第四节课:spring bean的实例化过程

比如我们容器中 TulingLog tulingLog = tcx.getBean(TulingLog.class); **容器中的过程是什么?**

i1:>org.springframework.beans.factory.support.AbstractBeanFactory#getBean(java.lang.String)

i2>org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean

i2.1>:org.springframework.beans.factory.support.AbstractBeanFactory#transformedBeanName **转换**beanName

i2.2>:org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton **去缓存中获取**bean

i2.3>:org.springframework.beans.factory.support.AbstractBeanFactory#getObjectForBeanInstance **对缓存中的获取的**bean**进行后续处理**

i2.4>:org.springframework.beans.factory.support.AbstractBeanFactory#isPrototypeCurrentlyInCreation **判断原型**bean**的依赖注入**

i2.5>:org.springframework.beans.factory.support.AbstractBeanFactory#getParentBeanFactory **检查父容器加载**bean

i2.6>:org.springframework.beans.factory.support.AbstractBeanFactory#getMergedLocalBeanDefinition **将**bean**定义转为**RootBeanDifination

i2.7>:**检查**bean**的依赖(**bean**加载顺序的依赖)**

i2.8>:org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton**根据**scope **的添加来创建**bean

i3>:org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBean**创建**bean**的方法**

i4>org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#doCreateBean **真正的创建**bean**的逻辑**

i4.1>:org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBeanInstance **调用构造函数创建对象**

i4.2>:**判断是否需要提早暴露对象**(mbd.isSingleton() && this.allowCircularReferences && isSingletonCurrentlyInCreation(beanName));

i4.3>org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#addSingletonFactory **暴露对象解决循环依赖**

i4.4>:org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#populateBean **给创建的**bean**进行赋值**

i4.5>:org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#initializeBean**对**bean**进行初始化**

i4.5.1>:org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#invokeAwareMethods **调用**XXAware**接口**

i4.5.2>applyBeanPostProcessorsBeforeInitialization **调用**bean**的后置处理器进行对处理**

i4.5.3>org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#invokeInitMethods **对象的初始化方法**

i4.5.3.1>:org.springframework.beans.factory.InitializingBean#afterPropertiesSet **调用**InitializingBean**的方法**

i4.5.3.2>:String initMethodName = mbd.getInitMethodName(); **自定义的初始化方法**

i5>:org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#addSingleton **把创建好的实例化好的**bean**加载缓存中**

i6>:org.springframework.beans.factory.support.AbstractBeanFactory#getObjectForBeanInstance**对创建的**bean**进行后续的加工**

i1:>org.springframework.beans.factory.support.AbstractBeanFactory#getBean(java.lang.String)

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

i2>org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean

```
protected <T> T doGetBean(final String name, final Class<T> requiredType, final Object[] args, boolean typeCheckC
	/**
	 * 转换对应的beanName  你们可能认为传入进来的name 不就是beanName么?
	 * 传入进来的可能是别名,也有可能是是factoryBean
	 * 1）去除factoryBean的修饰符  name="&instA"=====>instA
	 * 2)取指定的alias所表示的最终beanName  比如传入的是别名为ia---->指向为instA的bean，那么就返回instA
	 **/
	final String beanName = transformedBeanName(name);
	Object bean;
/**
 * 设计的精髓
 * 检查实例缓存中对象工厂缓存中是包含对象(从这里返回的可能是实例话好的,也有可能是没有实例化好的)
 * 为什么要这段代码?
 * 因为单实例bean创建可能存主依赖注入的情况，而为了解决循环依赖问题，在对象刚刚创建好(属性还没有赋值)
 * 的时候，就会把对象包装为一个对象工厂暴露出去(加入到对象工厂缓存中),一但下一个bean要依赖他，就直接可以从缓存中获
 * */

 //直接从缓存中获取或者从对象工厂缓存去取。
	Object sharedInstance = getSingleton(beanName);
	if (sharedInstance != null && args == null) {
		if (logger.isDebugEnabled()) {
			if (isSingletonCurrentlyInCreation(beanName)) {
				logger.debug("Returning eagerly cached instance of singleton bean '" + beanName +
						"' that is not fully initialized yet - a consequence of a circular reference");
			}
			else {
				logger.debug("Returning cached instance of singleton bean '" + beanName + "'");
			}
```

```
            }


            /**
             * 若从缓存中的sharedInstance是原始的bean(属性还没有进行实例化,那么在这里进行处理)
             * 或者是factoryBean 返回的是工厂bean的而不是我们想要的getObject()返回的bean ,就会在这里处理
             **/
            bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
        }

        else {
            /**
             * 为什么spring对原型对象就不能解决循环依赖的问题了?
             * 因为spring ioc对原型对象不进行缓存,所以无法提前暴露对象,每次调用都会创建新的对象.
             *
             * 比如对象A中的属性对象B,对象B中有属性A, 在创建A的时候 检查到依赖对象B，那么就会返过来创建对象B，在创建
             * 又发现依赖对象A,由于是原型对象的，ioc容器是不会对实例进行缓存的 所以无法解决循环依赖的问题
             *
             * */
            if (isPrototypeCurrentlyInCreation(beanName)) {
                throw new BeanCurrentlyInCreationException(beanName);
            }

            //获取父容器
            BeanFactory parentBeanFactory = getParentBeanFactory();
            //如果beanDefinitionMap中所有以及加载的bean不包含 本次加载的beanName，那么尝试取父容器取检测
            if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
                // Not found -> check parent.
                String nameToLookup = originalBeanName(name);
                if (args != null) {
                    //父容器递归查询
                    return (T) parentBeanFactory.getBean(nameToLookup, args);
                }
                else {
                    // No args -> delegate to standard getBean method.
                    return parentBeanFactory.getBean(nameToLookup, requiredType);
                }
            }

        //如果这里不是做类型检查，而是创建bean,这里需要标记一下.
            if (!typeCheckOnly) {
                markBeanAsCreated(beanName);
            }

            try {
                /**
            合并父 BeanDefinition 与子 BeanDefinition，后面会单独分析这个方法
                 *
                 * */
                final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
                checkMergedBeanDefinition(mbd, beanName, args);

                //用来处理bean加载的顺序依赖 比如要创建instA 的情况下 必须需要先创建instB
                /**
                 * <bean id="beanA" class="BeanA" depends-on="beanB">
            <bean id="beanB" class="BeanB" depends-on="beanA">
            创建A之前 需要创建B 创建B之前需要创建A 就会抛出异常
                 * */
                String[] dependsOn = mbd.getDependsOn();
                if (dependsOn != null) {
                    for (String dep : dependsOn) {
                        if (isDependent(beanName, dep)) {
```

```
                        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                                "Circular depends-on relationship between '" + beanName + "' and '" + dep
                    }
                    //注册依赖
                    registerDependentBean(dep, beanName);
                    try {
                        //优先创建依赖的对象
                        getBean(dep);
                    }
                    catch (NoSuchBeanDefinitionException ex) {
                        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                                "'" + beanName + "' depends on missing bean '" + dep + "'", ex);
                    }
                }
            }

            //创建bean（单例的）
            if (mbd.isSingleton()) {
                //创建单实例bean
                sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {
                    //在getSingleton房中进行回调用的
                    @Override
                    public Object getObject() throws BeansException {
                        try {
                            return createBean(beanName, mbd, args);
                        }
                        catch (BeansException ex) {
                            // Explicitly remove instance from singleton cache: It might have been put there
                            // eagerly by the creation process, to allow for circular reference resolution.
                            // Also remove any beans that received a temporary reference to the bean.
                            destroySingleton(beanName);
                            throw ex;
                        }
                    }
                });
                bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
            }

        //创建非单实例bean
            else if (mbd.isPrototype()) {
                // It's a prototype -> create a new instance.
                Object prototypeInstance = null;
                try {

                    beforePrototypeCreation(beanName);
                    prototypeInstance = createBean(beanName, mbd, args);
                }
                finally {
                    afterPrototypeCreation(beanName);
                }
                bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
            }

            else {
                String scopeName = mbd.getScope();
                final Scope scope = this.scopes.get(scopeName);
                if (scope == null) {
                    throw new IllegalStateException("No Scope registered for scope name '" + scopeName + "'
                }
                try {
                    Object scopedInstance = scope.get(beanName, new ObjectFactory<Object>() {
                        @Override
```

```
                                public Object getObject() throws BeansException {
                                        beforePrototypeCreation(beanName);
                                        try {
                                                return createBean(beanName, mbd, args);
                                        }
                                        finally {
                                                afterPrototypeCreation(beanName);
                                        }
                                }
                        });
                        bean = getObjectForBeanInstance(scopedInstance, name, beanName, mbd);
                }
                catch (IllegalStateException ex) {
                        throw new BeanCreationException(beanName,
                                        "Scope '" + scopeName + "' is not active for the current thread; consider " +
                                        "defining a scoped proxy for this bean if you intend to refer to it from a singletc
                                        ex);
                }
        }
        catch (BeansException ex) {
                cleanupAfterBeanCreationFailure(beanName);
                throw ex;
        }
    }

    // Check if required type matches the type of the actual bean instance.
    if (requiredType != null && bean != null && !requiredType.isInstance(bean)) {
        try {
            return getTypeConverter().convertIfNecessary(bean, requiredType);
        }
        catch (TypeMismatchException ex) {
            if (logger.isDebugEnabled()) {
                logger.debug("Failed to convert bean '" + name + "' to required type '" +
                        ClassUtils.getQualifiedName(requiredType) + "'", ex);
            }
            throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
        }
    }
    return (T) bean;
}
```

