课程目标

1、通过分析Spring源码，深刻掌握核心原理和设计思想。
2、通过本课的学习，完全掌握Spring DI的重要细节。
3、手绘Spring DI运行时序图。

内容定位

1、Spring使用不熟练者不适合学习本章内容。
2、先掌握执行流程，再理解设计思想，这个过程至少要花1个月。
3、Spring 源码非常经典，体系也非常庞大，看一遍是远远不够的。

1. Spring自动装配之依赖注入

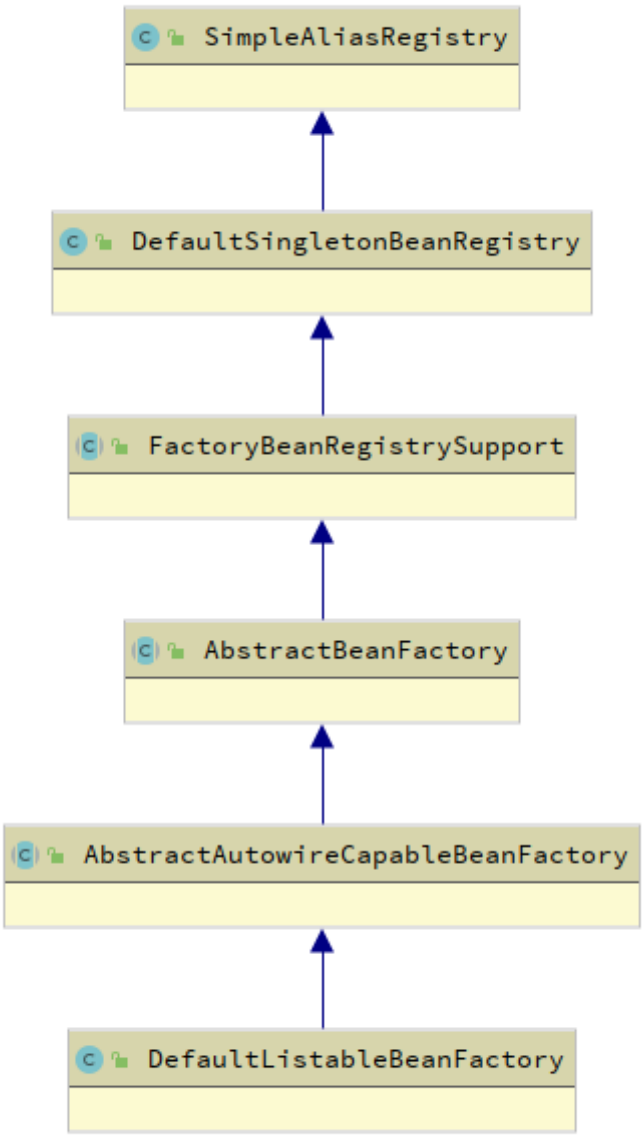
1.1. 依赖注入发生的时间

当Spring IoC容器完成了Bean定义资源的定位、载入和解析注册以后IoC容器中已经管理了Bean定义的相关数据，但是此时IoC 容器还没有对所管理的Bean进行依赖注入，依赖注入在以下两种情况发生：

1）、用户第一次调用getBean()方法时，IoC容器触发依赖注入。

2）、当用户在配置文件中将 `<bean>` 元素配置了lazy-init=false属性，即让容器在解析注册Bean定义时进行预实例化，触发依赖注入。

BeanFactory接口定义了Spring IoC 容器的基本功能规范，是Spring IoC容器所应遵守的最底层和最基本的编程规范。BeanFactory 接口中定义了几个getBean()方法，就是用户向IoC容器索取管理的Bean的方法，我们通过分析其子类的具体实现，理解 Spring IoC容器在用户索取Bean时如何完成依赖注入。



在BeanFactory 中我们可以看到getBean（String..）方法，但它具体实现在AbstractBeanFactory中。

1.2. 寻找获取Bean的入口

AbstractBeanFactory的getBean()相关方法的源码如下：

```
    //获取IOC容器中指定名称的Bean
    @Override
    public Object getBean(String name) throws BeansException {
        //doGetBean才是真正向IoC容器获取被管理Bean的过程
        return doGetBean(name, null, null, false);
    }

    //获取IOC容器中指定名称和类型的Bean
    @Override
    public <T> T getBean(String name, @Nullable Class<T> requiredType) throws BeansExce
        //doGetBean才是真正向IoC容器获取被管理Bean的过程
        return doGetBean(name, requiredType, null, false);
    }

    //获取IOC容器中指定名称和参数的Bean
    @Override
    public Object getBean(String name, Object... args) throws BeansException {
        //doGetBean才是真正向IoC容器获取被管理Bean的过程
        return doGetBean(name, null, args, false);
    }

    /**
     * Return an instance, which may be shared or independent, of the specified bean.
     * @param name the name of the bean to retrieve
     * @param requiredType the required type of the bean to retrieve
     * @param args arguments to use when creating a bean instance using explicit argume
     * (only applied when creating a new instance as opposed to retrieving an existing
     * @return an instance of the bean
     * @throws BeansException if the bean could not be created
     */
    //获取IOC容器中指定名称、类型和参数的Bean
    public <T> T getBean(String name, @Nullable Class<T> requiredType, @Nullable Object
            throws BeansException {
        //doGetBean才是真正向IoC容器获取被管理Bean的过程
        return doGetBean(name, requiredType, args, false);
```

```
    }

    /**
     * Return an instance, which may be shared or independent, of the specified bean.
     * @param name the name of the bean to retrieve
     * @param requiredType the required type of the bean to retrieve
     * @param args arguments to use when creating a bean instance using explicit argume
     * (only applied when creating a new instance as opposed to retrieving an existing
     * @param typeCheckOnly whether the instance is obtained for a type check,
     * not for actual use
     * @return an instance of the bean
     * @throws BeansException if the bean could not be created
     */
    @SuppressWarnings("unchecked")
    //真正实现向IOC容器获取Bean的功能，也是触发依赖注入功能的地方
    protected <T> T doGetBean(final String name, @Nullable final Class<T> requiredType,
            @Nullable final Object[] args, boolean typeCheckOnly) throws BeansExceptior

        //根据指定的名称获取被管理Bean的名称，剥离指定名称中对容器的相关依赖
        //如果指定的是别名，将别名转换为规范的Bean名称
        final String beanName = transformedBeanName(name);
        Object bean;

        // Eagerly check singleton cache for manually registered singletons.
        //先从缓存中取是否已经有被创建过的单态类型的Bean
        //对于单例模式的Bean整个IOC容器中只创建一次，不需要重复创建
        Object sharedInstance = getSingleton(beanName);
        //IOC容器创建单例模式Bean实例对象
        if (sharedInstance != null && args == null) {
            if (logger.isDebugEnabled()) {
                //如果指定名称的Bean在容器中已有单例模式的Bean被创建
                //直接返回已经创建的Bean
                if (isSingletonCurrentlyInCreation(beanName)) {
                    logger.debug("Returning eagerly cached instance of singleton bean '
                            "' that is not fully initialized yet - a consequence of a c
                }
                else {
                    logger.debug("Returning cached instance of singleton bean '" + bear
                }
            }
            //获取给定Bean的实例对象，主要是完成FactoryBean的相关处理
            //注意：BeanFactory是管理容器中Bean的工厂，而FactoryBean是
            //创建创建对象的工厂Bean，两者之间有区别
```

```
            bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
        }

        else {
            // Fail if we're already creating this bean instance:
            // We're assumably within a circular reference.
            //缓存没有正在创建的单例模式Bean
            //缓存中已经有已经创建的原型模式Bean
            //但是由于循环引用的问题导致实例化对象失败
            if (isPrototypeCurrentlyInCreation(beanName)) {
                throw new BeanCurrentlyInCreationException(beanName);
            }

            // Check if bean definition exists in this factory.
            //对IOC容器中是否存在指定名称的BeanDefinition进行检查，首先检查是否
            //能在当前的BeanFactory中获取的所需要的Bean，如果不能则委托当前容器
            //的父级容器去查找，如果还是找不到则沿着容器的继承体系向父级容器查找
            BeanFactory parentBeanFactory = getParentBeanFactory();
            //当前容器的父级容器存在，且当前容器中不存在指定名称的Bean
            if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
                // Not found -> check parent.
                //解析指定Bean名称的原始名称
                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.
                    //委派父级容器根据指定名称和显式的参数查找
                    return (T) parentBeanFactory.getBean(nameToLookup, args);
                }
                else {
                    // No args -> delegate to standard getBean method.
                    //委派父级容器根据指定名称和类型查找
                    return parentBeanFactory.getBean(nameToLookup, requiredType);
                }
            }

            //创建的Bean是否需要进行类型验证，一般不需要
            if (!typeCheckOnly) {
                //向容器标记指定的Bean已经被创建
                markBeanAsCreated(beanName);
```

```
122             }
123 
124             try {
125                 //根据指定Bean名称获取其父级的Bean定义
126                 //主要解决Bean继承时子类合并父类公共属性问题
127                 final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
128                 checkMergedBeanDefinition(mbd, beanName, args);
129 
130                 // Guarantee initialization of beans that the current bean depends on.
131                 //获取当前Bean所有依赖Bean的名称
132                 String[] dependsOn = mbd.getDependsOn();
133                 //如果当前Bean有依赖Bean
134                 if (dependsOn != null) {
135                     for (String dep : dependsOn) {
136                         if (isDependent(beanName, dep)) {
137                             throw new BeanCreationException(mbd.getResourceDescription(
138                                     "Circular depends-on relationship between '" + bean
139                         }
140                         //递归调用getBean方法，获取当前Bean的依赖Bean
141                         registerDependentBean(dep, beanName);
142                         //把被依赖Bean注册给当前依赖的Bean
143                         getBean(dep);
144                     }
145                 }
146 
147                 // Create bean instance.
148                 //创建单例模式Bean的实例对象
149                 if (mbd.isSingleton()) {
150                     //这里使用了一个匿名内部类，创建Bean实例对象，并且注册给所依赖的对象
151                     sharedInstance = getSingleton(beanName, () -> {
152                         try {
153                             //创建一个指定Bean实例对象，如果有父级继承，则合并子类和父类
154                             return createBean(beanName, mbd, args);
155                         }
156                         catch (BeansException ex) {
157                             // Explicitly remove instance from singleton cache: It migh
158                             // eagerly by the creation process, to allow for circular r
159                             // Also remove any beans that received a temporary referenc
160                             //显式地从容器单例模式Bean缓存中清除实例对象
161                             destroySingleton(beanName);
162                             throw ex;
163                         }
164                     });
```

```
                    //获取给定Bean的实例对象
                    bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd
                }

                //IOC容器创建原型模式Bean实例对象
                else if (mbd.isPrototype()) {
                    // It's a prototype -> create a new instance.
                    //原型模式(Prototype)是每次都会创建一个新的对象
                    Object prototypeInstance = null;
                    try {
                        //回调beforePrototypeCreation方法，默认的功能是注册当前创建的原
                        beforePrototypeCreation(beanName);
                        //创建指定Bean对象实例
                        prototypeInstance = createBean(beanName, mbd, args);
                    }
                    finally {
                        //回调afterPrototypeCreation方法，默认的功能告诉IOC容器指定Bean
                        afterPrototypeCreation(beanName);
                    }
                    //获取给定Bean的实例对象
                    bean = getObjectForBeanInstance(prototypeInstance, name, beanName,
                }

                //要创建的Bean既不是单例模式，也不是原型模式，则根据Bean定义资源中
                //配置的生命周期范围，选择实例化Bean的合适方法，这种在Web应用程序中
                //比较常用，如：request、session、application等生命周期
                else {
                    String scopeName = mbd.getScope();
                    final Scope scope = this.scopes.get(scopeName);
                    //Bean定义资源中没有配置生命周期范围，则Bean定义不合法
                    if (scope == null) {
                        throw new IllegalStateException("No Scope registered for scope
                    }
                    try {
                        //这里又使用了一个匿名内部类，获取一个指定生命周期范围的实例
                        Object scopedInstance = scope.get(beanName, () -> {
                            beforePrototypeCreation(beanName);
                            try {
                                return createBean(beanName, mbd, args);
                            }
                            finally {
                                afterPrototypeCreation(beanName);
                            }
```

```
                        });
                        //获取给定Bean的实例对象
                        bean = getObjectForBeanInstance(scopedInstance, name, beanName,
                    }
                    catch (IllegalStateException ex) {
                        throw new BeanCreationException(beanName,
                                "Scope '" + scopeName + "' is not active for the currer
                                "defining a scoped proxy for this bean if you intend to
                                ex);
                    }
                }
            }
            catch (BeansException ex) {
                cleanupAfterBeanCreationFailure(beanName);
                throw ex;
            }
        }

        // Check if required type matches the type of the actual bean instance.
        //对创建的Bean实例对象进行类型检查
        if (requiredType != null && !requiredType.isInstance(bean)) {
            try {
                T convertedBean = getTypeConverter().convertIfNecessary(bean, requiredT
                if (convertedBean == null) {
                    throw new BeanNotOfRequiredTypeException(name, requiredType, bean.g
                }
                return convertedBean;
            }
            catch (TypeMismatchException ex) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Failed to convert bean '" + name + "' to required typ
                            ClassUtils.getQualifiedName(requiredType) + "'", ex);
                }
                throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getCl
            }
        }
        return (T) bean;
    }
```

