

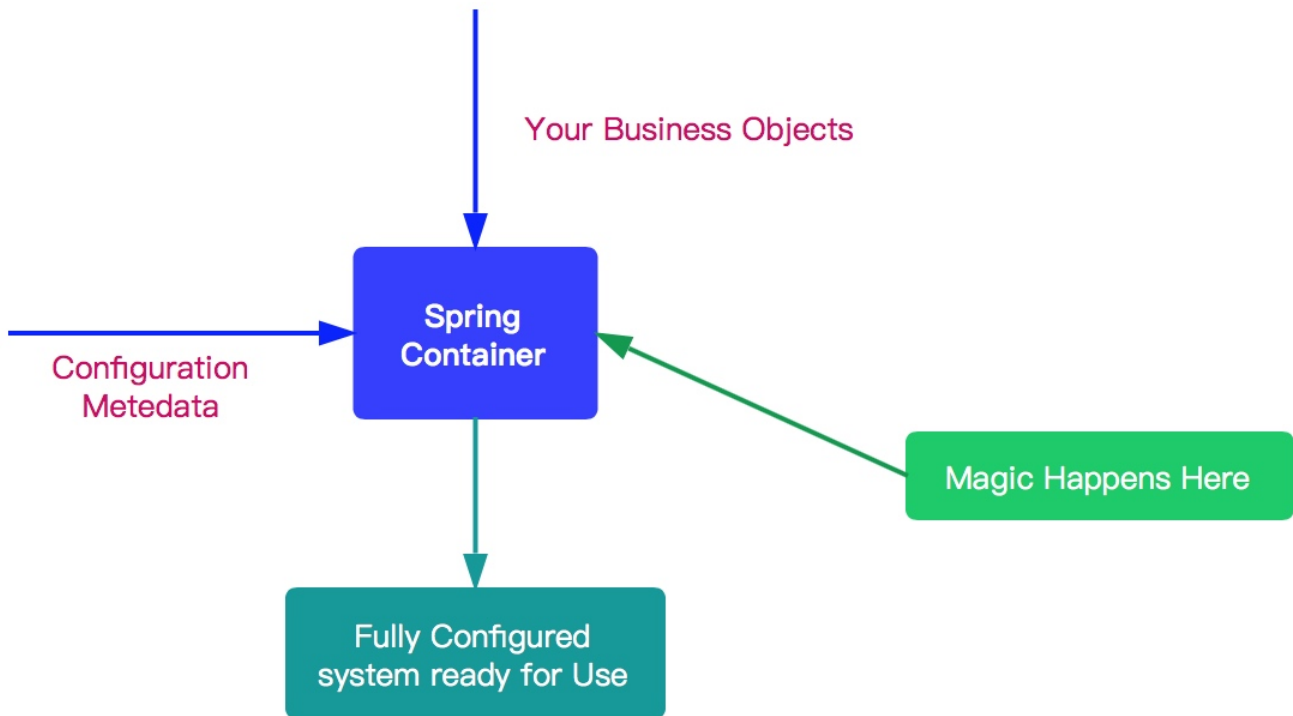


当前位置: Java 技术驿站 (<http://cmsblogs.com>) > 死磕Java (<http://cmsblogs.com/?cat=189>) > 死磕 Spring (<http://cmsblogs.com/?cat=206>) > 正文

【死磕 Spring】—— IOC 之开启 bean 的加载 (<http://cmsblogs.com/?p=2806>)

2018-10-18 分类: 死磕 Spring (<http://cmsblogs.com/?cat=206>) 阅读(12773) 评论(7)


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




(<https://gitee.com/chenssy/blog-home/raw/master/image/201810/15397758271868.jpg>) (此图来自《Spring 揭秘》) Spring IOC 容器所起的作用如上图所示, 它会以某种方式加载 Configuration Metadata, 将其解析注册到容器内部, 然后回根据这些信息绑定整个系统的对象, 最终组装成一个可用的基于轻量级容器的应用系统。Spring 在实现上述功能中, 将整个流程分为两个阶段: 容器初始化阶段和加载bean 阶段。

- **容器初始化阶段**: 首先通过某种方式加载 Configuration Metadata (主要是依据 Resource、ResourceLoader 两个体系), 然后容器会对加载的 Configuration MetaData 进行解析和分析, 并将分析的信息组装成 BeanDefinition, 并将其保存注册到相应的 BeanDefinitionRegistry 中。至此, Spring IOC 的初始化工作完成。
- **加载 bean 阶段**: 经过容器初始化阶段后, 应用程序中定义的 bean 信息已经全部加载到系统中了, 当我们显示或者隐式地调用 `getBean()` 时, 则会触发加载 bean 阶段。在这阶段, 容器会首先检查所请求的对象是否已经初始化完成了, 如果没有, 则会根据注册的 bean 信息实例化请求的对象, 并为其注册依赖, 然后将其返回给请求方。至此第二个阶段也已经完成。

