
回到首页

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
2019-05-10
Spring

【死磕 Spring】—— IoC 之加载 Bean：分析各 scope 的 Bean 创建

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 Spring 中存在着不同的 scope，默认是 singleton ，还有 prototype、request 等等其他的 scope 。他们的初始化步骤是怎样的呢？这个答案在这篇博客中给出。

1. singleton

Spring 的 scope 默认为 singleton ，其初始化的代码如下：

```
// AbstractBeanFactory.java

if (mbd.isSingleton()) { // 单例模式
    sharedInstance = getSingleton(beanName, () -> {
        try {
            return createBean(beanName, mbd, args);
        }
        catch (BeansException ex) {
            // Explicitly remove instance from singleton cache: It might have been put there
            // eagerly by the creation process, to allow for circular reference resolution.
            // Also remove any beans that received a temporary reference to the bean.
            // 显式从单例缓存中删除 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。 TODO 芋艿
            destroySingleton(beanName);
            throw ex;
        }
    });
    // <x>
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
}
```

在 《【死磕 Spring】—— IoC 之加载 Bean：从单例缓存中获取单》 中，已经分析了从缓存中获取单例模式的 bean 。但是如果缓存中不存在呢？则需要从头开始加载 Bean ，这个过程由 #getSingleton(String beanName, ObjectFactory<?> singletonFactory) 方法来实现。代码如下：

```
// DefaultSingletonBeanRegistry.java

public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(beanName, "Bean name must not be null");
    // 全局加锁
    synchronized (this.singletonObjects) {
        // <1> 从缓存中检查一遍
        // 因为 singleton 模式其实就是复用已经创建的 bean 所以这步骤必须检查
        Object singletonObject = this.singletonObjects.get(beanName);
        //  为空，开始加载过程
        if (singletonObject == null) {
            if (this.singletonsCurrentlyInDestruction) {
                throw new BeanCreationNotAllowedException(beanName,
                        "Singleton bean creation not allowed while singletons of this factory are in destructio
                        "(Do not request a bean from a BeanFactory in a destroy method implementation!)");
            }
            if (logger.isDebugEnabled()) {
                logger.debug("Creating shared instance of singleton bean '" + beanName + "'");
            }
            // <2> 加载前置处理
            beforeSingletonCreation(beanName);
            boolean newSingleton = false;
            boolean recordSuppressedExceptions = (this.suppressedExceptions == null);
            if (recordSuppressedExceptions) {
                this.suppressedExceptions = new LinkedHashSet<>();
            }
            try {
                // <3> 初始化 bean
                // 这个过程其实是调用 createBean() 方法
                singletonObject = singletonFactory.getObject();
                newSingleton = true;
            } catch (IllegalStateException ex) {
                // Has the singleton object implicitly appeared in the meantime ->
                // if yes, proceed with it since the exception indicates that state.
                singletonObject = this.singletonObjects.get(beanName);
                if (singletonObject == null) {
                    throw ex;
                }
            } catch (BeanCreationException ex) {
                if (recordSuppressedExceptions) {
                    for (Exception suppressedException : this.suppressedExceptions) {
                        ex.addRelatedCause(suppressedException);
                    }
                }
                throw ex;
            } finally {
                if (recordSuppressedExceptions) {
                    this.suppressedExceptions = null;
                }
                // <4> 后置处理
                afterSingletonCreation(beanName);
            }
            // <5> 加入缓存中
            if (newSingleton) {
                addSingleton(beanName, singletonObject);
            }
```

```
            }
            return singletonObject;
        }
    }
```

- 其实，这个过程并没有真正创建 Bean 对象，仅仅只是做了一部分准备和预处理步骤。真正获取单例 bean 的方法，其实是由 <3> 处的 singletonFactory.getObject() 这部分代码块来实现，而 singletonFactory 由回调方法产生。
- 那么这个方法做了哪些准备呢？
  - <1> 处，再次检查缓存是否已经加载过，如果已经加载了则直接返回，否则开始加载过程。
  - <2> 处，调用 #beforeSingletonCreation(String beanName) 方法，记录加载单例 bean 之前的加载状态，即前置处理。在 《【死磕 Spring】—— IoC 之加载 Bean：从单例缓存中获取单例 Bean》 中，已经详细解析。
  - <3> 处，调用参数传递的 ObjectFactory 的 #getObject() 方法，实例化 bean 。【重要】后续文章，详细解析。
  - <4> 处，调用 #afterSingletonCreation(String beanName) 方法，进行加载单例后的后置处理。在 《【死磕 Spring】—— IoC 之加载 Bean：从单例缓存中获取单例 Bean》 中，已经详细解析。
  - <5> 处，调用 #addSingleton(String beanName, Object singletonObject) 方法，将结果记录并加入值缓存中，同时删除加载 bean 过程中所记录的一些辅助状态。详细解析，见「1.1 addSingleton」 。