i2.2>:org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton **去缓存中获取**bean**源码分析**

```
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
   //去缓存map中获取以及实例化好的bean对象
    Object singletonObject = this.singletonObjects.get(beanName);
    //缓存中没有获取到,并且当前bean是否在正在创建
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
       //加锁，防止并发创建
        synchronized (this.singletonObjects) {
           //保存早期对象缓存中是否有该对象
            singletonObject = this.earlySingletonObjects.get(beanName);
            //早期对象缓存没有
            if (singletonObject == null && allowEarlyReference) {
               //早期对象暴露工厂缓存(用来解决循环依赖的)
                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    //调用方法获早期对象
```

```
                        singletonObject = singletonFactory.getObject();
                        //放入到早期对象缓存中
                        this.earlySingletonObjects.put(beanName, singletonObject);
                        this.singletonFactories.remove(beanName);
                    }
                }
            }
        }
        return (singletonObject != NULL_OBJECT ? singletonObject : null);
    }
```

i2.3>:org.springframework.beans.factory.support.AbstractBeanFactory#getObjectForBeanInstance

在Bean的生命周期中，getObjectForBeanInstance方法是频繁使用的方法，无论是从缓存中获取出来的bean还是根据scope创建出来的bean,都要通过该方法进行检查。

①:检查当前bean是否为factoryBean,如果是就需要调用该对象的getObject()方法来返回我们需要的bean对象

```
    protected Object getObjectForBeanInstance(
            Object beanInstance, String name, String beanName, RootBeanDefinition mbd) {

        //判断name为以 &开头的但是 又不是factoryBean类型的 就抛出异常
        if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance instanceof FactoryBean)) {
            throw new BeanIsNotAFactoryException(transformedBeanName(name), beanInstance.getClass());
        }

        /**
         * 现在我们有了这个bean，它可能是一个普通bean 也有可能是工厂bean
         * 1)若是工厂bean，我们使用他来创建实例，当如果想要获取的是工厂实例而不是工厂bean的getObject()对应的bean,我
         * */
        if (!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.isFactoryDereference(name)) {
            return beanInstance;
        }

    //加载factoryBean
        Object object = null;
        if (mbd == null) {
            /*
       * 如果 mbd 为空，则从缓存中加载 bean。FactoryBean 生成的单例 bean 会被缓存
       * 在 factoryBeanObjectCache 集合中，不用每次都创建
       */
            object = getCachedObjectForFactoryBean(beanName);
        }
        if (object == null) {
            // 经过前面的判断，到这里可以保证 beanInstance 是 FactoryBean 类型的，所以可以进行类型转换
            FactoryBean<?> factory = (FactoryBean<?>) beanInstance;
            // 如果 mbd 为空，则判断是否存在名字为 beanName 的 BeanDefinition
            if (mbd == null && containsBeanDefinition(beanName)) {
                //合并我们的bean定义
                mbd = getMergedLocalBeanDefinition(beanName);
            }

            boolean synthetic = (mbd != null && mbd.isSynthetic());

            // 调用 getObjectFromFactoryBean 方法继续获取实例
            object = getObjectFromFactoryBean(factory, beanName, !synthetic);
        }
        return object;
    }
```

```
=====================================核心方法==========getObjectFromFactoryBean()方法==
    protected Object getObjectFromFactoryBean(FactoryBean<?> factory, String beanName, boolean shouldPostProcess)
            /*
      * FactoryBean 也有单例和非单例之分，针对不同类型的 FactoryBean，这里有两种处理方式：
      *   1. 单例 FactoryBean 生成的 bean 实例也认为是单例类型。需放入缓存中，供后续重复使用
      *   2. 非单例 FactoryBean 生成的 bean 实例则不会被放入缓存中，每次都会创建新的实例
      */
        if (factory.isSingleton() && containsSingleton(beanName)) {
            //加锁，防止重复创建 可以使用缓存提高性能
            synchronized (getSingletonMutex()) {
                //从缓存中获取
                Object object = this.factoryBeanObjectCache.get(beanName);
                if (object == null) {
                    //没有获取到，使用factoryBean的getObject（）方法去获取对象
                    object = doGetObjectFromFactoryBean(factory, beanName);

                    Object alreadyThere = this.factoryBeanObjectCache.get(beanName);
                    if (alreadyThere != null) {
                        object = alreadyThere;
                    }
                    else {
                        if (object != null && shouldPostProcess) {
                            if (isSingletonCurrentlyInCreation(beanName)) {
                                // Temporarily return non-post-processed object, not storing it yet..
                                return object;
                            }
                            beforeSingletonCreation(beanName);
                            try {
                                //调用ObjectFactory的后置处理器
                                object = postProcessObjectFromFactoryBean(object, beanName);
                            }
                            catch (Throwable ex) {
                                throw new BeanCreationException(beanName,
                                        "Post-processing of FactoryBean's singleton object failed", ex);
                            }
                            finally {
                                afterSingletonCreation(beanName);
                            }
                        }
                        if (containsSingleton(beanName)) {
                            this.factoryBeanObjectCache.put(beanName, (object != null ? object : NULL_OBJECT));
                        }
                    }
                }
                return (object != NULL_OBJECT ? object : null);
            }
        }
        else {
            Object object = doGetObjectFromFactoryBean(factory, beanName);
            if (object != null && shouldPostProcess) {
                try {
                    object = postProcessObjectFromFactoryBean(object, beanName);
                }
                catch (Throwable ex) {
                    throw new BeanCreationException(beanName, "Post-processing of FactoryBean's object failed",
                }
            }
            return object;
        }
    }
```

```
==========================================doGetObjectFromFactoryBean()作用==================
private Object doGetObjectFromFactoryBean(final FactoryBean<?> factory, final String beanName)
                throws BeanCreationException {

        Object object;
        try {
           //安全检查
            if (System.getSecurityManager() != null) {
                AccessControlContext acc = getAccessControlContext();
                try {
                    object = AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {
                        @Override
                        public Object run() throws Exception {
                            //调用工厂bean的getObject()方法
                                return factory.getObject();
                            }
                        }, acc);
                }
                catch (PrivilegedActionException pae) {
