alsace_的
在Spring中，我们将对象交给框架来管理，由IOC容器来负责对象的创建与管理。Spring中Bean的生命周期是指

Spring源码分析之单例bean的生命周期

1.前言 bean的生命周期，无非是bean的创建--->初始化---->销毁这三步，为了给这个概念具象化，下面给出一个例子说明。比如：现在有个类A，那么上述三个步骤分别是

spring学习（六）——2 Bean的生命周期 大风的
参考文章： http://www.iocoder.cn/ 关于Bean的生命周期，我们先看一张图，这张图标识Bean创建和销毁的流程 从上图，我们可以看到整个流程是： bean实例化：创建b

Spring Bean生命周期执行流程
创建前准备、创建实例、依赖注入、容器缓存、销毁实例。

【Java面试小短文】Spring Bean生命周期的执行流程 卓越无关环境，保持空杯心态——靡不有初，鲜克有
生命周期全过程大致分为五个阶段：创建前准备阶段、创建实例阶段、 依赖注入阶段、 容器缓存阶段和销毁实例阶段。 快来看看详细解释！

spring的Bean生命周期流程图 zkr1234562的
spring的bean生命周期流程图

spring生命周期和bean生命周期 **最新发布**
Spring 生命周期和 Bean 生命周期是密切相关的。在 Spring 容器中，每个 Bean 都有一个完整的生命周期，即从实例化、依赖注入，到销毁的过程，Spring 容器为我们管

Ewen Seong
码龄6年 暂无认证

83	3464	1万+	15万+	
原创	周排名	总排名	访问	等级
1685	2533	800	31	814
积分	粉丝	获赞	评论	收藏

私信 已关注

搜博主文章

热门文章

Spring系列-9 Async注解使用与原理 4592
Spring系列-6 占位符使用和原理 4402
Spring系列-1 启动流程 4324
SpringMVC系列-1 使用方式和启动流程 3950
事务-2 Spring与Mybatis事务实现原理 3814

分类专栏

最新评论

前端系列-7 Vue3响应式数据
全栈小5: 文章写的很详细，条理清晰，很容易看进去，学到了很多知识，感谢博主分 ...

前端系列-7 Vue3响应式数据
ha_lydms: 非常不错的技术领域文章分享，解决了我在实践中的大问题！博主很有才 ...

多线程系列-2 线程中断机制
Ewen Seong: 可以结合"多线程系列-1 线程的状态"理解线程中断的概念

Nginx系列-7 upstream与负载均衡
阿登_: 描述得很详细 很到位👍

Lua使用方式介绍
Ewen Seong: lua官网地址: https://www.lua.org/

最新文章

LocalDateTime的序列化和反序列化

前端系列-9 Vue3生命周期和computed和watch

Nginx系列-12 Nginx使用Lua脚本进行JWT校验

2024年 35篇	2023年 18篇
2022年 17篇	2021年 13篇

目录