第一个阶段前面已经用了 10 多篇博客深入分析了 (总结参考【死磕 Spring】----- IOC 之 IOC 初始化总结())。所以从这篇开始分析第二个阶段: 加载 bean 阶段。当我们显示或者隐式地调用 `getBean()` 时, 则会触发加载 bean 阶段。如下:



```
public Object getBean(String name) throws BeansException {  
    return doGetBean(name, null, null, false);  
}
```



内部调用 doGetBean() 方法，其接受四个参数：

- name：要获取 bean 的名字
- requiredType：要获取 bean 的类型
- args：创建 bean 时传递的参数。这个参数仅限于创建 bean 时使用
- typeCheckOnly：是否为类型检查

这个方法的代码比较长，各位耐心看下：



protected <T> T doGetBean(final String name, @Nullable final Class<T> requiredType,
@Nullable final Object[] args, boolean typeCheckOnly) throws BeansException



```
{
    // 获取 beanName, 这里是一个转换动作, 将 name 转换为 beanName, 比如传进去的 name 是 alias 别名, 则经过
    // transformedBeanName 方法转换为真实的 beanIdName
    final String beanName = transformedBeanName(name);
    Object bean;
    // 从缓存中或者实例工厂中获取 bean
    // *** 这里会涉及到解决循环依赖 bean 的问题
    Object sharedInstance = getSingleton(beanName);
    if (sharedInstance != null && args == null) {

        if (logger.isDebugEnabled()) {
            if (isSingletonCurrentlyInCreation(beanName)) {
                logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +
                    "' that is not fully initialized yet - a consequence of a circular reference"
                );
            }
            else {
                logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
            }
        }
        bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
    }

    else {

        // 因为 Spring 只解决单例模式下得循环依赖, 在原型模式下如果存在循环依赖则会抛出异常
        // **关于循环依赖后续会单独出文详细说明**
        if (isPrototypeCurrentlyInCreation(beanName)) {
            throw new BeanCurrentlyInCreationException(beanName);
        }

        // 如果容器中没有找到, 则从父类容器中加载
        BeanFactory parentBeanFactory = getParentBeanFactory();
        if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
            String nameToLookup = originalBeanName(name);
            if (parentBeanFactory instanceof AbstractBeanFactory) {
                return ((AbstractBeanFactory) parentBeanFactory).doGetBean(
                    nameToLookup, requiredType, args, typeCheckOnly);
            }
            else if (args != null) {
                return (T) parentBeanFactory.getBean(nameToLookup, args);
            }
            else {
                return parentBeanFactory.getBean(nameToLookup, requiredType);
            }
        }

        // 如果不是仅仅做类型检查则是创建 bean, 这里需要记录
        if (!typeCheckOnly) {
```



```

markBeanAsCreated(beanName);
    }

    try {
        // 从容器中获取 beanName 相应的 GenericBeanDefinition, 并将其转换为 RootBeanDefinition
        final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);

        // 检查给定的合并的 BeanDefinition
        checkMergedBeanDefinition(mbd, beanName, args);

        // 处理所依赖的 bean
        String[] dependsOn = mbd.getDependsOn();
        if (dependsOn != null) {
            for (String dep : dependsOn) {
                // 若给定的依赖 bean 已经注册为依赖给定的bean
                // 循环依赖的情况
                if (isDependent(beanName, dep)) {
                    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                        "Circular depends-on relationship between '" + beanName + "' and '" +
dep + "'");
                }
                // 缓存依赖调用
                registerDependentBean(dep, beanName);
                try {
                    getBean(dep);
                }
                catch (NoSuchBeanDefinitionException ex) {
                    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                        "'" + beanName + "' depends on missing bean '" + dep + "'", ex);
                }
            }
        }

        // bean 实例化
        // 单例模式
        if (mbd.isSingleton()) {
            sharedInstance = getSingleton(beanName, () -> {
                try {
                    return createBean(beanName, mbd, args);
                }
                catch (BeansException ex) {
                    // 显示从单例缓存中删除 bean 实例
                    // 因为单例模式下为了解决循环依赖, 可能他已经存在了, 所以销毁它
                    destroySingleton(beanName);
                    throw ex;
                }
            });
            bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
        }

        // 原型模式
        else if (mbd.isPrototype()) {