                    throw pae.getException();
                }
            }
            else {
               //调用工厂bean的getObject()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 object creation", ex);
        }

        if (object == null && isSingletonCurrentlyInCreation(beanName)) {
            throw new BeanCurrentlyInCreationException(
                    beanName, "FactoryBean which is currently in creation returned null from getObject");
        }
        return object;
    }


==================================================postProcessObjectFromFactoryBean(object,
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#applyBeanPostProcessorsAfterInitializa
    @Override
    public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)
            throws BeansException {

        Object result = existingBean;
        for (BeanPostProcessor processor : getBeanPostProcessors()) {
            result = processor.postProcessAfterInitialization(result, beanName);
            if (result == null) {
                return result;
            }
        }
        return result;
    }
```

i2.6>:org.springframework.beans.factory.support.AbstractBeanFactory#getMergedLocalBeanDefinition **将bean定义转为**RootBeanDifination

合并父子bean**定义**

```
<bean id="tulingParentCompent" class="com.tuling.testparentsonbean.TulingParentCompent" abstract="true">
  <property name="tulingCompent" ref="tulingCompent"></property>
</bean>

<bean id="tulingSonCompent" class="com.tuling.testparentsonbean.TulingSonCompent" parent="tulingParentCompen

<bean id="tulingLog" class="com.tuling.testcreatebeaninst.TulingLog"></bean>

    protected RootBeanDefinition getMergedLocalBeanDefinition(String beanName) throws BeansException {
        //去合并的bean定义缓存中 判断当前的bean是否合并过
        RootBeanDefinition mbd = this.mergedBeanDefinitions.get(beanName);
        if (mbd != null) {
            return mbd;
        }
        //没有合并，调用合并分方法
        return getMergedBeanDefinition(beanName, getBeanDefinition(beanName));
    }

根据beanName获取到当前的bean定义信息
============================getBeanDefinition============================
    @Override
    public BeanDefinition getBeanDefinition(String beanName) throws NoSuchBeanDefinitionException {
        BeanDefinition bd = this.beanDefinitionMap.get(beanName);
        if (bd == null) {
            if (logger.isTraceEnabled()) {
                logger.trace("No bean named '" + beanName + "' found in " + this);
            }
            throw new NoSuchBeanDefinitionException(beanName);
        }
        return bd;
    }

    protected RootBeanDefinition getMergedBeanDefinition(String beanName, BeanDefinition bd)
            throws BeanDefinitionStoreException {

        return getMergedBeanDefinition(beanName, bd, null);
    }

protected RootBeanDefinition getMergedBeanDefinition(
            String beanName, BeanDefinition bd, BeanDefinition containingBd)
            throws BeanDefinitionStoreException {

        synchronized (this.mergedBeanDefinitions) {
            RootBeanDefinition mbd = null;

            //去缓存中获取一次bean定义
            if (containingBd == null) {
                mbd = this.mergedBeanDefinitions.get(beanName);
            }
      //尝试没有获取到
            if (mbd == null) {
               //当前bean定义是否有父bean
                if (bd.getParentName() == null) {  //没有
                    //转为rootBeanDefinaition 然后深度克隆返回
                    if (bd instanceof RootBeanDefinition) {
                        mbd = ((RootBeanDefinition) bd).cloneBeanDefinition();
                    }
                    else {
                        mbd = new RootBeanDefinition(bd);
                    }
                }
```

```
            else {  //有父bean
                //定义一个父的bean定义
                BeanDefinition pbd;
                try {
                    //获取父bena的名称
                    String parentBeanName = transformedBeanName(bd.getParentName());
                /** 判断父类 beanName 与子类 beanName 名称是否相同。若相同，则父类 bean 一定
* 在父容器中。原因也很简单，容器底层是用 Map 缓存 <beanName, bean> 键值对
* 的。同一个容器下，使用同一个 beanName 映射两个 bean 实例显然是不合适的。
* 有的朋友可能会觉得可以这样存储：<beanName, [bean1, bean2]> ，似乎解决了
* 一对多的问题。但是也有问题，调用 getName(beanName) 时，到底返回哪个 bean
* 实例好呢?
*/
                    if (!beanName.equals(parentBeanName)) {
                     /*
   * 这里再次调用 getMergedBeanDefinition，只不过参数值变为了
   * parentBeanName，用于合并父 BeanDefinition 和爷爷辈的
   * BeanDefinition。如果爷爷辈的 BeanDefinition 仍有父
   * BeanDefinition，则继续合并
   */

                        pbd = getMergedBeanDefinition(parentBeanName);
                    }
                    else {
                        //获取父容器
                        BeanFactory parent = getParentBeanFactory();
                        if (parent instanceof ConfigurableBeanFactory) {
                            //从父容器获取父bean的定义 //若父bean中有父bean 存储递归合并
                            pbd = ((ConfigurableBeanFactory) parent).getMergedBeanDefinition(parentBeanN
                        }
                        else {
                            throw new NoSuchBeanDefinitionException(parentBeanName,
                                    "Parent name '" + parentBeanName + "' is equal to bean name '" + be
                                    "': cannot be resolved without an AbstractBeanFactory parent");
                        }
                    }
                }
                catch (NoSuchBeanDefinitionException ex) {
                    throw new BeanDefinitionStoreException(bd.getResourceDescription(), beanName,
                            "Could not resolve parent bean definition '" + bd.getParentName() + "'", ex);
                }
                //以父 BeanDefinition 的配置信息为蓝本创建 RootBeanDefinition，也就是"已合并的 BeanDefinition"
                mbd = new RootBeanDefinition(pbd);
                //用子 BeanDefinition 中的属性覆盖父 BeanDefinition 中的属性
                mbd.overrideFrom(bd);
            }

            //若之前没有定义,就把当前的设置为单例的
            if (!StringUtils.hasLength(mbd.getScope())) {
                mbd.setScope(RootBeanDefinition.SCOPE_SINGLETON);
            }

            // A bean contained in a non-singleton bean cannot be a singleton itself.
            // Let's correct this on the fly here, since this might be the result of
            // parent-child merging for the outer bean, in which case the original inner bean
            // definition will not have inherited the merged outer bean's singleton status.
            if (containingBd != null && !containingBd.isSingleton() && mbd.isSingleton()) {
                mbd.setScope(containingBd.getScope());
            }

            // 缓存合并后的 BeanDefinition
            if (containingBd == null && isCacheBeanMetadata()) {
```

```
                this.mergedBeanDefinitions.put(beanName, mbd);
            }
        }

        return mbd;
    }
}
```