通过上面对向IoC容器获取Bean方法的分析，我们可以看到在Spring中，如果Bean定义的单例模式（Singleton），则容器在创建之前先从缓存中查找，以确保整个容器中只存在一个实例对象。如果

Bean定义的是原型模式（Prototype），则容器每次都会创建一个新的实例对象。除此之外，Bean定义还可以扩展为指定其生命周期范围。

上面的源码只是定义了根据Bean定义的模式采取的不同创建Bean实例对象的策略具体的Bean实例对象的创建过程由实现了ObjectFactory 接口的匿名内部类的createBean()方法完成，ObjectFactory使用委派模式，具体的Bean实例创建过程交由其实现类AbstractAutowireCapableBeanFactory完成，我们继续分析AbstractAutowireCapableBeanFactory的createBean()方法的源码，理解其创建Bean实例的具体实现过程。

1.3. 开始实例化

AbstractAutowireCapableBeanFactory 类实现了ObjectFactory接口，创建容器指定的Bean实例对象，同时还对创建的Bean实例对象进行初始化处理。其创建Bean实例对象的方法源码如下：

```
    //创建Bean实例对象
    @Override
    protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Obje
            throws BeanCreationException {

        if (logger.isDebugEnabled()) {
            logger.debug("Creating instance of bean '" + beanName + "'");
        }
        RootBeanDefinition mbdToUse = mbd;

        // Make sure bean class is actually resolved at this point, and
        // clone the bean definition in case of a dynamically resolved Class
        // which cannot be stored in the shared merged bean definition.
        //判断需要创建的Bean是否可以实例化，即是否可以通过当前的类加载器加载
        Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
        if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != r
            mbdToUse = new RootBeanDefinition(mbd);
            mbdToUse.setBeanClass(resolvedClass);
        }

        // Prepare method overrides.
        //校验和准备Bean中的方法覆盖
        try {
            mbdToUse.prepareMethodOverrides();
        }
        catch (BeanDefinitionValidationException ex) {
            throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),
                    beanName, "Validation of method overrides failed", ex);
```

```
29          }
30
31          try {
32              // Give BeanPostProcessors a chance to return a proxy instead of the target
33              //如果Bean配置了初始化前和初始化后的处理器，则试图返回一个需要创建Bean的代理
34              Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
35              if (bean != null) {
36                  return bean;
37              }
38          }
39          catch (Throwable ex) {
40              throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName
41                      "BeanPostProcessor before instantiation of bean failed", ex);
42          }
43
44          try {
45              //创建Bean的入口
46              Object beanInstance = doCreateBean(beanName, mbdToUse, args);
47              if (logger.isDebugEnabled()) {
48                  logger.debug("Finished creating instance of bean '" + beanName + "'");
49              }
50              return beanInstance;
51          }
52          catch (BeanCreationException ex) {
53              // A previously detected exception with proper bean creation context alread
54              throw ex;
55          }
56          catch (ImplicitlyAppearedSingletonException ex) {
57              // An IllegalStateException to be communicated up to DefaultSingletonBeanRe
58              throw ex;
59          }
60          catch (Throwable ex) {
61              throw new BeanCreationException(
62                      mbdToUse.getResourceDescription(), beanName, "Unexpected exception
63          }
64      }
65
66      /**
67       * Actually create the specified bean. Pre-creation processing has already happened
68       * at this point, e.g. checking {@code postProcessBeforeInstantiation} callbacks.
69       * <p>Differentiates between default bean instantiation, use of a
70       * factory method, and autowiring a constructor.
71       * @param beanName the name of the bean
```

```
     * @param mbd the merged bean definition for the bean
     * @param args explicit arguments to use for constructor or factory method invocati
     * @return a new instance of the bean
     * @throws BeanCreationException if the bean could not be created
     * @see #instantiateBean
     * @see #instantiateUsingFactoryMethod
     * @see #autowireConstructor
     */
    //真正创建Bean的方法
    protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd,
            throws BeanCreationException {

        // Instantiate the bean.
        //封装被创建的Bean对象
        BeanWrapper instanceWrapper = null;
        if (mbd.isSingleton()) {
            instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
        }
        if (instanceWrapper == null) {
            instanceWrapper = createBeanInstance(beanName, mbd, args);
        }
        final Object bean = instanceWrapper.getWrappedInstance();
        //获取实例化对象的类型
        Class<?> beanType = instanceWrapper.getWrappedClass();
        if (beanType != NullBean.class) {
            mbd.resolvedTargetType = beanType;
        }

        // Allow post-processors to modify the merged bean definition.
        //调用PostProcessor后置处理器
        synchronized (mbd.postProcessingLock) {
            if (!mbd.postProcessed) {
                try {
                    applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
                }
                catch (Throwable ex) {
                    throw new BeanCreationException(mbd.getResourceDescription(), beanN
                            "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.
        //向容器中缓存单例模式的Bean对象，以防循环引用
        boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularRefere
                isSingletonCurrentlyInCreation(beanName));
        if (earlySingletonExposure) {
            if (logger.isDebugEnabled()) {
                logger.debug("Eagerly caching bean '" + beanName +
                        "' to allow for resolving potential circular references");
            }
            //这里是一个匿名内部类，为了防止循环引用，尽早持有对象的引用
            addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, be
        }

        // Initialize the bean instance.
        //Bean对象的初始化，依赖注入在此触发
        //这个exposedObject在初始化完成之后返回作为依赖注入完成后的Bean
        Object exposedObject = bean;
        try {
            //将Bean实例对象封装，并且Bean定义中配置的属性值赋值给实例对象
            populateBean(beanName, mbd, instanceWrapper);
            //初始化Bean对象
            exposedObject = initializeBean(beanName, exposedObject, mbd);
        }
        catch (Throwable ex) {
            if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationEx
                throw (BeanCreationException) ex;
            }
            else {
                throw new BeanCreationException(
                        mbd.getResourceDescription(), beanName, "Initialization of bean
            }
        }

        if (earlySingletonExposure) {
            //获取指定名称的已注册的单例模式Bean对象
            Object earlySingletonReference = getSingleton(beanName, false);
            if (earlySingletonReference != null) {
                //根据名称获取的已注册的Bean和正在实例化的Bean是同一个
                if (exposedObject == bean) {
                    //当前实例化的Bean初始化完成
                    exposedObject = earlySingletonReference;
                }
```

```
                //当前Bean依赖其他Bean，并且当发生循环引用时不允许新创建实例对象
                else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(bea
                    String[] dependentBeans = getDependentBeans(beanName);
                    Set<String> actualDependentBeans = new LinkedHashSet<>(dependentBea
                    //获取当前Bean所依赖的其他Bean
                    for (String dependentBean : dependentBeans) {
                        //对依赖Bean进行类型检查
                        if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                            actualDependentBeans.add(dependentBean);
                        }
                    }
                    if (!actualDependentBeans.isEmpty()) {
                        throw new BeanCurrentlyInCreationException(beanName,
                                "Bean with name '" + beanName + "' has been injected ir
                                StringUtils.collectionToCommaDelimitedString(actualDepe
                                "] in its raw version as part of a circular reference,
                                "wrapped. This means that said other beans do not use t
                                "bean. This is often the result of over-eager type matc
                                "'getBeanNamesOfType' with the 'allowEagerInit' flag tu
                    }
                }
            }
        }

        // Register bean as disposable.
        //注册完成依赖注入的Bean
        try {
            registerDisposableBeanIfNecessary(beanName, bean, mbd);
        }
        catch (BeanDefinitionValidationException ex) {
            throw new BeanCreationException(
                    mbd.getResourceDescription(), beanName, "Invalid destruction signat
        }

        return exposedObject;
    }
```

通过上面的源码注释，我们看到具体的依赖注入实现其实就在以下两个方法中：

1）、createBeanInstance()方法，生成Bean 所包含的java对象实例。

2）、populateBean()方法，对Bean 属性的依赖注入进行处理。

下面继续分析这两个方法的代码实现。

1.4. 选择Bean实例化策略

在createBeanInstance()方法中，根据指定的初始化策略，使用简单工厂、工厂方法或者容器的自动装配特性生成Java实例对象，创建对象的源码如下：

```
    //创建Bean的实例对象
    protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, @
        // Make sure bean class is actually resolved at this point.
        //检查确认Bean是可实例化的
        Class<?> beanClass = resolveBeanClass(mbd, beanName);

        //使用工厂方法对Bean进行实例化
        if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.i
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Bean class isn't public, and non-public access not allowed: " + be
        }

        Supplier<?> instanceSupplier = mbd.getInstanceSupplier();
        if (instanceSupplier != null) {
            return obtainFromSupplier(instanceSupplier, beanName);
        }

        if (mbd.getFactoryMethodName() != null)  {
            //调用工厂方法实例化
            return instantiateUsingFactoryMethod(beanName, mbd, args);
        }

        // Shortcut when re-creating the same bean...
        //使用容器的自动装配方法进行实例化
        boolean resolved = false;
        boolean autowireNecessary = false;
        if (args == null) {
            synchronized (mbd.constructorArgumentLock) {
                if (mbd.resolvedConstructorOrFactoryMethod != null) {
                    resolved = true;
                    autowireNecessary = mbd.constructorArgumentsResolved;
                }
            }
        }
        if (resolved) {
            if (autowireNecessary) {
                //配置了自动装配属性，使用容器的自动装配实例化
```

```
                //容器的自动装配是根据参数类型匹配Bean的构造方法
                return autowireConstructor(beanName, mbd, null, null);
            }
            else {
                //使用默认的无参构造方法实例化
                return instantiateBean(beanName, mbd);
            }
        }

        // Need to determine the constructor...
        //使用Bean的构造方法进行实例化
        Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass,
        if (ctors != null ||
                mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_CONSTRUCTO
                mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args))  {
            //使用容器的自动装配特性，调用匹配的构造方法实例化
            return autowireConstructor(beanName, mbd, ctors, args);
        }

        // No special handling: simply use no-arg constructor.
        //使用默认的无参构造方法实例化
        return instantiateBean(beanName, mbd);
    }

    //使用默认的无参构造方法实例化Bean对象
    protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefiniti
        try {
            Object beanInstance;
            final BeanFactory parent = this;
            //获取系统的安全管理接口，JDK标准的安全管理API
            if (System.getSecurityManager() != null) {
                //这里是一个匿名内置类，根据实例化策略创建实例对象
                beanInstance = AccessController.doPrivileged((PrivilegedAction<Object>)
                        getInstantiationStrategy().instantiate(mbd, beanName, parent),
                        getAccessControlContext());
            }
            else {
                //将实例化的对象封装起来
                beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, pa
            }
            BeanWrapper bw = new BeanWrapperImpl(beanInstance);
            initBeanWrapper(bw);
            return bw;
```

```
        }
        catch (Throwable ex) {
            throw new BeanCreationException(
                    mbd.getResourceDescription(), beanName, "Instantiation of bean fail
        }
    }
```