在 <x> 处，加载了单例 bean 后，调用 #getObjectForBeanInstance(Object beanInstance, String name, String beanName, RootBeanDefinition mbd) 方法，从 bean 实例中获取对象。该方法已经在 《【死磕 Spring】—— IoC 之加载 Bean：从单例缓存中获取单例 Bean》 中，详细分析了。

1.1 addSingleton

```
// DefaultSingletonBeanRegistry.java

/**
 * Cache of singleton objects: bean name to bean instance.
 *
 * 存放的是单例 bean 的映射。
 *
 * 对应关系为 bean name --> bean instance
 */
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

/**
 * Cache of singleton factories: bean name to ObjectFactory.
 *
 * 存放的是【早期】的单例 bean 的映射。
 *
 * 对应关系也是 bean name --> bean instance。
 *
 * 它与 {@link #singletonObjects} 的区别区别在，于 earlySingletonObjects 中存放的 bean 不一定是完整的。
 *
 * 从 {@link #getSingleton(String)} 方法中，中我们可以了解，bean 在创建过程中就已经加入到 earlySingletonObjects 中了，
 * 所以当在 bean 的创建过程中就可以通过 getBean() 方法获取。
 * 这个 Map 也是解决【循环依赖】的关键所在。
 **/
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);
```

```
/**
 * Cache of early singleton objects: bean name to bean instance.
 *
 * 存放的是 ObjectFactory 的映射，可以理解为创建单例 bean 的 factory 。
 *
 * 对应关系是 bean name --> ObjectFactory
 */
private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);

/** Set of registered singletons, containing the bean names in registration order. */
private final Set<String> registeredSingletons = new LinkedHashSet<>(256);

protected void addSingleton(String beanName, Object singletonObject) {
 synchronized (this.singletonObjects) {
     this.singletonObjects.put(beanName, singletonObject);
     this.singletonFactories.remove(beanName);
     this.earlySingletonObjects.remove(beanName);
     this.registeredSingletons.add(beanName);
    }
}
```

一个 put、一个 add、两个 remove 操作。
【put】singletonObjects 属性，单例 bean 的缓存。
【remove】singletonFactories 属性，单例 bean Factory 的缓存。
【remove】earlySingletonObjects 属性，“早期”创建的单例 bean 的缓存。
【add】registeredSingletons 属性，已经注册的单例缓存。

2. 原型模式

```
// AbstractBeanFactory.java

else if (mbd.isPrototype()) {
    Object prototypeInstance = null;
    try {
       // <1> 加载前置处理
        beforePrototypeCreation(beanName);
        // <2> 创建 Bean 对象
        prototypeInstance = createBean(beanName, mbd, args);
    } finally {
       // <3> 加载后缀处理
        afterPrototypeCreation(beanName);
    }
    // <4> 从 Bean 实例中获取对象
    bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
}
```

原型模式的初始化过程很简单：直接创建一个新的 Bean 的实例就可以了。过程如下：

在 <1> 处，调用 #beforePrototypeCreation(String beanName) 方法，记录加载原型模式 bean 之前的加载状态，即前置处理。详细解析，见 「2.1 beforePrototypeCreation」 。
在 <2> 处，调用 #createBean(String beanName) 方法，创建一个 bean 实例对象。【重要】后续文章，详细解析。
在 <3> 处，调用 #afterSingletonCreation(String beanName) 方法，进行加载原型模式 bean 后的后置处理。详细解析，见 「2.3 afterSingletonCreation」 。

在 <4> 处，加载了单例 bean 后，调用 #getObjectForBeanInstance(Object beanInstance, String name, String beanName, RootBeanDefinition mbd) 方法，从 bean 实例中获取对象。该方法已经在 《【死磕 Spring】—— IoC 之加载 Bean：从单例缓存中获取单例 Bean》 中，详细分析了。