```

这里是根据beanName和BeanDefinition创建具体的对象比如 Person实例, sharedInstance就是这个对象, 初始化结束后单例对象person被放到, Beanfactory内部的单例hashmap中, key是beanName, value就是单例对象。



```
// It's a prototype -> create a new instance.
Object prototypeInstance = null;
try {
    beforePrototypeCreation(beanName);
    prototypeInstance = createBean(beanName, mbd, args);
}
finally {
    afterPrototypeCreation(beanName);
}
bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
}

else {
    // 从指定的 scope 下创建 bean
    String scopeName = mbd.getScope();
    final Scope scope = this.scopes.get(scopeName);
    if (scope == null) {
        throw new IllegalStateException("No Scope registered for scope name '" + scopeName
e + "'");
    }
    try {
        Object scopedInstance = scope.get(beanName, () -> {
            beforePrototypeCreation(beanName);
            try {
                return createBean(beanName, mbd, args);
            }
            finally {
                afterPrototypeCreation(beanName);
            }
        });
        bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);
    }
    catch (IllegalStateException ex) {
        throw new BeanCreationException(beanName,
            "Scope '" + scopeName + "' is not active for the current thread; consider
" +
            "defining a scoped proxy for this bean if you intend to refer to
it from a singleton",
            ex);
    }
}
catch (BeansException ex) {
    cleanupAfterBeanCreationFailure(beanName);
    throw ex;
}

// 检查需要的类型是否符合 bean 的实际类型
if (requiredType != null && !requiredType.isInstance(bean)) {
    try {
        T convertedBean = getTypeConverter().convertIfNecessary(bean, requiredType);

```



```

        if (convertedBean == null) {
            throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
        }
        return convertedBean;
    }
    catch (TypeMismatchException ex) {
        if (logger.isDebugEnabled()) {
            logger.debug("Failed to convert bean '" + name + "' to required type '" +
                ClassUtils.getQualifiedName(requiredType) + "'", ex);
        }
        throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
    }
}
return (T) bean;
}

```

▼ bean = {Person@2057} "Person{age=1, name='AAA'}"

f age = 1
 > f name = "AAA"

代码是相当长，处理逻辑也是相当复杂，下面将其进行拆分阐述。 **1. 获取 beanName**

```
final String beanName = transformedBeanName(name);
```

这里传递的是 name，不一定就是 beanName，可能是 aliasName，也有可能是 FactoryBean，所以这里需要调用 transformedBeanName() 方法对 name 进行一番转换，主要如下：



```
protected String transformedBeanName(String name) {
    return canonicalName(BeanFactoryUtils.transformedBeanName(name));
}
```

// 去除 FactoryBean 的修饰符

```
public static String transformedBeanName(String name) {
    Assert.notNull(name, "'name' must not be null");
    String beanName = name;
    while (beanName.startsWith(BeanFactory.FACTORY_BEAN_PREFIX)) {
        beanName = beanName.substring(BeanFactory.FACTORY_BEAN_PREFIX.length());
    }
    return beanName;
}
```

// 转换 aliasName

```
public String canonicalName(String name) {
    String canonicalName = name;
    // Handle aliasing...
    String resolvedName;
    do {
        resolvedName = this.aliasMap.get(canonicalName);
        if (resolvedName != null) {
            canonicalName = resolvedName;
        }
    }
    while (resolvedName != null);
    return canonicalName;
}
```

主要处理过程包括两步：

1. 去除 FactoryBean 的修饰符。如果 name 以 "&" 为前缀，那么会去掉该 "&"，例如，name = "&studentService"，则会变成 name = "studentService"。
2. 取指定的 alias 所表示的最终 beanName。主要是一个循环获取 beanName 的过程，例如别名 A 指向名称为 B 的 bean 则返回 B，若别名 A 指向别名 B，别名 B 指向名称为 C 的 bean，则返回 C。

2.从单例 bean 缓存中获取 bean 对应代码段如下：



```
Object sharedInstance = getSingleton(beanName);
if (sharedInstance != null && args == null) {
    if (logger.isDebugEnabled()) {
        if (isSingletonCurrentlyInCreation(beanName))
            logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +
                "' that is not fully initialized yet - a consequence of a circular reference"
            );
    }
    else {
        logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
    }
}
bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
```

this.singletonObjects.get(beanName)
this指的是DefaultListableBeanFactory，内部有hashmap=earlySingletonObjects，key是beanName，value就是创建的单例实例对象。

```

this.singletonObjects = {ConcurrentHashMap@1559} size = 1
> == "person2" -> {Person@1914} "Person{age=1, name='AAA'}"