经过对上面的代码分析，我们可以看出，对使用工厂方法和自动装配特性的Bean的实例化相当比较清楚，调用相应的工厂方法或者参数匹配的构造方法即可完成实例化对象的工作，但是对于我们最常使用的默认无参构造方法就需要使用相应的初始化策略（JDK的反射机制或者CGLib）来进行初始化了，在方法getInstantiationStrategy().instantiate()中就具体实现类使用初始策略实例化对象。

1.5. 执行Bean实例化

在使用默认的无参构造方法创建Bean的实例化对象时，方法getInstantiationStrategy().instantiate())调用了SimpleInstantiationStrategy类中的实例化Bean的方法，其源码如下：

```
    //使用初始化策略实例化Bean对象
    @Override
    public Object instantiate(RootBeanDefinition bd, @Nullable String beanName, BeanFac
        // Don't override the class with CGLIB if no overrides.
        //如果Bean定义中没有方法覆盖，则就不需要CGLIB父类类的方法
        if (!bd.hasMethodOverrides()) {
            Constructor<?> constructorToUse;
            synchronized (bd.constructorArgumentLock) {
                //获取对象的构造方法或工厂方法
                constructorToUse = (Constructor<?>) bd.resolvedConstructorOrFactoryMeth
                //如果没有构造方法且没有工厂方法
                if (constructorToUse == null) {
                    //使用JDK的反射机制，判断要实例化的Bean是否是接口
                    final Class<?> clazz = bd.getBeanClass();
                    if (clazz.isInterface()) {
                        throw new BeanInstantiationException(clazz, "Specified class is
                    }
                    try {
                        if (System.getSecurityManager() != null) {
                            //这里是一个匿名内置类，使用反射机制获取Bean的构造方法
                            constructorToUse = AccessController.doPrivileged(
                                    (PrivilegedExceptionAction<Constructor<?>>) () -> c
                        }
                        else {
```

```
                        constructorToUse =    clazz.getDeclaredConstructor();
                    }
                    bd.resolvedConstructorOrFactoryMethod = constructorToUse;
                }
                catch (Throwable ex) {
                    throw new BeanInstantiationException(clazz, "No default constru
                }
            }
        }
        //使用BeanUtils实例化，通过反射机制调用”构造方法.newInstance(arg)”来进行实例
        return BeanUtils.instantiateClass(constructorToUse);
    }
    else {
        // Must generate CGLIB subclass.
        //使用CGLIB来实例化对象
        return instantiateWithMethodInjection(bd, beanName, owner);
    }
}
```

通过上面的代码分析，我们看到了如果Bean有方法被覆盖了，则使用JDK的反射机制进行实例化，否则，使用CGLib进行实例化。

instantiateWithMethodInjection()方法调用SimpleInstantiationStrategy的子类CGLibSubclassingInstantiationStrategy 使用CGLib来进行初始化，其源码如下：

```
    //使用CGLIB进行Bean对象实例化
    public Object instantiate(@Nullable Constructor<?> ctor, @Nullable Object... ar
        //创建代理子类
        Class<?> subclass = createEnhancedSubclass(this.beanDefinition);
        Object instance;
        if (ctor == null) {
            instance = BeanUtils.instantiateClass(subclass);
        }
        else {
            try {
                Constructor<?> enhancedSubclassConstructor = subclass.getConstructo
                instance = enhancedSubclassConstructor.newInstance(args);
            }
            catch (Exception ex) {
                throw new BeanInstantiationException(this.beanDefinition.getBeanCla
                        "Failed to invoke constructor for CGLIB enhanced subclass [
            }
```

```
            }
            // SPR-10785: set callbacks directly on the instance instead of in the
            // enhanced class (via the Enhancer) in order to avoid memory leaks.
            Factory factory = (Factory) instance;
            factory.setCallbacks(new Callback[] {NoOp.INSTANCE,
                    new LookupOverrideMethodInterceptor(this.beanDefinition, this.owner
                    new ReplaceOverrideMethodInterceptor(this.beanDefinition, this.owne
            return instance;
        }

        /**
         * Create an enhanced subclass of the bean class for the provided bean
         * definition, using CGLIB.
         */
        private Class<?> createEnhancedSubclass(RootBeanDefinition beanDefinition) {
            //CGLIB中的类
            Enhancer enhancer = new Enhancer();
            //将Bean本身作为其基类
            enhancer.setSuperclass(beanDefinition.getBeanClass());
            enhancer.setNamingPolicy(SpringNamingPolicy.INSTANCE);
            if (this.owner instanceof ConfigurableBeanFactory) {
                ClassLoader cl = ((ConfigurableBeanFactory) this.owner).getBeanClassLoa
                enhancer.setStrategy(new ClassLoaderAwareGeneratorStrategy(cl));
            }
            enhancer.setCallbackFilter(new MethodOverrideCallbackFilter(beanDefinition)
            enhancer.setCallbackTypes(CALLBACK_TYPES);
            //使用CGLIB的createClass方法生成实例对象
            return enhancer.createClass();
        }
    }
```

CGLib是一个常用的字节码生成器的类库，它提供了一系列API实现Java字节码的生成和转换功能。我们在学习JDK的动态代理时都知道，JDK的动态代理只能针对接口，如果一个类没有实现任何接口，要对其进行动态代理只能使用CGLib。

1.6. 准备依赖注入

在前面的分析中我们已经了解到Bean的依赖注入主要分为两个步骤，首先调用createBeanInstance()方法生成Bean所包含的Java对象实例。然后，调用populateBean()方法，对 Bean 属性的依赖注入进行处理。

上面我们已经分析了容器初始化生成Bean所包含的Java实例对象的过程，现在我们继续分析生成对象后，Spring IoC容器是如何将Bean的属性依赖关系注入Bean实例对象中并设置好的，回到

AbstractAutowireCapableBeanFactory的 populateBean()方法，对属性依赖注入的代码如下：

```
    //将Bean属性设置到生成的实例对象上
    protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable Bear
        if (bw == null) {
            if (mbd.hasPropertyValues()) {
                throw new BeanCreationException(
                        mbd.getResourceDescription(), beanName, "Cannot apply property
            }
            else {
                // Skip property population phase for null instance.
                return;
            }
        }

        // Give any InstantiationAwareBeanPostProcessors the opportunity to modify the
        // state of the bean before properties are set. This can be used, for example,
        // to support styles of field injection.
        boolean continueWithPropertyPopulation = true;

        if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
            for (BeanPostProcessor bp : getBeanPostProcessors()) {
                if (bp instanceof InstantiationAwareBeanPostProcessor) {
                    InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPo
                    if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), bea
                        continueWithPropertyPopulation = false;
                        break;
                    }
                }
            }
        }

        if (!continueWithPropertyPopulation) {
            return;
        }
        //获取容器在解析Bean定义资源时为BeanDefiniton中设置的属性值
        PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null)

        //对依赖注入处理，首先处理autowiring自动装配的依赖注入
        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
                mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
            MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
```

```
41
42              // Add property values based on autowire by name if applicable.
43              //根据Bean名称进行autowiring自动装配处理
44              if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
45                  autowireByName(beanName, mbd, bw, newPvs);
46              }
47
48              // Add property values based on autowire by type if applicable.
49              //根据Bean类型进行autowiring自动装配处理
50              if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
51                  autowireByType(beanName, mbd, bw, newPvs);
52              }
53
54              pvs = newPvs;
55          }
56
57          //对非autowiring的属性进行依赖注入处理
58
59          boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();
60          boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDE
61
62          if (hasInstAwareBpps || needsDepCheck) {
63              if (pvs == null) {
64                  pvs = mbd.getPropertyValues();
65              }
66              PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCh
67              if (hasInstAwareBpps) {
68                  for (BeanPostProcessor bp : getBeanPostProcessors()) {
69                      if (bp instanceof InstantiationAwareBeanPostProcessor) {
70                          InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBe
71                          pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWra
72                          if (pvs == null) {
73                              return;
74                          }
75                      }
76                  }
77              }
78              if (needsDepCheck) {
79                  checkDependencies(beanName, mbd, filteredPds, pvs);
80              }
81          }
82
83          if (pvs != null) {
```

```
84              //对属性进行注入
85              applyPropertyValues(beanName, mbd, bw, pvs);
86          }
87      }
88
89      //解析并注入依赖属性的过程
90      protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper
91          if (pvs.isEmpty()) {
92              return;
93          }
94
95          //封装属性值
96          MutablePropertyValues mpvs = null;
97          List<PropertyValue> original;
98
99          if (System.getSecurityManager() != null) {
100             if (bw instanceof BeanWrapperImpl) {
101                 //设置安全上下文，JDK安全机制
102                 ((BeanWrapperImpl) bw).setSecurityContext(getAccessControlContext());
103             }
104         }
105
106         if (pvs instanceof MutablePropertyValues) {
107             mpvs = (MutablePropertyValues) pvs;
108             //属性值已经转换
109             if (mpvs.isConverted()) {
110                 // Shortcut: use the pre-converted values as-is.
111                 try {
112                     //为实例化对象设置属性值
113                     bw.setPropertyValues(mpvs);
114                     return;
115                 }
116                 catch (BeansException ex) {
117                     throw new BeanCreationException(
118                             mbd.getResourceDescription(), beanName, "Error setting prop
119                 }
120             }
121             //获取属性值对象的原始类型值
122             original = mpvs.getPropertyValueList();
123         }
124         else {
125             original = Arrays.asList(pvs.getPropertyValues());
126         }
```

```
127
128         //获取用户自定义的类型转换
129         TypeConverter converter = getCustomTypeConverter();
130         if (converter == null) {
131             converter = bw;
132         }
133         //创建一个Bean定义属性值解析器，将Bean定义中的属性值解析为Bean实例对象的实际值
134         BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver(thi
135
136         // Create a deep copy, resolving any references for values.
137
138         //为属性的解析值创建一个拷贝，将拷贝的数据注入到实例对象中
139         List<PropertyValue> deepCopy = new ArrayList<>(original.size());
140         boolean resolveNecessary = false;
141         for (PropertyValue pv : original) {
142             //属性值不需要转换
143             if (pv.isConverted()) {
144                 deepCopy.add(pv);
145             }
146             //属性值需要转换
147             else {
148                 String propertyName = pv.getName();
149                 //原始的属性值，即转换之前的属性值
150                 Object originalValue = pv.getValue();
151                 //转换属性值，例如将引用转换为IOC容器中实例化对象引用
152                 Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, origir
153                 //转换之后的属性值
154                 Object convertedValue = resolvedValue;
155                 //属性值是否可以转换
156                 boolean convertible = bw.isWritableProperty(propertyName) &&
157                         !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);
158                 if (convertible) {
159                     //使用用户自定义的类型转换器转换属性值
160                     convertedValue = convertForProperty(resolvedValue, propertyName, bw
161                 }
162                 // Possibly store converted value in merged bean definition,
163                 // in order to avoid re-conversion for every created bean instance.
164                 //存储转换后的属性值，避免每次属性注入时的转换工作
165                 if (resolvedValue == originalValue) {
166                     if (convertible) {
167                         //设置属性转换之后的值
168                         pv.setConvertedValue(convertedValue);
169                     }
```

```
                    deepCopy.add(pv);
                }
                //属性是可转换的，且属性原始值是字符串类型，且属性的原始类型值不是
                //动态生成的字符串，且属性的原始值不是集合或者数组类型
                else if (convertible && originalValue instanceof TypedStringValue &&
                        !((TypedStringValue) originalValue).isDynamic() &&
                        !(convertedValue instanceof Collection || ObjectUtils.isArray(c
                    pv.setConvertedValue(convertedValue);
                    //重新封装属性的值
                    deepCopy.add(pv);
                }
                else {
                    resolveNecessary = true;
                    deepCopy.add(new PropertyValue(pv, convertedValue));
                }
            }
        }
        if (mpvs != null && !resolveNecessary) {
            //标记属性值已经转换过
            mpvs.setConverted();
        }

        // Set our (possibly massaged) deep copy.
        //进行属性依赖注入
        try {
            bw.setPropertyValues(new MutablePropertyValues(deepCopy));
        }
        catch (BeansException ex) {
            throw new BeanCreationException(
                    mbd.getResourceDescription(), beanName, "Error setting property val
        }
    }
```

