通过单例模式可以保证系统中一个类只有一个实例。spring依赖注入时，使用了多重判断加锁的单例模式。

Spring的依赖注入Bean实例默认是单例的。

Spring的依赖注入（包括lazy-init方式）都是发生在AbstractBeanFactory的getBean里。getBean的doGetBean方法调用getSingleton进行bean的创建。lazy-init方式，在容器初始化时候进行调用，非lazy-init方式，在用户向容器第一次索要bean时进行调用。

同步线程安全的单例核心代码：

@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;  
}

spring依赖注入时，使用了双重判断加锁的单例模式，

1. 首先从缓存中获取bean实例
2. 如果为null，对缓存加锁，然后再从缓存中获取bean
3. 如果继续为null，就创建一个bean。

这样双重判断，能够避免在加锁的瞬间，有其他依赖注入引发bean实例的创建，从而造成重复创建的结果。

在这里Spring并没有使用私有构造方法来创建bean，而是通过singletonFactory.getObject()返回具体beanName对应的ObjectFactory来创建bean。我们一路跟踪下去，发现实际上是调用了AbstractAutowireCapableBeanFactory的doCreateBean方法，返回了BeanWrapper包装并创建的bean实例。

（ObjectFactory主要检查是否有用户定义的BeanPostProcessor后处理内容，并在创建bean时进行处理，如果没有，就直接返回bean本身）

创建bean实例返回给BeanWrapper；

addSingletonFactory增加beanName和ObjectFactory的键值对应关系。

protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd,

final @Nullable Object[] args)  
 throws BeanCreationException {  
  
 // Instantiate the bean.  
 BeanWrapper instanceWrapper = null;  
 if (mbd.isSingleton()) {  
 instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);  
 }  
 if (instanceWrapper == null) {  
 instanceWrapper = createBeanInstance(beanName, mbd, args);  
 }  
 final Object bean = instanceWrapper.getWrappedInstance();  
 Class<?> beanType = instanceWrapper.getWrappedClass();  
 if (beanType != NullBean.class) {  
 mbd.resolvedTargetType = beanType;  
 }  
  
 // Allow post-processors to modify the merged bean definition.  
 synchronized (mbd.postProcessingLock) {  
 if (!mbd.postProcessed) {  
 try {  
 applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
 "Post-processing of merged bean definition failed", ex);  
 }  
 mbd.postProcessed = true;  
 }  
 }  
  
 // 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) {  
 if (logger.isTraceEnabled()) {  
 logger.trace("Eagerly caching bean '" + beanName +  
 "' to allow for resolving potential circular references");  
 }  
 addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));  
 }  
  
 // Initialize the bean instance.  
 Object exposedObject = bean;  
 try {  
 populateBean(beanName, mbd, instanceWrapper);  
 exposedObject = initializeBean(beanName, exposedObject, mbd);  
 }  
 catch (Throwable ex) {  
 if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {  
 throw (BeanCreationException) ex;  
 }  
 else {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);  
 }  
 }  
  
 if (earlySingletonExposure) {  
 Object earlySingletonReference = getSingleton(beanName, false);  
 if (earlySingletonReference != null) {  
 if (exposedObject == bean) {  
 exposedObject = earlySingletonReference;  
 }  
 else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {  
 String[] dependentBeans = getDependentBeans(beanName);  
 Set<String> actualDependentBeans = new LinkedHashSet<>(dependentBeans.length);  
 for (String dependentBean : dependentBeans) {  
 if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {  
 actualDependentBeans.add(dependentBean);  
 }  
 }  
 if (!actualDependentBeans.isEmpty()) {  
 throw new BeanCurrentlyInCreationException(beanName,  
 "Bean with name '" + beanName + "' has been injected into other beans [" +  
 StringUtils.*collectionToCommaDelimitedString*(actualDependentBeans) +  
 "] in its raw version as part of a circular reference, but has eventually been " +  
 "wrapped. This means that said other beans do not use the final version of the " +  
 "bean. This is often the result of over-eager type matching - consider using " +  
 "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");  
 }  
 }  
 }  
 }  
  
 // Register bean as disposable.  
 try {  
 registerDisposableBeanIfNecessary(beanName, bean, mbd);  
 }  
 catch (BeanDefinitionValidationException ex) {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);  
 }  
  
 return exposedObject;  
}

getEarlyBeanReference获取bean的所有后处理器，并进行处理。如果是SmartInstantiationAwareBeanPostProcessor类型，就进行处理，如果没有相关处理内容，就返回默认的实现。

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;  
}