2.1 beforePrototypeCreation

```
// AbstractBeanFactory.java

/** Names of beans that are currently in creation. */
private final ThreadLocal<Object> prototypesCurrentlyInCreation =
     new NamedThreadLocal<>("Prototype beans currently in creation");

protected void beforePrototypeCreation(String beanName) {
    Object curVal = this.prototypesCurrentlyInCreation.get();
 if (curVal == null) { // String
     this.prototypesCurrentlyInCreation.set(beanName);
    } else if (curVal instanceof String) { // String => Set
        Set<String> beanNameSet = new HashSet<>(2);
        beanNameSet.add((String) curVal);
        beanNameSet.add(beanName);
     this.prototypesCurrentlyInCreation.set(beanNameSet);
    } else { // Set
        Set<String> beanNameSet = (Set<String>) curVal;
        beanNameSet.add(beanName);
    }
}
```

2.2 afterSingletonCreation

```
// AbstractBeanFactory.java

protected void afterPrototypeCreation(String beanName) {
    Object curVal = this.prototypesCurrentlyInCreation.get();
 if (curVal instanceof String) { // String => null
     this.prototypesCurrentlyInCreation.remove();
    } else if (curVal instanceof Set) { // Set
        Set<String> beanNameSet = (Set<String>) curVal;
        beanNameSet.remove(beanName);
     if (beanNameSet.isEmpty()) { // Set => null
         this.prototypesCurrentlyInCreation.remove();
        }
    }
}
```

3. 其它作用域

```
// AbstractBeanFactory.java

else {
    // 获得 scopeName 对应的 Scope 对象
    String scopeName = mbd.getScope();
```

```
    final Scope scope = this.scopes.get(scopeName);
    if (scope == null) {
        throw new IllegalStateException("No Scope registered for scope name '" + scopeName + "'");
    }
    try {
        // 从指定的 scope 下创建 bean
        Object scopedInstance = scope.get(beanName, () -> {
            // 加载前置处理
            beforePrototypeCreation(beanName);
            try {
                // 创建 Bean 对象
                return createBean(beanName, mbd, args);
            } finally {
                // 加载后缀处理
                afterPrototypeCreation(beanName);
            }
        });
        // 从 Bean 实例中获取对象
        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);
    }
}
```

核心流程和原型模式一样，只不过获取 bean 实例是由 Scope#get(String name, ObjectFactory<?> objectFactory) 方法来实现。代码如下：

```
// SimpleThreadScope.java

private final ThreadLocal<Map<String, Object>> threadScope =
    new NamedThreadLocal<Map<String, Object>>("SimpleThreadScope") {
        @Override
        protected Map<String, Object> initialValue() {
            return new HashMap<>();
        }
    };

@Override
public Object get(String name, ObjectFactory<?> objectFactory) {
    // 获取 scope 缓存
    Map<String, Object> scope = this.threadScope.get();
    Object scopedObject = scope.get(name);
    if (scopedObject == null) {
        scopedObject = objectFactory.getObject();
        // 加入缓存
        scope.put(name, scopedObject);
    }
    return scopedObject;
}
```

- org.springframework.beans.factory.config.Scope 接口，有多种实现类。其他的 Scope 实现类，感兴趣的胖友，可以单独去看。

4. 小结

对于上面三个模块，其中最重要的有两个方法：

> 一个是 `#createBean(String beanName, RootBeanDefinition mbd, Object[] args)` 方法。
> 一个是 `#getObjectForBeanInstance(Object beanInstance, String name, String beanName, RootBeanDefinition mbd)` 方法。

这两个方法在上面三个模块都有调用。

> `#createBean(String beanName, RootBeanDefinition mbd, Object[] args)` 方法，后续详细说明。

> `#getObjectForBeanInstance(Object beanInstance, String name, String beanName, RootBeanDefinition mbd)` 方法，在博客 《【死磕 Spring】—— IoC 之加载 Bean：从单例缓存中获取单》 中有详细讲解。这里再次阐述下（此段内容来自《Spring 源码深度解析》）：
>
> > 这个方法主要是验证以下我们得到的 bean 的正确性，其实就是检测当前 bean 是否是 FactoryBean 类型的 bean 。
> >
> > 如果是，那么需要调用该 bean 对应的 FactoryBean 实例的 `#getObject()` 方法，作为返回值。
> >
> > 无论是从缓存中获得到的 bean 还是通过不同的 scope 策略加载的 bean 都只是最原始的 bean 状态，并不一定就是我们最终想要的 bean。
> >
> > 举个例子，加入我们需要对工厂 bean 进行处理，那么这里得到的其实是工厂 bean 的初始状态，但是我们真正需要的是工厂 bean 中定义 `factory-method` 方法中返回的 bean，而 `#getObjectForBeanInstance(Object beanInstance, String name, String beanName, RootBeanDefinition mbd)` 方法，就是完成这个工作的。

至此，Spring 加载 bean 的三个部分（LZ自己划分的）已经分析完毕了。

文章目录