分析上述代码，我们可以看出，对属性的注入过程分以下两种情况：
1）、属性值类型不需要强制转换时，不需要解析属性值，直接准备进行依赖注入。
2）、属性值需要进行类型强制转换时，如对其他对象的引用等，首先需要解析属性值，然后对解析后的属性值进行依赖注入。

对属性值的解析是在BeanDefinitionValueResolver类中的resolveValueIfNecessary()方法中进行的，对属性值的依赖注入是通过bw.setPropertyValues()方法实现的，在分析属性值的依赖注入之前，我们先分析一下对属性值的解析过程。

1.7. 解析属性注入规则

当容器在对属性进行依赖注入时，如果发现属性值需要进行类型转换，如属性值是容器中另一个Bean实例对象的引用，则容器首先需要根据属性值解析出所引用的对象，然后才能将该引用对象注入到目标实例对象的属性上去，对属性进行解析的由 resolveValuelfNecessary()方法实现，其源码如下：

```
    //解析属性值，对注入类型进行转换
    @Nullable
    public Object resolveValueIfNecessary(Object argName, @Nullable Object value) {
        // We must check each value to see whether it requires a runtime reference
        // to another bean to be resolved.
        //对引用类型的属性进行解析
        if (value instanceof RuntimeBeanReference) {
            RuntimeBeanReference ref = (RuntimeBeanReference) value;
            //调用引用类型属性的解析方法
            return resolveReference(argName, ref);
        }
        //对属性值是引用容器中另一个Bean名称的解析
        else if (value instanceof RuntimeBeanNameReference) {
            String refName = ((RuntimeBeanNameReference) value).getBeanName();
            refName = String.valueOf(doEvaluate(refName));
            //从容器中获取指定名称的Bean
            if (!this.beanFactory.containsBean(refName)) {
                throw new BeanDefinitionStoreException(
                        "Invalid bean name '" + refName + "' in bean reference for " +
            }
            return refName;
        }
        //对Bean类型属性的解析，主要是Bean中的内部类
        else if (value instanceof BeanDefinitionHolder) {
            // Resolve BeanDefinitionHolder: contains BeanDefinition with name and alia
            BeanDefinitionHolder bdHolder = (BeanDefinitionHolder) value;
            return resolveInnerBean(argName, bdHolder.getBeanName(), bdHolder.getBeanDe
        }
        else if (value instanceof BeanDefinition) {
            // Resolve plain BeanDefinition, without contained name: use dummy name.
            BeanDefinition bd = (BeanDefinition) value;
            String innerBeanName = "(inner bean)" + BeanFactoryUtils.GENERATED_BEAN_NAM
                    ObjectUtils.getIdentityHexString(bd);
            return resolveInnerBean(argName, innerBeanName, bd);
        }
```

```
        //对集合数组类型的属性解析
        else if (value instanceof ManagedArray) {
            // May need to resolve contained runtime references.
            ManagedArray array = (ManagedArray) value;
            //获取数组的类型
            Class<?> elementType = array.resolvedElementType;
            if (elementType == null) {
                //获取数组元素的类型
                String elementTypeName = array.getElementTypeName();
                if (StringUtils.hasText(elementTypeName)) {
                    try {
                        //使用反射机制创建指定类型的对象
                        elementType = ClassUtils.forName(elementTypeName, this.beanFact
                        array.resolvedElementType = elementType;
                    }
                    catch (Throwable ex) {
                        // Improve the message by showing the context.
                        throw new BeanCreationException(
                                this.beanDefinition.getResourceDescription(), this.bear
                                "Error resolving array type for " + argName, ex);
                    }
                }
                //没有获取到数组的类型，也没有获取到数组元素的类型
                //则直接设置数组的类型为Object
                else {
                    elementType = Object.class;
                }
            }
            //创建指定类型的数组
            return resolveManagedArray(argName, (List<?>) value, elementType);
        }
        //解析list类型的属性值
        else if (value instanceof ManagedList) {
            // May need to resolve contained runtime references.
            return resolveManagedList(argName, (List<?>) value);
        }
        //解析set类型的属性值
        else if (value instanceof ManagedSet) {
            // May need to resolve contained runtime references.
            return resolveManagedSet(argName, (Set<?>) value);
        }
        //解析map类型的属性值
        else if (value instanceof ManagedMap) {
```

```
            // May need to resolve contained runtime references.
            return resolveManagedMap(argName, (Map<?, ?>) value);
        }
        //解析props类型的属性值，props其实就是key和value均为字符串的map
        else if (value instanceof ManagedProperties) {
            Properties original = (Properties) value;
            //创建一个拷贝，用于作为解析后的返回值
            Properties copy = new Properties();
            original.forEach((propKey, propValue) -> {
                if (propKey instanceof TypedStringValue) {
                    propKey = evaluate((TypedStringValue) propKey);
                }
                if (propValue instanceof TypedStringValue) {
                    propValue = evaluate((TypedStringValue) propValue);
                }
                if (propKey == null || propValue == null) {
                    throw new BeanCreationException(
                            this.beanDefinition.getResourceDescription(), this.beanName
                            "Error converting Properties key/value pair for " + argName
                }
                copy.put(propKey, propValue);
            });
            return copy;
        }
        //解析字符串类型的属性值
        else if (value instanceof TypedStringValue) {
            // Convert value to target type here.
            TypedStringValue typedStringValue = (TypedStringValue) value;
            Object valueObject = evaluate(typedStringValue);
            try {
                //获取属性的目标类型
                Class<?> resolvedTargetType = resolveTargetType(typedStringValue);
                if (resolvedTargetType != null) {
                    //对目标类型的属性进行解析，递归调用
                    return this.typeConverter.convertIfNecessary(valueObject, resolvedT
                }
                //没有获取到属性的目标对象，则按Object类型返回
                else {
                    return valueObject;
                }
            }
            catch (Throwable ex) {
                // Improve the message by showing the context.
```

```
                throw new BeanCreationException(
                        this.beanDefinition.getResourceDescription(), this.beanName,
                        "Error converting typed String value for " + argName, ex);
            }
        }
        else if (value instanceof NullBean) {
            return null;
        }
        else {
            return evaluate(value);
        }
    }

    //解析引用类型的属性值
    @Nullable
    private Object resolveReference(Object argName, RuntimeBeanReference ref) {
        try {
            Object bean;
            //获取引用的Bean名称
            String refName = ref.getBeanName();
            refName = String.valueOf(doEvaluate(refName));
            //如果引用的对象在父类容器中，则从父类容器中获取指定的引用对象
            if (ref.isToParent()) {
                if (this.beanFactory.getParentBeanFactory() == null) {
                    throw new BeanCreationException(
                            this.beanDefinition.getResourceDescription(), this.beanName
                            "Can't resolve reference to bean '" + refName +
                            "' in parent factory: no parent factory available");
                }
                bean = this.beanFactory.getParentBeanFactory().getBean(refName);
            }
            //从当前的容器中获取指定的引用Bean对象，如果指定的Bean没有被实例化
            //则会递归触发引用Bean的初始化和依赖注入
            else {
                bean = this.beanFactory.getBean(refName);
                //将当前实例化对象的依赖引用对象
                this.beanFactory.registerDependentBean(refName, this.beanName);
            }
            if (bean instanceof NullBean) {
                bean = null;
            }
            return bean;
        }
```

```
        catch (BeansException ex) {
            throw new BeanCreationException(
                    this.beanDefinition.getResourceDescription(), this.beanName,
                    "Cannot resolve reference to bean '" + ref.getBeanName() + "' while
        }
    }

    /**
     * For each element in the managed array, resolve reference if necessary.
     */
    //解析array类型的属性
    private Object resolveManagedArray(Object argName, List<?> ml, Class<?> elementType
        //创建一个指定类型的数组，用于存放和返回解析后的数组
        Object resolved = Array.newInstance(elementType, ml.size());
        for (int i = 0; i < ml.size(); i++) {
            //递归解析array的每一个元素，并将解析后的值设置到resolved数组中，索引为i
            Array.set(resolved, i,
                    resolveValueIfNecessary(new KeyedArgName(argName, i), ml.get(i)));
        }
        return resolved;
    }
```

通过上面的代码分析，我们明白了Spring是如何将引用类型，内部类以及集合类型等属性进行解析的，属性值解析完成后就可以进行依赖注入了，依赖注入的过程就是Bean对象实例设置到它所依赖的Bean对象属性上去。而真正的依赖注入是通过bw.setPropertyValues()方法实现的，该方法也使用了委托模式，在BeanWrapper接口中至少定义了方法声明，依赖注入的具体实现交由其实现类BeanWrapperImpl 来完成，下面我们就分析BeanWrapperImpl中依赖注入相关的源码。