i2.8>:org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton**根据**scope**的添加来创建**bean

```
public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
        Assert.notNull(beanName, "'beanName' must not be null");

        synchronized (this.singletonObjects) {
            //从缓存中获取对象
            Object singletonObject = this.singletonObjects.get(beanName);
            if (singletonObject == null) {
                if (this.singletonsCurrentlyInDestruction) {
                    throw new BeanCreationNotAllowedException(beanName,
                            "Singleton bean creation not allowed while singletons of this factory are in destructio
                            "(Do not request a bean from a BeanFactory in a destroy method implementation!)");
                }
                if (logger.isDebugEnabled()) {
                    logger.debug("Creating shared instance of singleton bean '" + beanName + "'");
                }

                //打标.....把正在创建的bean 的标识设置为ture singletonsCurrentlyInDestruction
                beforeSingletonCreation(beanName);
                boolean newSingleton = false;
                boolean recordSuppressedExceptions = (this.suppressedExceptions == null);
                if (recordSuppressedExceptions) {
                    this.suppressedExceptions = new LinkedHashSet<Exception>();
                }
                try {
                    //调用单实例bean的创建
                    singletonObject = singletonFactory.getObject();
                    newSingleton = true;
                }
                catch (IllegalStateException ex) {
                    // Has the singleton object implicitly appeared in the meantime ->
                    // if yes, proceed with it since the exception indicates that state.
                    singletonObject = this.singletonObjects.get(beanName);
                    if (singletonObject == null) {
                        throw ex;
                    }
                }
                catch (BeanCreationException ex) {
                    if (recordSuppressedExceptions) {
                        for (Exception suppressedException : this.suppressedExceptions) {
                            ex.addRelatedCause(suppressedException);
                        }
                    }
                    throw ex;
                }
```

```
                    finally {
                        if (recordSuppressedExceptions) {
                            this.suppressedExceptions = null;
                        }
                        afterSingletonCreation(beanName);
                    }
                    if (newSingleton) {
                        //加载到缓存中
                        addSingleton(beanName, singletonObject);
                    }
                }
                return (singletonObject != NULL_OBJECT ? singletonObject : null);
            }
        }

==========================singletonObject = singletonFactory.getObject()=====================

                    sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {
                        @Override
                        public Object getObject() throws BeansException {
                            try {
                                return createBean(beanName, mbd, args);
                            }
                            catch (BeansException ex) {
                                // Explicitly remove instance from singleton cache: It might have been put there
                                // eagerly by the creation process, to allow for circular reference resolution.
                                // Also remove any beans that received a temporary reference to the bean.
                                destroySingleton(beanName);
                                throw ex;
                            }
                        }
                    });
================================================addSingleton(beanName, singletonObject);==
        protected void addSingleton(String beanName, Object singletonObject) {
            synchronized (this.singletonObjects) {
                //加入到缓存
                this.singletonObjects.put(beanName, (singletonObject != null ? singletonObject : NULL_OBJECT));
                //从早期对象缓存和解决依赖缓存中移除.....................
                this.singletonFactories.remove(beanName);
                this.earlySingletonObjects.remove(beanName);
                this.registeredSingletons.add(beanName);
            }
        }
```

i3>org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBean

```
protected Object createBean(String beanName, RootBeanDefinition mbd, Object[] args) throws BeanCreationException {
            if (logger.isDebugEnabled()) {
                logger.debug("Creating instance of bean '" + beanName + "'");
            }
            RootBeanDefinition mbdToUse = mbd;

        //根据bean定义和beanName解析class
            Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
            if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() != null) {
                mbdToUse = new RootBeanDefinition(mbd);
                mbdToUse.setBeanClass(resolvedClass);
            }
```

```
            // Prepare method overrides.
            try {
                  mbdToUse.prepareMethodOverrides();
            }
            catch (BeanDefinitionValidationException ex) {
                  throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),
                              beanName, "Validation of method overrides failed", ex);
            }

            try {

                  //给bean的后置处理器一个机会来生成一个代理对象返回,在aop模块进行详细讲解
                  Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
                  if (bean != null) {
                        return bean;
                  }
            }
            catch (Throwable ex) {
                  throw new BeanCreationException(mbdToUse.getResourceDescription(), beanName,
                              "BeanPostProcessor before instantiation of bean failed", ex);
            }

      //真正进行主要的业务逻辑方法来进行创建bean
            Object beanInstance = doCreateBean(beanName, mbdToUse, args);
            if (logger.isDebugEnabled()) {
                  logger.debug("Finished creating instance of bean '" + beanName + "'");
            }
            return beanInstance;
      }
```

