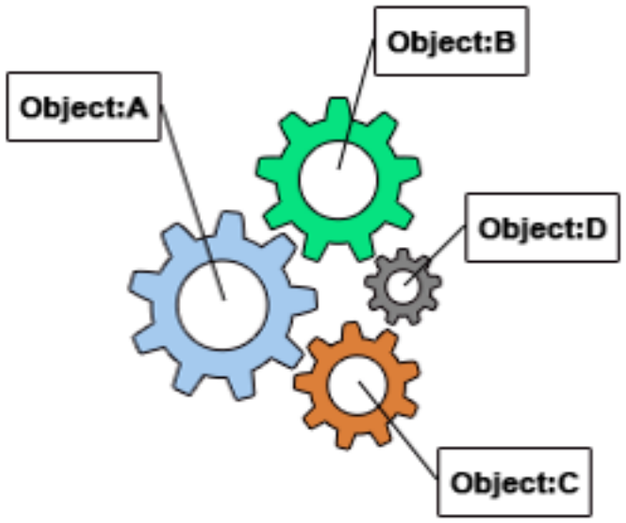
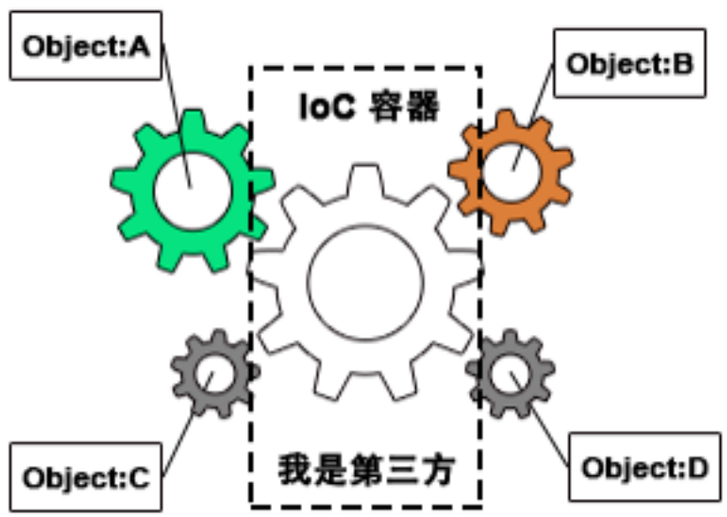
Spring 原理

IOC 容器解耦


Object:A
Object:B
Object:D
Object:C


Object:A
IoC 容器
Object:B
Object:C
我是第三方
Object:D

GetBean过程

```
TeacherService teacherService = context.getBean(TeacherService.class);

StudentService studentService = (StudentService) context.getBean("studentService");

ProtoService protoA = context.getBean(ProtoService.class);

ProtoService protoB = context.getBean(ProtoService.class);
```

spring bean加载相关的缓存有以下这些：

```
/** Cache of singleton objects: bean name --> bean instance */
```
已经完全实例化的*Bean*，*beanName*和*bean*实例之间的关系
```
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

/** Cache of singleton factories: bean name --> ObjectFactory */
```
记录*beanName*和创建*bean*工厂之间的关系
```
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);

/** Cache of early singleton objects: bean name --> bean instance */
```
为了解决循环依赖的问题，提前暴露*SingletonBean*，*beanName*和原始*bean*实例之间的关系，即使*bean*还在创建过程中，也可以通过*getBean*获取到
```
private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);
```

循环依赖问题

spring单例在同一个spring容器中只创建一次，之后在获取bean的时候，会首先尝试从缓存加载bean，首先从singletonObjects中获取，singletonObjects中存储的是BeanName->Bean Instance, 如果缓存为空，但该bean正在创建过程中（isSingletonCurrentlyInCreation）则尝试从singletonFactories中获取。这是因为spring创建单例bean的时候，存在循环依赖的问题。

比如创建bean a的时候发现bean a引用了bean b，此时会去创建bean b，但又发现bean b引用了bean c，所以此时会去创建bean c，在创建bean c的过程中发现bean c引用bean a。这三个bean就形成了一个环。

为了解决循环依赖的问题，spring采取了一种将创建的bean实例提早暴露加入到缓存中，一旦下一个bean创建的时候需要依赖上个bean，则直接使用ObjectFactory来获取bean。提前暴露bean实例到缓存的时机是在bean实例创建（调用构造方法）之后，初始化bean实例（属性注入）之前。