1.8. 注入赋值

BeanWrapperImpl 类主要是对容器中完成初始化的Bean实例对象进行属性的依赖注入，即把Bean对象设置到它所依赖的另一个Bean的属性中去。然而，BeanWrapperImpl中的注入方法实际上由AbstractNestablePropertyAccessor来实现的，其相关源码如下：

```
    //实现属性依赖注入功能
    protected void setPropertyValue(PropertyTokenHolder tokens, PropertyValue pv) throw
        if (tokens.keys != null) {
            processKeyedProperty(tokens, pv);
        }
        else {
            processLocalProperty(tokens, pv);
        }
```

```
    }

    //实现属性依赖注入功能
    @SuppressWarnings("unchecked")
    private void processKeyedProperty(PropertyTokenHolder tokens, PropertyValue pv) {
        //调用属性的getter方法，获取属性的值
        Object propValue = getPropertyHoldingValue(tokens);
        PropertyHandler ph = getLocalPropertyHandler(tokens.actualName);
        if (ph == null) {
            throw new InvalidPropertyException(
                    getRootClass(), this.nestedPath + tokens.actualName, "No property h
        }
        Assert.state(tokens.keys != null, "No token keys");
        String lastKey = tokens.keys[tokens.keys.length - 1];

        //注入array类型的属性值
        if (propValue.getClass().isArray()) {
            Class<?> requiredType = propValue.getClass().getComponentType();
            int arrayIndex = Integer.parseInt(lastKey);
            Object oldValue = null;
            try {
                if (isExtractOldValueForEditor() && arrayIndex < Array.getLength(propVa
                    oldValue = Array.get(propValue, arrayIndex);
                }
                Object convertedValue = convertIfNecessary(tokens.canonicalName, oldVal
                        requiredType, ph.nested(tokens.keys.length));
                //获取集合类型属性的长度
                int length = Array.getLength(propValue);
                if (arrayIndex >= length && arrayIndex < this.autoGrowCollectionLimit)
                    Class<?> componentType = propValue.getClass().getComponentType();
                    Object newArray = Array.newInstance(componentType, arrayIndex + 1);
                    System.arraycopy(propValue, 0, newArray, 0, length);
                    setPropertyValue(tokens.actualName, newArray);
                    //调用属性的getter方法，获取属性的值
                    propValue = getPropertyValue(tokens.actualName);
                }
                //将属性的值赋值给数组中的元素
                Array.set(propValue, arrayIndex, convertedValue);
            }
            catch (IndexOutOfBoundsException ex) {
                throw new InvalidPropertyException(getRootClass(), this.nestedPath + to
                        "Invalid array index in property path '" + tokens.canonicalName
            }
```

```
52          }
53
54          //注入list类型的属性值
55          else if (propValue instanceof List) {
56              //获取list集合的类型
57              Class<?> requiredType = ph.getCollectionType(tokens.keys.length);
58              List<Object> list = (List<Object>) propValue;
59              //获取list集合的size
60              int index = Integer.parseInt(lastKey);
61              Object oldValue = null;
62              if (isExtractOldValueForEditor() && index < list.size()) {
63                  oldValue = list.get(index);
64              }
65              //获取list解析后的属性值
66              Object convertedValue = convertIfNecessary(tokens.canonicalName, oldValue,
67                      requiredType, ph.nested(tokens.keys.length));
68              int size = list.size();
69              //如果list的长度大于属性值的长度，则多余的元素赋值为null
70              if (index >= size && index < this.autoGrowCollectionLimit) {
71                  for (int i = size; i < index; i++) {
72                      try {
73                          list.add(null);
74                      }
75                      catch (NullPointerException ex) {
76                          throw new InvalidPropertyException(getRootClass(), this.nestedP
77                                  "Cannot set element with index " + index + " in List of
78                                  size + ", accessed using property path '" + tokens.canc
79                                  "': List does not support filling up gaps with null ele
80                      }
81                  }
82                  list.add(convertedValue);
83              }
84              else {
85                  try {
86                      //将值添加到list中
87                      list.set(index, convertedValue);
88                  }
89                  catch (IndexOutOfBoundsException ex) {
90                      throw new InvalidPropertyException(getRootClass(), this.nestedPath
91                              "Invalid list index in property path '" + tokens.canonicalN
92                  }
93              }
94          }
```

```
95
96          //注入map类型的属性值
97          else if (propValue instanceof Map) {
98              //获取map集合key的类型
99              Class<?> mapKeyType = ph.getMapKeyType(tokens.keys.length);
100             //获取map集合value的类型
101             Class<?> mapValueType = ph.getMapValueType(tokens.keys.length);
102             Map<Object, Object> map = (Map<Object, Object>) propValue;
103             // IMPORTANT: Do not pass full property name in here - property editors
104             // must not kick in for map keys but rather only for map values.
105             TypeDescriptor typeDescriptor = TypeDescriptor.valueOf(mapKeyType);
106             //解析map类型属性key值
107             Object convertedMapKey = convertIfNecessary(null, null, lastKey, mapKeyType
108             Object oldValue = null;
109             if (isExtractOldValueForEditor()) {
110                 oldValue = map.get(convertedMapKey);
111             }
112             // Pass full property name and old value in here, since we want full
113             // conversion ability for map values.
114             //解析map类型属性value值
115             Object convertedMapValue = convertIfNecessary(tokens.canonicalName, oldValu
116                     mapValueType, ph.nested(tokens.keys.length));
117             //将解析后的key和value值赋值给map集合属性
118             map.put(convertedMapKey, convertedMapValue);
119         }
120
121         else {
122             throw new InvalidPropertyException(getRootClass(), this.nestedPath + tokens
123                     "Property referenced in indexed property path '" + tokens.canonical
124                     "' is neither an array nor a List nor a Map; returned value was ["
125         }
126     }
127
128     private Object getPropertyHoldingValue(PropertyTokenHolder tokens) {
129         // Apply indexes and map keys: fetch value for all keys but the last one.
130         Assert.state(tokens.keys != null, "No token keys");
131         PropertyTokenHolder getterTokens = new PropertyTokenHolder(tokens.actualName);
132         getterTokens.canonicalName = tokens.canonicalName;
133         getterTokens.keys = new String[tokens.keys.length - 1];
134         System.arraycopy(tokens.keys, 0, getterTokens.keys, 0, tokens.keys.length - 1);
135
136         Object propValue;
137         try {
```

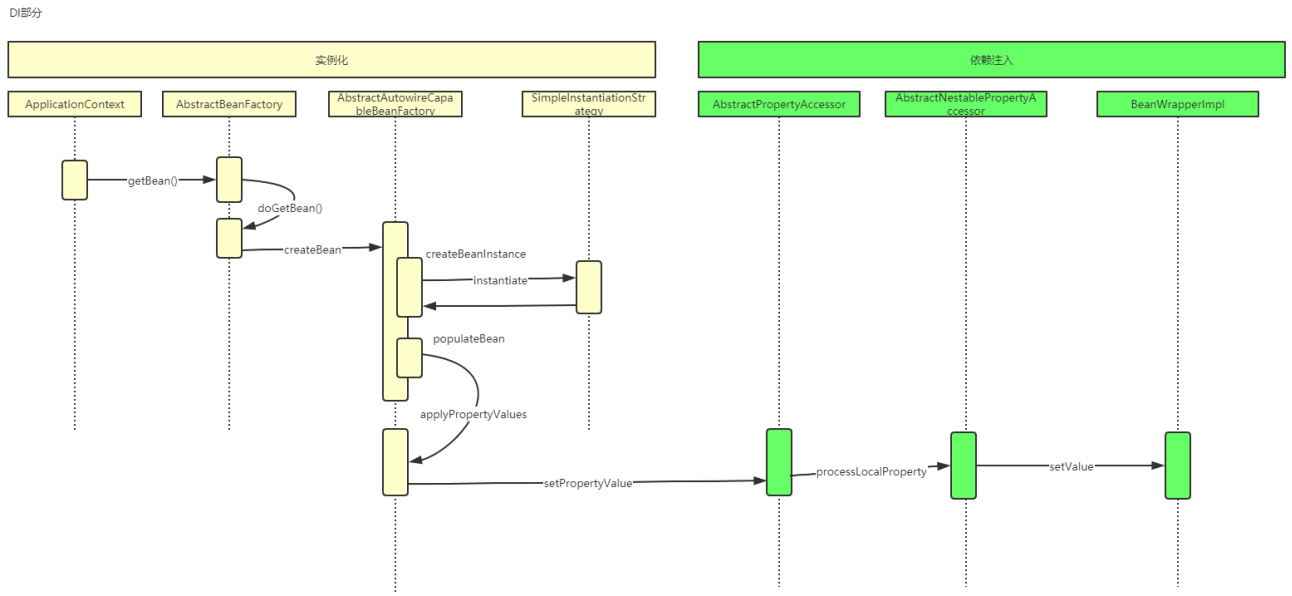
```
            //获取属性值
            propValue = getPropertyValue(getterTokens);
        }
        catch (NotReadablePropertyException ex) {
            throw new NotWritablePropertyException(getRootClass(), this.nestedPath + to
                    "Cannot access indexed value in property referenced " +
                    "in indexed property path '" + tokens.canonicalName + "'", ex);
        }

        if (propValue == null) {
            // null map value case
            if (isAutoGrowNestedPaths()) {
                int lastKeyIndex = tokens.canonicalName.lastIndexOf('[');
                getterTokens.canonicalName = tokens.canonicalName.substring(0, lastKeyI
                propValue = setDefaultValue(getterTokens);
            }
            else {
                throw new NullValueInNestedPathException(getRootClass(), this.nestedPat
                        "Cannot access indexed value in property referenced " +
                        "in indexed property path '" + tokens.canonicalName + "': retur
            }
        }
        return propValue;
    }
```

通过对上面注入依赖代码的分析，我们已经明白了Spring IoC 容器是如何将属性的值注入到Bean实例对象中去的：

1）、对于集合类型的属性，将其属性值解析为目标类型的集合后直接赋值给属性。

2）、对于非集合类型的属性，大量使用了JDK的反射机制，通过属性的 getter()方法获取指定属性注入以前的值，同时调用属性的setter()方法为属性设置注入后的值。看到这里相信很多人都明白了Spring的setter（ ）注入原理。



至此Spring IoC容器对Bean定义资源文件的定位，载入、解析和依赖注入已经全部分析完毕，现在Spring IoC 容器中管理了一系列靠依赖关系联系起来的Bean，程序不需要应用自己手动创建所需的对象，Spring IoC容器会在我们使用的时候自动为我们创建，并且为我们注入好相关的依赖，这就是Spring 核心功能的控制反转和依赖注入的相关功能。

2. IoC容器中那些鲜为人知的细节

通过前面章节中对Spring IoC容器的源码分析，我们已经基本上了解了Spring IoC容器对Bean定义资源的定位、载入和注册过程，同时也清楚了当用户通过 getBean()方法向IoC容器获取被管理的Bean时，IoC容器对Bean进行的初始化和依赖注入过程，这些是Spring IoC容器的基本功能特性。

Spring IoC 容器还有一些高级特性，如使用lazy-init 属性对Bean 预初始化、FactoryBean 产生或者修饰Bean 对象的生成、IoC容器初始化Bean过程中使用BeanPostProcessor 后置处理器对Bean声明周期事件管理等。

2.1. 关于延时加载

通过前面我们对IoC容器的实现和工作原理分析我们已经知道IoC容器的初始化过程就是对Bean定义资源的定位、载入和注册，此时容器对Bean的依赖注入并没有发生，依赖注入主要是在应用程序第一次向容器索取Bean时，通过 getBean()方法的调用完成。
当Bean定义资源的 `<Bean>` 元素中配置了lazy-init=false属性时，容器将会在初始化的时候对所配置的Bean 进行预实例化，Bean的依赖注入在容器初始化的时候就已经完成。这样，当应用程序第一次向容器索取被管理的Bean时，就不用再初始化和对Bean进行依赖注入了，直接从容器中获取已经完成依赖注入的现成Bean，可以提高应用第一次向容器获取Bean的性能。