i4:org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#doCreateBean **真正的创建**bean**的逻辑**

```
protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[] args)
                  throws BeanCreationException {

            // Instantiate the bean.
            BeanWrapper instanceWrapper = null;
            if (mbd.isSingleton()) {
                  instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
            }

            //调用构造方法创建bean的实例()
            if (instanceWrapper == null) {
               /**
                * 如果存在工厂方法则使用工厂方法进行初始化
                * 一个类有多个构造函数，每个构造函数都有不同的参数，所以需要根据参数锁定构造 函数并进行初始化。
                * 如果既不存在工厂方法也不存在带有参数的构造函数，则使用默认的构造函数进行 bean 的实例化
                * */
                  instanceWrapper = createBeanInstance(beanName, mbd, args);
            }
            final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);
            Class<?> beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() : null);
            mbd.resolvedTargetType = beanType;

            // Allow post-processors to modify the merged bean definition.
            synchronized (mbd.postProcessingLock) {
                  if (!mbd.postProcessed) {
```

```
                try {
                    /*
                    bean的后置处理器
                    *bean 合并后的处理， Autowired 注解正是通过此方法实现诸如类型的预解析。
                    **/
                        applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);

                }
                catch (Throwable ex) {
                        throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                                "Post-processing of merged bean definition failed", ex);
                }
                mbd.postProcessed = true;
            }
        }

    //判断当前bean是否需要暴露到 缓存对象中
        boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
                isSingletonCurrentlyInCreation(beanName));
        if (earlySingletonExposure) {
            if (logger.isDebugEnabled()) {
                logger.debug("Eagerly caching bean '" + beanName +
                        "' to allow for resolving potential circular references");
            }
            //暴露早期对象到缓存中用于解决依赖的。
            addSingletonFactory(beanName, new ObjectFactory<Object>() {
                @Override
                public Object getObject() throws BeansException {
                    return getEarlyBeanReference(beanName, mbd, bean);
                }
            });
        }

        // Initialize the bean instance.
        Object exposedObject = bean;
        try {
            //为当前的bean 填充属性，发现依赖等....解决循环依赖就是在这个地方
            populateBean(beanName, mbd, instanceWrapper);
            if (exposedObject != null) {
                //调用bean的后置处理器以及 initionalBean和自己自定义的方法进行初始化
                exposedObject = initializeBean(beanName, exposedObject, mbd);
            }
        }
        catch (Throwable ex) {
            if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName(
                throw (BeanCreationException) ex;
            }
            else {
                throw new BeanCreationException(
                        mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);
            }
        }

        if (earlySingletonExposure) {
            //去缓存中获取对象  只有bean 没有循环依赖 earlySingletonReference才会为空
            Object earlySingletonReference = getSingleton(beanName, false);
            if (earlySingletonReference != null) {
                //检查当前的Bean 在初始化方法中没有被增强过(代理过)
                if (exposedObject == bean) {
                    exposedObject = earlySingletonReference;
                }
                else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {
```

```
                        String[] dependentBeans = getDependentBeans(beanName);
                        Set<String> actualDependentBeans = new LinkedHashSet<String>(dependentBeans.length);
                        for (String dependentBean : dependentBeans) {
                            if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                                actualDependentBeans.add(dependentBean);
                            }
                        }
                        if (!actualDependentBeans.isEmpty()) {
                            throw new BeanCurrentlyInCreationException(beanName,
                                    "Bean with name '" + beanName + "' has been injected into other beans [" +
                                    StringUtils.collectionToCommaDelimitedString(actualDependentBeans) +
                                    "] in its raw version as part of a circular reference, but has eventually been " +
                                    "wrapped. This means that said other beans do not use the final version of the " +
                                    "bean. This is often the result of over-eager type matching - consider using " +
                                    "'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.");
                        }
                    }
                }
            }

            // Register bean as disposable.
            try {
               //注册 DisposableBean。 如果配置了 destroy-method，这里需要注册以便于在销毁时候调用。
                registerDisposableBeanIfNecessary(beanName, bean, mbd);
            }
            catch (BeanDefinitionValidationException ex) {
                throw new BeanCreationException(
                        mbd.getResourceDescription(), beanName, "Invalid destruction signature", ex);
            }

            return exposedObject;
        }
```

i4.1>:org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBeanInstance

调用构造函数创建对象

```
==================================createBeanInstance(String beanName, RootBeanDefinition mbd, Object[
因为一个类可能有多个构造函数，所以需要根据配置文件中配置的参数或者传入的参数确定最终调用的构造函数。因为判断过程会比较
所以Spring会将解析、确定好的构造函数缓存到BeanDefinition中的resolvedConstructorOrFactoryMethod字段中。在下次创建相同
会直接从RootBeanDefinition中的属性resolvedConstructorOrFactoryMethod缓存的值获取，避免再次解析

protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition mbd, Object[] args) {
            // Make sure bean class is actually resolved at this point.
            Class<?> beanClass = resolveBeanClass(mbd, beanName);

            if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers()) && !mbd.isNonPublicAccessAllowed()) {
                throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                        "Bean class isn't public, and non-public access not allowed: " + beanClass.getName());
            }

        // / 工厂方法不为空则使工厂方法初始化策略  也就是bean的配置过程中设置了factory-method方法
            if (mbd.getFactoryMethodName() != null)  {
                return instantiateUsingFactoryMethod(beanName, mbd, args);
            }


            boolean resolved = false;
```

```
            boolean autowireNecessary = false;
            if (args == null) {
                // 如果已缓存的解析的构造函数或者工厂方法不为空，则可以利用构造函数解析
            // 因为需要根据参数确认到底使用哪个构造函数，该过程比较消耗性能，所有采用缓存机制（缓存到bean定义中）
                    synchronized (mbd.constructorArgumentLock) {
                        if (mbd.resolvedConstructorOrFactoryMethod != null) {
                            resolved = true;
                            //从bean定义中解析出对应的构造函数
                            autowireNecessary = mbd.constructorArgumentsResolved;
                        }
                    }
            }
            //已经解析好了，直接注入即可
            if (resolved) {
                    if (autowireNecessary) {
                        //autowire 自动注入，调用构造函数自动注入
                        return autowireConstructor(beanName, mbd, null, null);
                    }
                    else {
                        //使用默认的构造函数
                        return instantiateBean(beanName, mbd);
                    }
            }

            //根据beanClass和beanName去bean的后置处理器中获取构造方法（SmartInstantiationAwareBeanPostProcessor --
            Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);
            if (ctors != null ||
                        mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_CONSTRUCTOR ||
                        mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args))  {
                    return autowireConstructor(beanName, mbd, ctors, args);
            }

            //使用
            return instantiateBean(beanName, mbd);
        }




==========================================提前暴露对象============================
        protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
            Assert.notNull(singletonFactory, "Singleton factory must not be null");
            synchronized (this.singletonObjects) {
                if (!this.singletonObjects.containsKey(beanName)) {
                    //把bean 作为objectFactory暴露出来.....
                    this.singletonFactories.put(beanName, singletonFactory);
                    this.earlySingletonObjects.remove(beanName);
                    this.registeredSingletons.add(beanName);
                }
            }
        }


/**
 * 在配置文件中是我们使用的@Bean的形式都是通过工厂方法的形式来实例化对象
 *
 * */


====================================autowireConstructor(beanName, mbd, ctors, args);很复杂 很复杂
1、autowireConstructor（带参）
```