在AbstractAutowireCapableBeanFactory类

protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[] args) {...}

将允许提前暴露的单例bean提前加入singletonFactories中，这样就可以在创建依赖的时候避免循环依赖问题。

在从singletonFactories获取bean后，会将其存储到earlySingletonObjects中，然后从singletonFactories移除该bean，之后在要获取该bean就直接从earlySingletonObjects获取。这是因为从singletonFactories获取bean过程中需要调用singletonFactory.getObject()，这里还有一些操作，这样可以进一步提升性能。

```java
/**
 * Return the (raw) singleton object registered under the given name.
 * <p>Checks already instantiated singletons and also allows for an early
 * reference to a currently created singleton (resolving a circular reference).
 * @param beanName the name of the bean to look for
 * @param allowEarlyReference whether early references should be created or not
 * @return the registered singleton object, or {@code null} if none found
 */
@Nullable
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        synchronized (this.singletonObjects) {
            singletonObject = this.earlySingletonObjects.get(beanName);
            if (singletonObject == null && allowEarlyReference) {
                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    singletonObject = singletonFactory.getObject();
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return singletonObject;
}
```

isSingletonCurrentlyInCreation 判断对应的单例对象是否在创建中，当单例对象没有被初始化完全

allowEarlyReference 是否允许从singletonFactories中通过getObject拿到对象

org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton(java.lang.String, boolean) L179

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBean(java.lang.String, org.springframework.beans.factory.support.RootBeanDefinition, java.lang.Object[]) L495 L496

条件断点 **"ABean"**.equals(beanName) || **"BBean"**.equals(beanName) || **"CBean"**.equals(beanName)

分析getSingleton的整个过程，Spring首先从singletonObjects（一级缓存）中尝试获取，如果获取不到并且对象在创建中，则尝试从earlySingletonObjects(二级缓存)中获取，如果还是获取不到并且允许从singletonFactories通过getObject获取，则通过singletonFactory.getObject()(三级缓存)获取。如果获取到了则

```
this.earlySingletonObjects.put(beanName, singletonObject);
this.singletonFactories.remove(beanName);
```

则移除对应的singletonFactory,将singletonObject放入到earlySingletonObjects

```
addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));

protected Object getEarlyBeanReference(String beanName, RootBeanDefinition
mbd, Object bean) {
    Object exposedObject = bean;
    if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {
                SmartInstantiationAwareBeanPostProcessor ibp =
(SmartInstantiationAwareBeanPostProcessor) bp;
                exposedObject = ibp.getEarlyBeanReference(exposedObject,
beanName);
            }
        }
    }
    return exposedObject;
}
```

```
protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
    Assert.notNull(singletonFactory, "Singleton factory must not be null");
    synchronized (this.singletonObjects) {
        if (!this.singletonObjects.containsKey(beanName)) {
            this.singletonFactories.put(beanName, singletonFactory);
            this.earlySingletonObjects.remove(beanName);
            this.registeredSingletons.add(beanName);
        }
    }
}
```

此处就是解决循环依赖的关键，这段代码发生在createBeanInstance之后，也就是说单例对象此时已经被创建出来的。这个对象已经被生产出来了，虽然还不完美，但是已经能被人认出来了（根据对象引用能定位到堆中的对象），所以Spring此时将这个对象提前曝光出来让大家认识，让大家使用。

这样做有什么好处呢？让我们来分析一下“A的某个field或者setter依赖了B的实例对象，同时B的某个field或者setter依赖了A的实例对象”这种循环依赖的情况。A首先完成了初始化的第一步，并且将自己提前曝光到singletonFactories中，此时进行初始化的第二步，发现自己依赖对象B，此时就尝试去get(B)，发现B还没有被create，所以走create流程，B在初始化第一步的时候发现自己依赖了对象A，于是尝试get(A)，尝试一级缓存singletonObjects(肯定没有，因为A还没初始化完全)，尝试二级缓存earlySingletonObjects（也没有），尝试三级缓存singletonFactories，由于A通过ObjectFactory将自己提前曝光了，所以B能够通过ObjectFactory.getObject拿到A对象(虽然A还没有初始化完全，但是总比没有好呀)，B拿到A对象后顺利完成了初始化阶段1、2、3，完全初始化之后将自己放入到一级缓存singletonObjects中。此时返回A中，A此时能拿到B的对象顺利完成自己的初始化阶段2、3，最终A也完成了初始化，长大成人，进去了一级缓存singletonObjects中，而且更加幸运的是，由于B拿到了A的对象引用，所以B现在hold住的A对象也蜕变完美了

Spring MVC

成功的流程

获取到Bean，包装，增加拦截器，获取到HandlerExecutionChain

A前置拦截器
B前置拦截器
Handler处理

B后置拦截器
A后置拦截器

B请求处理完毕回调方法
A请求处理完毕回调方法

错误的流程

A前置拦截器
B前置拦截器

Handler处理，但是抛出了异常
（注意此处没有后置拦截器的调用）

异常处理（若找得倒异常处理方法则处理异常，
若没有则继续抛出）

B请求处理完毕回调方法
A请求处理完毕回调方法

拦截器返回失败的流程

A前置拦截器（通过）
B前置拦截器（不通过）
(没有Handler处理)
(没有后置拦截器调用)

A请求处理完毕回调方法

参考：

https://www.jianshu.com/p/6c359768b1dc

https://my.oschina.net/wangzhenchao/blog/915897