2.1.1. refresh()方法

先从IoC容器的初始化过程开始，我们知道IoC容器读入已经定位的Bean定义资源是从refresh()方法开始的，我们首先从AbstractApplicationContext类的refresh()方法入手分析，源码如下：

```
```

```java
    @Override
    public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
            // Prepare this context for refreshing.
            //1、调用容器准备刷新的方法，获取容器的当时时间，同时给容器设置同步标识
            prepareRefresh();

            // Tell the subclass to refresh the internal bean factory.
            //2、告诉子类启动refreshBeanFactory()方法，Bean定义资源文件的载入从
            //子类的refreshBeanFactory()方法启动
            ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

            // Prepare the bean factory for use in this context.
            //3、为BeanFactory配置容器特性，例如类加载器、事件处理器等
            prepareBeanFactory(beanFactory);

            try {
                // Allows post-processing of the bean factory in context subclasses.
                //4、为容器的某些子类指定特殊的BeanPost事件处理器
                postProcessBeanFactory(beanFactory);

                // Invoke factory processors registered as beans in the context.
                //5、调用所有注册的BeanFactoryPostProcessor的Bean
                invokeBeanFactoryPostProcessors(beanFactory);

                // Register bean processors that intercept bean creation.
                //6、为BeanFactory注册BeanPost事件处理器.
                //BeanPostProcessor是Bean后置处理器，用于监听容器触发的事件
                registerBeanPostProcessors(beanFactory);

                // Initialize message source for this context.
                //7、初始化信息源，和国际化相关.
                initMessageSource();

                // Initialize event multicaster for this context.
                //8、初始化容器事件传播器.
                initApplicationEventMulticaster();

                // Initialize other special beans in specific context subclasses.
                //9、调用子类的某些特殊Bean初始化方法
                onRefresh();

                // Check for listener beans and register them.
```

```
                //10、为事件传播器注册事件监听器.
                registerListeners();

                // Instantiate all remaining (non-lazy-init) singletons.
                //11、初始化所有剩余的单例Bean
                finishBeanFactoryInitialization(beanFactory);

                // Last step: publish corresponding event.
                //12、初始化容器的生命周期事件处理器，并发布容器的生命周期事件
                finishRefresh();
            }

            catch (BeansException ex) {
                if (logger.isWarnEnabled()) {
                    logger.warn("Exception encountered during context initialization -
                            "cancelling refresh attempt: " + ex);
                }

                // Destroy already created singletons to avoid dangling resources.
                //13、销毁已创建的Bean
                destroyBeans();

                // Reset 'active' flag.
                //14、取消refresh操作，重置容器的同步标识。
                cancelRefresh(ex);

                // Propagate exception to caller.
                throw ex;
            }

            finally {
                // Reset common introspection caches in Spring's core, since we
                // might not ever need metadata for singleton beans anymore...
                //15、重设公共缓存
                resetCommonCaches();
            }
        }
    }
```

在refresh()方法中ConfigurableListableBeanFactorybeanFactory=obtainFreshBeanFactory()；启动了Bean 定义资源的载入、注册过程，而finishBeanFactoryInitialization 方法是对注册后的

Bean定义中的预实例化（lazy-init=false，Spring 默认就是预实例化，即为true）的Bean进行处理的地方。

2.1.2. finishBeanFactoryInitialization 处理预实例化Bean

当Bean定义资源被载入IoC容器之后，容器将Bean 定义资源解析为容器内部的数据结构BeanDefinition 注册到容器中AbstractApplicationContext类中的finishBeanFactoryInitialization()方法对配置了预实例化属性的Bean进行预初始化过程，源码如下：

```
    //对配置了lazy-init属性的Bean进行预实例化处理
    protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory bean
        // Initialize conversion service for this context.
        //这是Spring3以后新加的代码，为容器指定一个转换服务(ConversionService)
        //在对某些Bean属性进行转换时使用
        if (beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&
                beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME, ConversionService
            beanFactory.setConversionService(
                    beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME, ConversionService
        }

        // Register a default embedded value resolver if no bean post-processor
        // (such as a PropertyPlaceholderConfigurer bean) registered any before:
        // at this point, primarily for resolution in annotation attribute values.
        if (!beanFactory.hasEmbeddedValueResolver()) {
            beanFactory.addEmbeddedValueResolver(strVal -> getEnvironment().resolvePlac
        }

        // Initialize LoadTimeWeaverAware beans early to allow for registering their tr
        String[] weaverAwareNames = beanFactory.getBeanNamesForType(LoadTimeWeaverAware
        for (String weaverAwareName : weaverAwareNames) {
            getBean(weaverAwareName);
        }

        // Stop using the temporary ClassLoader for type matching.
        //为了类型匹配，停止使用临时的类加载器
        beanFactory.setTempClassLoader(null);

        // Allow for caching all bean definition metadata, not expecting further change
        //缓存容器中所有注册的BeanDefinition元数据，以防被修改
        beanFactory.freezeConfiguration();

```

```
        // Instantiate all remaining (non-lazy-init) singletons.
        //对配置了lazy-init属性的单态模式Bean进行预实例化处理
        beanFactory.preInstantiateSingletons();
    }
```

ConfigurablelistableBeanFactory是一个接口，其preInstantiateSingletons()方法由其子类DefaultListableBeanFactory提供。

2.1.3. DefaultListableBeanFactory 对配置lazy-init 属性单态Bean的预实例化

```
    //对配置lazy-init属性单态Bean的预实例化
    @Override
    public void preInstantiateSingletons() throws BeansException {
        if (this.logger.isDebugEnabled()) {
            this.logger.debug("Pre-instantiating singletons in " + this);
        }

        // Iterate over a copy to allow for init methods which in turn register new bea
        // While this may not be part of the regular factory bootstrap, it does otherwi
        List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);

        // Trigger initialization of all non-lazy singleton beans...
        for (String beanName : beanNames) {
            //获取指定名称的Bean定义
            RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
            //Bean不是抽象的，是单态模式的，且lazy-init属性配置为false
            if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
                //如果指定名称的bean是创建容器的Bean
                if (isFactoryBean(beanName)) {
                    //FACTORY_BEAN_PREFIX=”&”，当Bean名称前面加”&”符号
                    //时，获取的是产生容器对象本身，而不是容器产生的Bean.
                    //调用getBean方法，触发容器对Bean实例化和依赖注入过程
                    final FactoryBean<?> factory = (FactoryBean<?>) getBean(FACTORY_BEA
                    //标识是否需要预实例化
                    boolean isEagerInit;
                    if (System.getSecurityManager() != null && factory instanceof Smart
                        //一个匿名内部类
                        isEagerInit = AccessController.doPrivileged((PrivilegedAction<B
                                ((SmartFactoryBean<?>) factory).isEagerInit(),
                                getAccessControlContext());
```

```
                    }
                    else {
                        isEagerInit = (factory instanceof SmartFactoryBean &&
                                ((SmartFactoryBean<?>) factory).isEagerInit());
                    }
                    if (isEagerInit) {
                        //调用getBean方法，触发容器对Bean实例化和依赖注入过程
                        getBean(beanName);
                    }
                }
                else {
                    getBean(beanName);
                }
            }
        }
```

通过对lazy-init处理源码的分析，我们可以看出，如果设置了lazy-init属性，则容器在完成Bean定义的注册之后，会通过 getBean 方法，触发对指定Bean的初始化和依赖注入过程，这样当应用第一次向容器索取所需的Bean时，容器不再需要对Bean进行初始化和依赖注入，直接从已经完成实例化和依赖注入的Bean中取一个现成的Bean，这样就提高了第一次获取Bean的性能。

2.2. 关于FactoryBean和BeanFactory

在Spring中，有两个很容易混淆的类：BeanFactory和FactoryBean。

BeanFactory:Bean工厂，是一个工厂（Factory），我们Spring IoC容器的最顶层接口就是这个BeanFactory，它的作用是管理Bean，即实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

FactoryBean：工厂Bean，是一个Bean，作用是产生其他bean实例。通常情况下，这种Bean没有什么特别的要求，仅需要提供一个工厂方法，该方法用来返回其他Bean实例。通常情况下，Bean无须自己实现工厂模式，Spring容器担任工厂角色；但少数情况下，容器中的Bean本身就是工厂，其作用是产生其它Bean实例。

当用户使用容器本身时，可以使用转义字符"&”来得到FactoryBean本身，以区别通过FactoryBean产生的实例对象和FactoryBean对象本身。在BeanFactory中通过如下代码定义了该转义字符：String FACTORY_BEAN_PREFIX="&；如果myJndiObject是一个FactoryBean，则使用&myJndiObject得到的是myJndiObject对象，而不是myJndiObject 产生出来的对象。

2.2.1. FactoryBean源码

```
/*
 * Copyright 2002-2016 the original author or authors.
```

```
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.beans.factory;

import org.springframework.lang.Nullable;

/**
 * Interface to be implemented by objects used within a {@link BeanFactory} which
 * are themselves factories for individual objects. If a bean implements this
 * interface, it is used as a factory for an object to expose, not directly as a
 * bean instance that will be exposed itself.
 *
 * <p><b>NB: A bean that implements this interface cannot be used as a normal bean.</b>
 * A FactoryBean is defined in a bean style, but the object exposed for bean
 * references ({@link #getObject()}) is always the object that it creates.
 *
 * <p>FactoryBeans can support singletons and prototypes, and can either create
 * objects lazily on demand or eagerly on startup. The {@link SmartFactoryBean}
 * interface allows for exposing more fine-grained behavioral metadata.
 *
 * <p>This interface is heavily used within the framework itself, for example for
 * the AOP {@link org.springframework.aop.framework.ProxyFactoryBean} or the
 * {@link org.springframework.jndi.JndiObjectFactoryBean}. It can be used for
 * custom components as well; however, this is only common for infrastructure code.
 *
 * <p><b>{@code FactoryBean} is a programmatic contract. Implementations are not
 * supposed to rely on annotation-driven injection or other reflective facilities.</b>
 * {@link #getObjectType()} {@link #getObject()} invocations may arrive early in
 * the bootstrap process, even ahead of any post-processor setup. If you need access
 * other beans, implement {@link BeanFactoryAware} and obtain them programmatically.
 *
```

```
 * <p>Finally, FactoryBean objects participate in the containing BeanFactory's
 * synchronization of bean creation. There is usually no need for internal
 * synchronization other than for purposes of lazy initialization within the
 * FactoryBean itself (or the like).
 *
 * @author Rod Johnson
 * @author Juergen Hoeller
 * @since 08.03.2003
 * @see org.springframework.beans.factory.BeanFactory
 * @see org.springframework.aop.framework.ProxyFactoryBean
 * @see org.springframework.jndi.JndiObjectFactoryBean
 */
//工厂Bean，用于产生其他对象
public interface FactoryBean<T> {

    //获取容器管理的对象实例
    @Nullable
    T getObject() throws Exception;

    //获取Bean工厂创建的对象的类型
    @Nullable
    Class<?> getObjectType();

    //Bean工厂创建的对象是否是单态模式，如果是单态模式，则整个容器中只有一个实例
    //对象，每次请求都返回同一个实例对象
    default boolean isSingleton() {
        return true;
    }

}
```

2.2.2. AbstractBeanFactory 的 getBean()方法调用FactoryBean

在前面我们分析Spring IoC 容器实例化Bean并进行依赖注入过程的源码时，提到在getBean()方法触发容器实例化Bean的时候会调用AbstractBeanFactory的doGetBean()方法来进行实例化的过程，源码如下：

```
    //真正实现向IOC容器获取Bean的功能，也是触发依赖注入功能的地方
    protected <T> T doGetBean(final String name, @Nullable final Class<T> requiredType,
            @Nullable final Object[] args, boolean typeCheckOnly) throws BeansExceptior