对于实例的创建，Spring分为通用的实例化（默认无参构造函数），以及带有参数的实例化
下面代码是带有参数情况的实例化。因为需要确定使用的构造函数，所以需要有大量工作花在根据参数个数、类型来确定构造函数上（

```
    public BeanWrapper autowireConstructor(final String beanName, final RootBeanDefinition mbd,
            Constructor<?>[] chosenCtors, final Object[] explicitArgs) {

    //用于包装bean实例的
        BeanWrapperImpl bw = new BeanWrapperImpl();
        this.beanFactory.initBeanWrapper(bw);

    //定义一个参数用来保存  使用的构造函数
        Constructor<?> constructorToUse = null;
        //定义一个参数用来保存  使用的参数持有器
        ArgumentsHolder argsHolderToUse = null;
        //用于保存 使用的参数
        Object[] argsToUse = null;

    //判断传入的参数是不是空
        if (explicitArgs != null) {
           //赋值给argsToUse 然后执行使用
            argsToUse = explicitArgs;
        }
        //传入进来的是空,需要从配置文件中解析出来
        else {
           //存解析的参数
            Object[] argsToResolve = null;
            //加锁
            synchronized (mbd.constructorArgumentLock) {
               //从缓存中获取解析出来的构造参数
                constructorToUse = (Constructor<?>) mbd.resolvedConstructorOrFactoryMethod;
                //缓存中有构造方法
                if (constructorToUse != null && mbd.constructorArgumentsResolved) {
                    //从缓存中获取解析的参数
                    argsToUse = mbd.resolvedConstructorArguments;
                    if (argsToUse == null) {
                       //没有缓存的参数，就需要获取配置文件中配置的参数
                        argsToResolve = mbd.preparedConstructorArguments;
                    }
                }
            }
            //缓存中有构造器参数
            if (argsToResolve != null) {
               //解析参数类型， 如给定方法的构造函数 A( int , int ） 则通过此方法后就会把配置中的 / / （ "1", "l"）转换为
                argsToUse = resolvePreparedArguments(beanName, mbd, bw, constructorToUse, argsToResolve);
            }
        }
    //如果没有缓存，就需要从构造函数开始解析
        if (constructorToUse == null) {
            //是否需要解析构造函数
            boolean autowiring = (chosenCtors != null ||
                    mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_CONSTRUCTOR);
            ConstructorArgumentValues resolvedValues = null;

       //用来保存getBeans传入进来的参数的个数
            int minNrOfArgs;
            if (explicitArgs != null) {
                minNrOfArgs = explicitArgs.length;
            }
            else {
               //从bean定义中解析出来构造参数的对象
                ConstructorArgumentValues cargs = mbd.getConstructorArgumentValues();
                resolvedValues = new ConstructorArgumentValues();
```

```
                //计算出构造参数参数的个数
                minNrOfArgs = resolveConstructorArguments(beanName, mbd, bw, cargs, resolvedValues);
            }

            //如果传入的构造器数组不为空，就使用传入的构造器参数，否则通过反射获取class中定义的构造器
            Constructor<?>[] candidates = chosenCtors;
            //传入的构造参数为空
            if (candidates == null) {
                //解析出对应的bean的class
                Class<?> beanClass = mbd.getBeanClass();
                try {
                    //获取构造方法
                    candidates = (mbd.isNonPublicAccessAllowed() ?
                            beanClass.getDeclaredConstructors() : beanClass.getConstructors());
                }
                catch (Throwable ex) {
                    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                            "Resolution of declared constructors on bean Class [" + beanClass.getName() +
                            "] from ClassLoader [" + beanClass.getClassLoader() + "] failed", ex);
                }
            }
            ////给构造函数排序，public构造函数优先、参数数量降序排序
            AutowireUtils.sortConstructors(candidates);
            int minTypeDiffWeight = Integer.MAX_VALUE;
            //不确定的构造函数
            Set<Constructor<?>> ambiguousConstructors = null;
            LinkedList<UnsatisfiedDependencyException> causes = null;

        //根据从bean定义解析出来的参数个数来推算出构造函数
        //循环所有的构造函数 查找合适的构造函数
            for (Constructor<?> candidate : candidates) {
                //获取正在循环的构造函数的个数
                Class<?>[] paramTypes = candidate.getParameterTypes();

                if (constructorToUse != null && argsToUse.length > paramTypes.length) {
                    //如果找到了已经构造函数,并且已经确定的构造函数的参数个数>正在当前循环的 那么就直接返回(candi
                    break;
                }
                //参数个数不匹配 直接进入下一个循环
                if (paramTypes.length < minNrOfArgs) {
                    continue;
                }

                ArgumentsHolder argsHolder;
                //从bean定义中解析的构造函数的参数对象
                if (resolvedValues != null) {
                    try {
                        //从注解 @ConstructorProperties获取参数名称
                        String[] paramNames = ConstructorPropertiesChecker.evaluate(candidate, paramTypes.len
                        //没有获取到
                        if (paramNames == null) {
                            //去容器中获取一个参数探测器
                            ParameterNameDiscoverer pnd = this.beanFactory.getParameterNameDiscoverer();
                            if (pnd != null) {
                                //通过参数探测器去探测当前正在循环的构造参数
                                paramNames = pnd.getParameterNames(candidate);
                            }
                        }
                        // / 根据参数名称和数据类型创建参数持有器
                        argsHolder = createArgumentArray(beanName, mbd, resolvedValues, bw, paramTypes, pa
                                getUserDeclaredConstructor(candidate), autowiring);
                    }
```

```
                        catch (UnsatisfiedDependencyException ex) {
                            if (this.beanFactory.logger.isTraceEnabled()) {
                                this.beanFactory.logger.trace(