```

单例对象在第一次getBean时候，创建单例对象比如person同时会加入到BeanFactor内部的单例hashmap中，key=beanName，value=单例对象

我们知道单例模式的 bean 在整个过程中只会被创建一次，第一次创建后会将该 bean 加载到缓存中，后面在获取 bean 就会直接从单例缓存中获取。如果从缓存中得到了 bean，则需要调用 getObjectForBeanInstance() 对 bean 进行实例化处理，因为缓存中记录的是最原始的 bean 状态，我们得到的不一定是我们最终想要的 bean。3.原型模式依赖检查与 parentBeanFactory 对应代码段

```

if (isPrototypeCurrentlyInCreation(beanName)) {
    throw new BeanCurrentlyInCreationException(beanName);
}

// Check if bean definition exists in this factory.
BeanFactory parentBeanFactory = getParentBeanFactory();
if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
    // Not found -> check parent.
    String nameToLookup = originalBeanName(name);
    if (parentBeanFactory instanceof AbstractBeanFactory) {
        return ((AbstractBeanFactory) parentBeanFactory).doGetBean(
            nameToLookup, requiredType, args, typeCheckOnly);
    }
    else if (args != null) {
        // Delegation to parent with explicit args.
        return (T) parentBeanFactory.getBean(nameToLookup, args);
    }
    else {
        // No args -> delegate to standard getBean method.
        return parentBeanFactory.getBean(nameToLookup, requiredType);
    }
}

```

Spring 只处理单例模式下得循环依赖，对于原型模式的循环依赖直接抛出异常。主要原因还是在于 Spring 解决循环依赖的策略有关。对于单例模式 Spring 在创建 bean 的时候并不是等 bean 完全创建完成后才会将 bean 添加至缓存中，而是不等 bean 创建完成就会将创建 bean 的 ObjectFactory 提早加入到缓存中，这样一旦下一个 bean 创建的时候需要依赖 bean 时则直接使用 ObjectFactory。但是原型模式我们知道是没法使用缓存的，所以 Spring 对原型模式的循环依赖处理策略则是不处理（关于循环依赖后面会有单独文章说

明)。如果容器缓存中没有相对应的 BeanDefinition 则会尝试从父类工厂 (parentBeanFactory) 中加载，然后再去递归调用 getBean()。3. 依赖处理 对应源码如下：

```
String[] dependsOn = mbd.getDependsOn();
if (dependsOn != null) {
    for (String dep : dependsOn) {
        if (isDependent(beanName, dep)) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                "Circular depends-on relationship between '" + beanName + "' and '" +
dep + "'");
        }
        registerDependentBean(dep, beanName);
        try {
            getBean(dep);
        }
        catch (NoSuchBeanDefinitionException ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                "'" + beanName + "' depends on missing bean '" + dep + "'", ex);
        }
    }
}
```

每个 bean 都不是单独工作的，它会依赖其他 bean，其他 bean 也会依赖它，对于依赖的 bean，它会优先加载，所以在 Spring 的加载顺序中，在初始化某一个 bean 的时候首先会初始化这个 bean 的依赖。**作用域处理** Spring bean 的作用域默认为 singleton，当然还有其他作用域，如prototype、request、session 等，不同的作用域会有不同的初始化策略。**类型转换** 在调用 doGetBean() 方法时，有一个 requiredType 参数，该参数的功能就是将返回的 bean 转换为 requiredType 类型。当然就一般而言我们是不需要进行类型转换的，也就是 requiredType 为空（比如 getBean(String name)），但有可能会存在这种情况，比如我们返回的 bean 类型为 String，我们在使用的时候需要将其转换为 Integer，那么这个时候 requiredType 就有用武之地了。当然我们一般是不需要这样做的。至此 getBean() 过程讲解完了。后续将会对该过程进行拆分，更加详细的说明，弄清楚其中的来龙去脉，所以这篇博客只能算是 Spring bean 加载过程的一个概览。拆分主要是分为三个部分：

1. 分析从缓存中获取单例 bean，以及对 bean 的实例中获取对象
2. 如果从单例缓存中获取 bean，Spring 是怎么加载的呢？所以第二部分是分析 bean 加载，以及 bean 的依赖处理
3. bean 已经加载了，依赖也处理完毕了，第三部分则分析各个作用域的 bean 初始化过程。

👍 赞(27)

¥ 打赏

【公告】版权声明 (http://cmsblogs.com/?page_id=1908)

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>)