```

```
        //根据指定的名称获取被管理Bean的名称，剥离指定名称中对容器的相关依赖
        //如果指定的是别名，将别名转换为规范的Bean名称
        final String beanName = transformedBeanName(name);
        Object bean;

        // Eagerly check singleton cache for manually registered singletons.
        //先从缓存中取是否已经有被创建过的单态类型的Bean
        //对于单例模式的Bean整个IOC容器中只创建一次，不需要重复创建
        Object sharedInstance = getSingleton(beanName);
        //IOC容器创建单例模式Bean实例对象
        if (sharedInstance != null && args == null) {
            if (logger.isDebugEnabled()) {
                //如果指定名称的Bean在容器中已有单例模式的Bean被创建
                //直接返回已经创建的Bean
                if (isSingletonCurrentlyInCreation(beanName)) {
                    logger.debug("Returning eagerly cached instance of singleton bean '
                            "' that is not fully initialized yet - a consequence of a c
                }
                else {
                    logger.debug("Returning cached instance of singleton bean '" + bear
                }
            }
            //获取给定Bean的实例对象，主要是完成FactoryBean的相关处理
            //注意：BeanFactory是管理容器中Bean的工厂，而FactoryBean是
            //创建创建对象的工厂Bean，两者之间有区别
            bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
        }

        else {
            // Fail if we're already creating this bean instance:
            // We're assumably within a circular reference.
            //缓存没有正在创建的单例模式Bean
            //缓存中已经有已经创建的原型模式Bean
            //但是由于循环引用的问题导致实例化对象失败
            if (isPrototypeCurrentlyInCreation(beanName)) {
                throw new BeanCurrentlyInCreationException(beanName);
            }

            // Check if bean definition exists in this factory.
            //对IOC容器中是否存在指定名称的BeanDefinition进行检查，首先检查是否
            //能在当前的BeanFactory中获取的所需要的Bean，如果不能则委托当前容器
            //的父级容器去查找，如果还是找不到则沿着容器的继承体系向父级容器查找
            BeanFactory parentBeanFactory = getParentBeanFactory();
```

```
            //当前容器的父级容器存在，且当前容器中不存在指定名称的Bean
            if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
                // Not found -> check parent.
                //解析指定Bean名称的原始名称
                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.
                    //委派父级容器根据指定名称和显式的参数查找
                    return (T) parentBeanFactory.getBean(nameToLookup, args);
                }
                else {
                    // No args -> delegate to standard getBean method.
                    //委派父级容器根据指定名称和类型查找
                    return parentBeanFactory.getBean(nameToLookup, requiredType);
                }
            }

            //创建的Bean是否需要进行类型验证，一般不需要
            if (!typeCheckOnly) {
                //向容器标记指定的Bean已经被创建
                markBeanAsCreated(beanName);
            }

            try {
                //根据指定Bean名称获取其父级的Bean定义
                //主要解决Bean继承时子类合并父类公共属性问题
                final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
                checkMergedBeanDefinition(mbd, beanName, args);

                // Guarantee initialization of beans that the current bean depends on.
                //获取当前Bean所有依赖Bean的名称
                String[] dependsOn = mbd.getDependsOn();
                //如果当前Bean有依赖Bean
                if (dependsOn != null) {
                    for (String dep : dependsOn) {
                        if (isDependent(beanName, dep)) {
                            throw new BeanCreationException(mbd.getResourceDescription(
                                    "Circular depends-on relationship between '" + bean
                            }
```

```
                        //递归调用getBean方法，获取当前Bean的依赖Bean
                        registerDependentBean(dep, beanName);
                        //把被依赖Bean注册给当前依赖的Bean
                        getBean(dep);
                    }
                }

                // Create bean instance.
                //创建单例模式Bean的实例对象
                if (mbd.isSingleton()) {
                    //这里使用了一个匿名内部类，创建Bean实例对象，并且注册给所依赖的对象
                    sharedInstance = getSingleton(beanName, () -> {
                        try {
                            //创建一个指定Bean实例对象，如果有父级继承，则合并子类和父类
                            return createBean(beanName, mbd, args);
                        }
                        catch (BeansException ex) {
                            // Explicitly remove instance from singleton cache: It migh
                            // eagerly by the creation process, to allow for circular r
                            // Also remove any beans that received a temporary referenc
                            //显式地从容器单例模式Bean缓存中清除实例对象
                            destroySingleton(beanName);
                            throw ex;
                        }
                    });
                    //获取给定Bean的实例对象
                    bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd
                }

                //IOC容器创建原型模式Bean实例对象
                else if (mbd.isPrototype()) {
                    // It's a prototype -> create a new instance.
                    //原型模式(Prototype)是每次都会创建一个新的对象
                    Object prototypeInstance = null;
                    try {
                        //回调beforePrototypeCreation方法，默认的功能是注册当前创建的原
                        beforePrototypeCreation(beanName);
                        //创建指定Bean对象实例
                        prototypeInstance = createBean(beanName, mbd, args);
                    }
                    finally {
                        //回调afterPrototypeCreation方法，默认的功能告诉IOC容器指定Bean
                        afterPrototypeCreation(beanName);
```

```
                    }
                    //获取给定Bean的实例对象
                    bean = getObjectForBeanInstance(prototypeInstance, name, beanName,
                }

                //要创建的Bean既不是单例模式，也不是原型模式，则根据Bean定义资源中
                //配置的生命周期范围，选择实例化Bean的合适方法，这种在Web应用程序中
                //比较常用，如：request、session、application等生命周期
                else {
                    String scopeName = mbd.getScope();
                    final Scope scope = this.scopes.get(scopeName);
                    //Bean定义资源中没有配置生命周期范围，则Bean定义不合法
                    if (scope == null) {
                        throw new IllegalStateException("No Scope registered for scope
                    }
                    try {
                        //这里又使用了一个匿名内部类，获取一个指定生命周期范围的实例
                        Object scopedInstance = scope.get(beanName, () -> {
                            beforePrototypeCreation(beanName);
                            try {
                                return createBean(beanName, mbd, args);
                            }
                            finally {
                                afterPrototypeCreation(beanName);
                            }
                        });
                        //获取给定Bean的实例对象
                        bean = getObjectForBeanInstance(scopedInstance, name, beanName,
                    }
                    catch (IllegalStateException ex) {
                        throw new BeanCreationException(beanName,
                                "Scope '" + scopeName + "' is not active for the currer
                                "defining a scoped proxy for this bean if you intend to
                                ex);
                    }
                }
            }
            catch (BeansException ex) {
                cleanupAfterBeanCreationFailure(beanName);
                throw ex;
            }
        }

```

```
        // Check if required type matches the type of the actual bean instance.
        //对创建的Bean实例对象进行类型检查
        if (requiredType != null && !requiredType.isInstance(bean)) {
            try {
                T convertedBean = getTypeConverter().convertIfNecessary(bean, requiredT
                if (convertedBean == null) {
                    throw new BeanNotOfRequiredTypeException(name, requiredType, bean.g
                }
                return convertedBean;
            }
            catch (TypeMismatchException ex) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Failed to convert bean '" + name + "' to required typ
                            ClassUtils.getQualifiedName(requiredType) + "'", ex);
                }
                throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getCl
            }
        }
        return (T) bean;
    }

    //获取给定Bean的实例对象，主要是完成FactoryBean的相关处理
    protected Object getObjectForBeanInstance(
            Object beanInstance, String name, String beanName, @Nullable RootBeanDefini

        // Don't let calling code try to dereference the factory if the bean isn't a fa
        //容器已经得到了Bean实例对象，这个实例对象可能是一个普通的Bean，
        //也可能是一个工厂Bean，如果是一个工厂Bean，则使用它创建一个Bean实例对象，
        //如果调用本身就想获得一个容器的引用，则指定返回这个工厂Bean实例对象
        //如果指定的名称是容器的解引用(dereference，即是对象本身而非内存地址)，
        //且Bean实例也不是创建Bean实例对象的工厂Bean
        if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance instanceof Fa
            throw new BeanIsNotAFactoryException(transformedBeanName(name), beanInstanc
        }

        // Now we have the bean instance, which may be a normal bean or a FactoryBean.
        // If it's a FactoryBean, we use it to create a bean instance, unless the
        // caller actually wants a reference to the factory.
        //如果Bean实例不是工厂Bean，或者指定名称是容器的解引用，
        //调用者向获取对容器的引用，则直接返回当前的Bean实例
        if (!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.isFactoryDerefer
            return beanInstance;
        }
```

```
220
221         //处理指定名称不是容器的解引用，或者根据名称获取的Bean实例对象是一个工厂Bean
222         //使用工厂Bean创建一个Bean的实例对象
223         Object object = null;
224         if (mbd == null) {
225             //从Bean工厂缓存中获取给定名称的Bean实例对象
226             object = getCachedObjectForFactoryBean(beanName);
227         }
228         //让Bean工厂生产给定名称的Bean对象实例
229         if (object == null) {
230             // Return bean instance from factory.
231             FactoryBean<?> factory = (FactoryBean<?>) beanInstance;
232             // Caches object obtained from FactoryBean if it is a singleton.
233             //如果从Bean工厂生产的Bean是单态模式的，则缓存
234             if (mbd == null && containsBeanDefinition(beanName)) {
235                 //从容器中获取指定名称的Bean定义，如果继承基类，则合并基类相关属性
236                 mbd = getMergedLocalBeanDefinition(beanName);
237             }
238             //如果从容器得到Bean定义信息，并且Bean定义信息不是虚构的，
239             //则让工厂Bean生产Bean实例对象
240             boolean synthetic = (mbd != null && mbd.isSynthetic());
241             //调用FactoryBeanRegistrySupport类的getObjectFromFactoryBean方法，
242             //实现工厂Bean生产Bean对象实例的过程
243             object = getObjectFromFactoryBean(factory, beanName, !synthetic);
244         }
245         return object;
246     }
```

在上面获取给定Bean的实例对象的getObjectForBeanInstance()方法中，会调用FactoryBeanRegistrySupport 类的 getObjectFromFactoryBean()方法，该方法实现了Bean工厂生产Bean实例对象。

Dereference（解引用）：一个在C/C++中应用比较多的术语，在C++中，"*”是解引用符号，而”
&”是引用符号，解引用是指变量指向的是所引用对象的本身数据，而不是引用对象的内存地址。

2.2.3. AbstractBeanFactory 生产Bean 实例对象

AbstractBeanFactory 类中生产Bean实例对象的主要源码如下：

```
1     protected Object getObjectFromFactoryBean(FactoryBean<?> factory, String beanName,
2         //Bean工厂是单态模式，并且Bean工厂缓存中存在指定名称的Bean实例对象
```

```
        if (factory.isSingleton() && containsSingleton(beanName)) {
            //多线程同步，以防止数据不一致
            synchronized (getSingletonMutex()) {
                //直接从Bean工厂缓存中获取指定名称的Bean实例对象
                Object object = this.factoryBeanObjectCache.get(beanName);
                //Bean工厂缓存中没有指定名称的实例对象，则生产该实例对象
                if (object == null) {
                    //调用Bean工厂的getObject方法生产指定Bean的实例对象
                    object = doGetObjectFromFactoryBean(factory, beanName);
                    // Only post-process and store if not put there already during getO
                    // (e.g. because of circular reference processing triggered by cust
                    Object alreadyThere = this.factoryBeanObjectCache.get(beanName);
                    if (alreadyThere != null) {
                        object = alreadyThere;
                    }
                    else {
                        if (shouldPostProcess) {
                            try {
                                object = postProcessObjectFromFactoryBean(object, beanN
                            }
                            catch (Throwable ex) {
                                throw new BeanCreationException(beanName,
                                        "Post-processing of FactoryBean's singleton obj
                            }
                        }
                        //将生产的实例对象添加到Bean工厂缓存中
                        this.factoryBeanObjectCache.put(beanName, object);
                    }
                }
                return object;
            }
        }
        //调用Bean工厂的getObject方法生产指定Bean的实例对象
        else {
            Object object = doGetObjectFromFactoryBean(factory, beanName);
            if (shouldPostProcess) {
                try {
                    object = postProcessObjectFromFactoryBean(object, beanName);
                }
                catch (Throwable ex) {
                    throw new BeanCreationException(beanName, "Post-processing of Facto
                }
            }
```

```
            return object;
        }
    }

    /**
     * Obtain an object to expose from the given FactoryBean.
     * @param factory the FactoryBean instance
     * @param beanName the name of the bean
     * @return the object obtained from the FactoryBean
     * @throws BeanCreationException if FactoryBean object creation failed
     * @see org.springframework.beans.factory.FactoryBean#getObject()
     */
    //调用Bean工厂的getObject方法生产指定Bean的实例对象
    private Object doGetObjectFromFactoryBean(final FactoryBean<?> factory, final Strin
            throws BeanCreationException {

        Object object;
        try {
            if (System.getSecurityManager() != null) {
                AccessControlContext acc = getAccessControlContext();
                try {
                    //实现PrivilegedExceptionAction接口的匿名内置类
                    //根据JVM检查权限，然后决定BeanFactory创建实例对象
                    object = AccessController.doPrivileged((PrivilegedExceptionAction<O
                            factory.getObject(), acc);
                }
                catch (PrivilegedActionException pae) {
                    throw pae.getException();
                }
            }
            else {
                //调用BeanFactory接口实现类的创建对象方法
                object = factory.getObject();
            }
        }
        catch (FactoryBeanNotInitializedException ex) {
            throw new BeanCurrentlyInCreationException(beanName, ex.toString());
        }
        catch (Throwable ex) {
            throw new BeanCreationException(beanName, "FactoryBean threw exception on o
        }

        // Do not accept a null value for a FactoryBean that's not fully
```

```
        // initialized yet: Many FactoryBeans just return null then.
        //创建出来的实例对象为null，或者因为单态对象正在创建而返回null
        if (object == null) {
            if (isSingletonCurrentlyInCreation(beanName)) {
                throw new BeanCurrentlyInCreationException(
                        beanName, "FactoryBean which is currently in creation returned
            }
            object = new NullBean();
        }
        return object;
    }
```