                                        "Ignoring constructor [" + candidate + "] of bean '" + beanName + "': " + ex
                            }
                            // Swallow and try next constructor.
                            if (causes == null) {
                                causes = new LinkedList<UnsatisfiedDependencyException>();
                            }
                            causes.add(ex);
                            continue;
                        }
                    }
                    else {
                        //解析出来的参数个数和从外面传递进来的个数不相等 进入下一个循环
                        if (paramTypes.length != explicitArgs.length) {
                            continue;
                        }
                        //把外面传递进来的参数封装为一个参数持有器
                        argsHolder = new ArgumentsHolder(explicitArgs);
                    }

                // / 探测是否有不确定性的构造函数存在， 例如不同构造函数的参数为父子关系
                    int typeDiffWeight = (mbd.isLenientConstructorResolution() ?
                            argsHolder.getTypeDifferenceWeight(paramTypes) : argsHolder.getAssignabilityWeight(par
                     //因为不同构造函数的参数个数相同，而且参数类型为父子关系，所以需要找出类型最符合的一个构造函数
                    //Spring用一种权重的形式来表示类型差异程度，差异权重越小越优先
                    if (typeDiffWeight < minTypeDiffWeight) {
                        //当前构造函数最为匹配的话，清空先前ambiguousConstructors列表
                        constructorToUse = candidate;
                        argsHolderToUse = argsHolder;
                        argsToUse = argsHolder.arguments;
                        minTypeDiffWeight = typeDiffWeight;
                        ambiguousConstructors = null;
                    }
                    ////存在相同权重的构造器，将构造器添加到一个ambiguousConstructors列表变量中
                    //注意,这时候constructorToUse 指向的仍是第一个匹配的构造函数
                    else if (constructorToUse != null && typeDiffWeight == minTypeDiffWeight) {
                        if (ambiguousConstructors == null) {
                            ambiguousConstructors = new LinkedHashSet<Constructor<?>>();
                            ambiguousConstructors.add(constructorToUse);
                        }
                        ambiguousConstructors.add(candidate);
                    }
                }

            //还是没有找到构造函数 就抛出异常
                if (constructorToUse == null) {
                    if (causes != null) {
                        UnsatisfiedDependencyException ex = causes.removeLast();
                        for (Exception cause : causes) {
                            this.beanFactory.onSuppressedException(cause);
                        }
                        throw ex;
                    }
                    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                            "Could not resolve matching constructor " +
                            "(hint: specify index/type/name arguments for simple parameters to avoid type ambiguitie
                }
                else if (ambiguousConstructors != null && !mbd.isLenientConstructorResolution()) {
                    throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                            "Ambiguous constructor matches found in bean '" + beanName + "' " +
```

```
                        "(hint: specify index/type/name arguments for simple parameters to avoid type ambiguitie
                        ambiguousConstructors);
            }

    //把解析出来的构造函数加入到缓存中
            if (explicitArgs == null) {
                argsHolderToUse.storeCache(mbd, constructorToUse);
            }
        }

    //调用构造函数进行反射创建
        try {
            Object beanInstance;

            if (System.getSecurityManager() != null) {
                final Constructor<?> ctorToUse = constructorToUse;
                final Object[] argumentsToUse = argsToUse;
                beanInstance = AccessController.doPrivileged(new PrivilegedAction<Object>() {
                    @Override
                    public Object run() {
                        return beanFactory.getInstantiationStrategy().instantiate(
                                mbd, beanName, beanFactory, ctorToUse, argumentsToUse);
                    }
                }, beanFactory.getAccessControlContext());
            }
            else {
                //获取生成实例策略类调用实例方法
                beanInstance = this.beanFactory.getInstantiationStrategy().instantiate(
                        mbd, beanName, this.beanFactory, constructorToUse, argsToUse);
            }

            bw.setBeanInstance(beanInstance);
            return bw;
        }
        catch (Throwable ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Bean instantiation via constructor failed", ex);
        }
    }

    ========================instantiate=========================================
    public Object instantiate(RootBeanDefinition bd, String beanName, BeanFactory owner,
            final Constructor<?> ctor, Object... args) {

        if (bd.getMethodOverrides().isEmpty()) {
            if (System.getSecurityManager() != null) {
                // use own privileged to change accessibility (when security is on)
                AccessController.doPrivileged(new PrivilegedAction<Object>() {
                    @Override
                    public Object run() {
                        ReflectionUtils.makeAccessible(ctor);
                        return null;
                    }
                });
            }
            //调用反射创建
            return BeanUtils.instantiateClass(ctor, args);
        }
        else {
            return instantiateWithMethodInjection(bd, beanName, owner, ctor, args);
        }
    }
```