从上面的源码分析中，我们可以看出，BeanFactory接口调用其实现类的 getObject 方法来实现创建 Bean实例对象的功能。

2.2.4. 工厂Bean的实现类getObject 方法创建Bean 实例对象

FactoryBean的实现类有非常多，比如：Proxy、RMI、JNDI、ServletContextFactoryBean等等，FactoryBean 接口为Spring容器提供了一个很好的封装机制，具体的getObject()有不同的实现类根据不同的实现策略来具体提供，我们分析一个最简单的AnnotationTestFactoryBean的实现源码：

```
public class AnnotationTestBeanFactory implements FactoryBean<FactoryCreatedAnnotationT

    private final FactoryCreatedAnnotationTestBean instance = new FactoryCreatedAnnotat

    public AnnotationTestBeanFactory() {
        this.instance.setName("FACTORY");
    }

    @Override
    public FactoryCreatedAnnotationTestBean getObject() throws Exception {
        return this.instance;
    }

    //AnnotationTestBeanFactory产生Bean实例对象的实现
    @Override
    public Class<? extends IJmxTestBean> getObjectType() {
        return FactoryCreatedAnnotationTestBean.class;
    }

    @Override
    public boolean isSingleton() {
```

```
        return true;
    }

}
```

其他的Proxy，RMl，JNDI等等，都是根据相应的策略提供 getObject()的实现。这里不做一一分析，这已经不是Spring的核心功能，感兴趣的小伙可以再去深入研究。

2.3. 再述autowiring

Spring IoC 容器提供了两种管理Bean依赖关系的方式：

1）、显式管理：通过BeanDefinition的属性值和构造方法实现Bean依赖关系管理。
2）、autowiring:Spring IoC容器的依赖自动装配功能，不需要对Bean属性的依赖关系做显式的声明，只需要在配置好autowiring属性，IoC容器会自动使用反射查找属性的类型和名称，然后基于属性的类型或者名称来自动匹配容器中管理的Bean，从而自动地完成依赖注入。
通过对autowiring 自动装配特性的理解，我们知道容器对Bean的自动装配发生在容器对Bean依赖注入的过程中。在前面对Spring IoC 容器的依赖注入过程源码分析中，我们已经知道了容器对Bean实例对象的属性注入的处理发生在AbstractAutoWireCapableBeanFactory 类中的populateBean()方法中，我们通过程序流程分析 autowiring的实现原理：

2.3.1. AbstractAutoWireCapableBeanFactory对Bean 实例进行属性依赖注入

应用第一次通过 getBean()方法（配置了lazy-init 预实例化属性的除外）向IoC容器索取Bean时，容器创建Bean实例对象，并且对Bean实例对象进行属性依赖注入，
AbstractAutoWireCapableBeanFactory的 populateBean()方法就是实现Bean属性依赖注入的功能，其主要源码如下：

```
    //将Bean属性设置到生成的实例对象上
    protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable Bean
        if (bw == null) {
            if (mbd.hasPropertyValues()) {
                throw new BeanCreationException(
                        mbd.getResourceDescription(), beanName, "Cannot apply property
            }
            else {
                // Skip property population phase for null instance.
                return;
            }
        }

```

```
        // Give any InstantiationAwareBeanPostProcessors the opportunity to modify the
        // state of the bean before properties are set. This can be used, for example,
        // to support styles of field injection.
        boolean continueWithPropertyPopulation = true;

        if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
            for (BeanPostProcessor bp : getBeanPostProcessors()) {
                if (bp instanceof InstantiationAwareBeanPostProcessor) {
                    InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPo
                    if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), bea
                        continueWithPropertyPopulation = false;
                        break;
                    }
                }
            }
        }

        if (!continueWithPropertyPopulation) {
            return;
        }
        //获取容器在解析Bean定义资源时为BeanDefiniton中设置的属性值
        PropertyValues pvs = (mbd.hasPropertyValues() ? mbd.getPropertyValues() : null)

        //对依赖注入处理，首先处理autowiring自动装配的依赖注入
        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
                mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
            MutablePropertyValues newPvs = new MutablePropertyValues(pvs);

            // Add property values based on autowire by name if applicable.
            //根据Bean名称进行autowiring自动装配处理
            if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
                autowireByName(beanName, mbd, bw, newPvs);
            }

            // Add property values based on autowire by type if applicable.
            //根据Bean类型进行autowiring自动装配处理
            if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
                autowireByType(beanName, mbd, bw, newPvs);
            }

            pvs = newPvs;
        }

```

```
        //对非autowiring的属性进行依赖注入处理
        ...
```

2.3.2. Spring IoC 容器根据Bean 名称或者类型进行 autowiring 自动依赖注入

```
    //根据类型对属性进行自动依赖注入
    protected void autowireByType(
            String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropert

        //获取用户定义的类型转换器
        TypeConverter converter = getCustomTypeConverter();
        if (converter == null) {
            converter = bw;
        }

        //存放解析的要注入的属性
        Set<String> autowiredBeanNames = new LinkedHashSet<>(4);
        //对Bean对象中非简单属性(不是简单继承的对象，如8中原始类型，字符
        //URL等都是简单属性)进行处理
        String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
        for (String propertyName : propertyNames) {
            try {
                //获取指定属性名称的属性描述器
                PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);
                // Don't try autowiring by type for type Object: never makes sense,
                // even if it technically is a unsatisfied, non-simple property.
                //不对Object类型的属性进行autowiring自动依赖注入
                if (Object.class != pd.getPropertyType()) {
                    //获取属性的setter方法
                    MethodParameter methodParam = BeanUtils.getWriteMethodParameter(pd)
                    // Do not allow eager init for type matching in case of a prioritiz
                    //检查指定类型是否可以被转换为目标对象的类型
                    boolean eager = !PriorityOrdered.class.isInstance(bw.getWrappedInst
                    //创建一个要被注入的依赖描述
                    DependencyDescriptor desc = new AutowireByTypeDependencyDescriptor(
                    //根据容器的Bean定义解析依赖关系，返回所有要被注入的Bean对象
                    Object autowiredArgument = resolveDependency(desc, beanName, autowi
                    if (autowiredArgument != null) {
                        //为属性赋值所引用的对象
```

```
                    pvs.add(propertyName, autowiredArgument);
                }
                for (String autowiredBeanName : autowiredBeanNames) {
                    //指定名称属性注册依赖Bean名称，进行属性依赖注入
                    registerDependentBean(autowiredBeanName, beanName);
                    if (logger.isDebugEnabled()) {
                        logger.debug("Autowiring by type from bean name '" + beanNa
                                propertyName + "' to bean named '" + autowiredBeanN
                    }
                }
                //释放已自动注入的属性
                autowiredBeanNames.clear();
            }
        }
        catch (BeansException ex) {
            throw new UnsatisfiedDependencyException(mbd.getResourceDescription(),
        }
    }
}
```

通过上面的源码分析，我们可以看出来通过属性名进行自动依赖注入的相对比通过属性类型进行自动依赖注入要稍微简单一些，但是真正实现属性注入的是DefaultSingletonBeanRegistry类的registerDependentBean()方法。

2.3.3. DefaultSingletonBeanRegistry 的 registerDependentBean()方法对属性注入

```
//为指定的Bean注入依赖的Bean
public void registerDependentBean(String beanName, String dependentBeanName) {
    // A quick check for an existing entry upfront, avoiding synchronization...
    //处理Bean名称，将别名转换为规范的Bean名称
    String canonicalName = canonicalName(beanName);
    Set<String> dependentBeans = this.dependentBeanMap.get(canonicalName);
    if (dependentBeans != null && dependentBeans.contains(dependentBeanName)) {
        return;
    }

    // No entry yet -> fully synchronized manipulation of the dependentBeans Set
    //多线程同步，保证容器内数据的一致性
    //先从容器中：bean名称-->全部依赖Bean名称集合找查找给定名称Bean的依赖Bean
    synchronized (this.dependentBeanMap) {
```

```
            //获取给定名称Bean的所有依赖Bean名称
            dependentBeans = this.dependentBeanMap.get(canonicalName);
            if (dependentBeans == null) {
                //为Bean设置依赖Bean信息
                dependentBeans = new LinkedHashSet<>(8);
                this.dependentBeanMap.put(canonicalName, dependentBeans);
            }
            //向容器中：bean名称-->全部依赖Bean名称集合添加Bean的依赖信息
            //即，将Bean所依赖的Bean添加到容器的集合中
            dependentBeans.add(dependentBeanName);
        }
        //从容器中：bean名称-->指定名称Bean的依赖Bean集合找查找给定名称Bean的依赖Bean
        synchronized (this.dependenciesForBeanMap) {
            Set<String> dependenciesForBean = this.dependenciesForBeanMap.get(dependent
            if (dependenciesForBean == null) {
                dependenciesForBean = new LinkedHashSet<>(8);
                this.dependenciesForBeanMap.put(dependentBeanName, dependenciesForBean)
            }
            //向容器中：bean名称-->指定Bean的依赖Bean名称集合添加Bean的依赖信息
            //即，将Bean所依赖的Bean添加到容器的集合中
            dependenciesForBean.add(canonicalName);
        }
    }
```

通过对autowiring的源码分析，我们可以看出，autowiring的实现过程：
a、对Bean的属性代调用 getBean()方法，完成依赖Bean的初始化和依赖注入。
b、将依赖Bean 的属性引用设置到被依赖的Bean属性上。
c、将依赖Bean的名称和被依赖Bean的名称存储在IoC容器的集合中。

Spring IoC 容器的autowiring 属性自动依赖注入是一个很方便的特性，可以简化开发时的配置，但是凡是都有两面性，自动属性依赖注入也有不足，首先，Bean的依赖关系在配置文件中无法很清楚地看出来，对于维护造成一定困难。其次，由于自动依赖注入是Spring 容器自动执行的，容器是不会智能判断的，如果配置不当，将会带来无法预料的后果，所以自动依赖注入特性在使用时还是综合考虑。