i4.2>:**判断是否需要提早暴露对象**(mbd.isSingleton() && this.allowCircularReferences && isSingletonCurrentlyInCreation(beanName));

i4.3>org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#addSingletonFactory **暴露对象解决循环依赖**

```
    //判断当前bean是否需要暴露到 缓存对象中
        boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
                isSingletonCurrentlyInCreation(beanName));
        if (earlySingletonExposure) {
            if (logger.isDebugEnabled()) {
                logger.debug("Eagerly caching bean '" + beanName +
                        "' to allow for resolving potential circular references");
            }
            //暴露早期对象到缓存中用于解决依赖的。
            addSingletonFactory(beanName, new ObjectFactory<Object>() {
                @Override
                public Object getObject() throws BeansException {
                    return getEarlyBeanReference(beanName, mbd, bean);
                }
            });
        }

//暴露早期对象到缓存中用于解决依赖的。
  protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
        Assert.notNull(singletonFactory, "Singleton factory must not be null");
        synchronized (this.singletonObjects) {
            if (!this.singletonObjects.containsKey(beanName)) {
                this.singletonFactories.put(beanName, singletonFactory);
                this.earlySingletonObjects.remove(beanName);
                this.registeredSingletons.add(beanName);
            }
        }
    }
```

i4.4>:org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#populateBean **给创建的**bean**进行赋值**

```
//我们看这个方法没多长是不是？但是调用的细节比较复杂,不过我们看IOC 源码 需要抓主干，有些方式我们完全可以看做
//为一个黑盒方法,只要知道他是干什么的 不需要去每一行 每一行代码都去了解.....
protected void populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw) {

        //从bean定义中获取属性列表
        PropertyValues pvs = mbd.getPropertyValues();

        if (bw == null) {
            if (!pvs.isEmpty()) {
                throw new BeanCreationException(
                        mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");
            }
            else {
                // Skip property population phase for null instance.
                return;
            }
        }

    /*
    * 在属性被填充前，给 InstantiationAwareBeanPostProcessor 类型的后置处理器一个修改
```

```
 * bean 状态的机会。官方的解释是：让用户可以自定义属性注入。比如用户实现一
 * 个 InstantiationAwareBeanPostProcessor 类型的后置处理器，并通过
 * postProcessAfterInstantiation 方法向 bean 的成员变量注入自定义的信息。当然，如果无
 * 特殊需求，直接使用配置中的信息注入即可。
 */
        boolean continueWithPropertyPopulation = true;
        if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
            for (BeanPostProcessor bp : getBeanPostProcessors()) {
                if (bp instanceof InstantiationAwareBeanPostProcessor) {
                    InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
                    if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
                        continueWithPropertyPopulation = false;
                        break;
                    }
                }
            }
        }

    //上面返回为flase 表示你已经通过自己写的InstantiationAwareBeanPostProcessor 类型的处理器已经设置过bean的属性值了
        if (!continueWithPropertyPopulation) {
            return;
        }

    /**
     * 判断注入模型是不是byName 或者是byType的
        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME ||
                mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
            //封装属性列表
            MutablePropertyValues newPvs = new MutablePropertyValues(pvs);

            // 若是基于byName自动转入的
            if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
                autowireByName(beanName, mbd, bw, newPvs);
            }

            //计入byType自动注入的
            if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
                autowireByType(beanName, mbd, bw, newPvs);
            }
      //把处理过的 属性覆盖原来的
            pvs = newPvs;
        }

    /**
     * 判断有没有InstantiationAwareBeanPostProcessors类型的处理器
     *
     * */
        boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();

        /**
         * 判断是否需要依赖检查（默认是0）
         * DEPENDENCY_CHECK_NONE(0) 不做检查
         * DEPENDENCY_CHECK_OBJECTS(1) 只检查对象引用
         * DEPENDENCY_CHECK_SIMPLE(2)检查简单属性
         * DEPENDENCY_CHECK_ALL(3)检查所有的
         * */
        boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDENCY_CHECK_NONE);

     /*
     * 这里又是一种后置处理，用于在 Spring 填充属性到 bean 对象前，对属性的值进行相应的处理，
     * 比如可以修改某些属性的值。这时注入到 bean 中的值就不是配置文件中的内容了，
     * 而是经过后置处理器修改后的内容
```

```
        */
            if (hasInstAwareBpps || needsDepCheck) {
                //过滤出所有需要进行依赖检查的属性编辑器 并且进行缓存起来
                PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);

                //通过后置处理器来修改属性
                if (hasInstAwareBpps) {
                    for (BeanPostProcessor bp : getBeanPostProcessors()) {
                        if (bp instanceof InstantiationAwareBeanPostProcessor) {
                            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
                            pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanNam
                            if (pvs == null) {
                                return;
                            }
                        }
                    }
                }
                //需要检查的化 ， 那么需要检查依赖
                if (needsDepCheck) {
                    checkDependencies(beanName, mbd, filteredPds, pvs);
                }
            }

        //设置属性到beanWapper中
            applyPropertyValues(beanName, mbd, bw, pvs);
        }
    //上诉代码的作用
    1)获取了bw的属性列表
    2)在属性列表中被填充的之前，通过InstantiationAwareBeanPostProcessor 对bw的属性进行修改
    3)判断自动装配模型来判断是调用byTypeh还是byName
    4）再次应用后置处理，用于动态修改属性列表 pvs 的内容
    5）把属性设置到bw中


    我们就来分析=======================================autowireByName====================
        protected void autowireByName(
                    String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) {


        /**
         *  spring认为的简单属性
         *  1. CharSequence 接口的实现类，比如 String
         *  2. Enum
         *  3. Date
         *  4. URI/URL
         *  5. Number 的继承类，比如 Integer/Long
         *  6. byte/short/int... 等基本类型
         *  7. Locale
         *  8. 以上所有类型的数组形式，比如 String[]、Date[]、int[] 等等
         * 不包含在当前bean的配置文件的中属性 !pvs.contains(pd.getName()
         * */
            String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
            //循环解析出来的属性名称
            for (String propertyName : propertyNames) {
                //若当前循环的属性名称是当前bean中定义的属性
                if (containsBean(propertyName)) {
                    //去ioc中获取指定的bean对象
                    Object bean = getBean(propertyName);
                    //并且设置到当前bean中的pvs中
                    pvs.add(propertyName, bean);
                    //注册属性依赖
                    registerDependentBean(propertyName, beanName);
```

```
                if (logger.isDebugEnabled()) {
                    logger.debug("Added autowiring by name from bean name '" + beanName +
                            "' via property '" + propertyName + "' to bean named '" + propertyName + "'");
                }
            }
            else {
                if (logger.isTraceEnabled()) {
                    logger.trace("Not autowiring property '" + propertyName + "' of bean '" + beanName +
                            "' by name: no matching bean found");
                }
            }
        }
    }

我们就来分析==========================================autowireByType====================
protected void autowireByType(
            String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutablePropertyValues pvs) {

    //获取自定义类型的转换器
        TypeConverter converter = getCustomTypeConverter();
        //没有获取到  把bw赋值给转换器(BeanWrapper实现了 TypeConverter接口 )
        if (converter == null) {
            converter = bw;
        }

        Set<String> autowiredBeanNames = new LinkedHashSet<String>(4);
        //获取非简单bean的属性
        String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
        //循环属性
        for (String propertyName : propertyNames) {
            try {
                //bw中是否有该属性的描述器
                PropertyDescriptor pd = bw.getPropertyDescriptor(propertyName);
        //若是Object的属性 不做解析
                if (Object.class != pd.getPropertyType()) {
                /*
           * 获取 setter 方法（write method）的参数信息，比如参数在参数列表中的
           * 位置，参数类型，以及该参数所归属的方法等信息
           */
                    MethodParameter methodParam = BeanUtils.getWriteMethodParameter(pd);
                    // Do not allow eager init for type matching in case of a prioritized post-processor.
                    boolean eager = !PriorityOrdered.class.isAssignableFrom(bw.getWrappedClass());
                    // 创建依赖描述对象
                    DependencyDescriptor desc = new AutowireByTypeDependencyDescriptor(methodParam, eager

                    //过程比较复杂，先把这里看成一个黑盒，我们只要知道这个方法可以帮我们解析出合适的依赖即可
                    Object autowiredArgument = resolveDependency(desc, beanName, autowiredBeanNames, conve
                    if (autowiredArgument != null) {
                        /// 将解析出的 bean 存入到属性值列表（pvs）中
                        pvs.add(propertyName, autowiredArgument);
                    }
                    //循环调用，注册到bean的依赖中
                    for (String autowiredBeanName : autowiredBeanNames) {
                        registerDependentBean(autowiredBeanName, beanName);
                        if (logger.isDebugEnabled()) {
                            logger.debug("Autowiring by type from bean name '" + beanName + "' via property '"
                                    propertyName + "' to bean named '" + autowiredBeanName + "'");
                        }
                    }
                    autowiredBeanNames.clear();
                }
            }
```

```
                catch (BeansException ex) {
                    throw new UnsatisfiedDependencyException(mbd.getResourceDescription(), beanName, propertyNam
                }
            }
        }


==================================applyPropertyValues(beanName, mbd, bw, pvs);==============
    protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw, PropertyValues pvs) {
        if (pvs == null || pvs.isEmpty()) {
            return;
        }

        if (System.getSecurityManager() != null && bw instanceof BeanWrapperImpl) {
            ((BeanWrapperImpl) bw).setSecurityContext(getAccessControlContext());
        }

        MutablePropertyValues mpvs = null;
        List<PropertyValue> original;

        if (pvs instanceof MutablePropertyValues) {
            mpvs = (MutablePropertyValues) pvs;
            if (mpvs.isConverted()) { //如果属性列表 pvs 被转换过，则直接返回即可
                // Shortcut: use the pre-converted values as-is.
                try {
                    bw.setPropertyValues(mpvs);
                    return;
                }
                catch (BeansException ex) {
                    throw new BeanCreationException(
                            mbd.getResourceDescription(), beanName, "Error setting property values", ex);
                }
            }
            //获取bw中的属性列表
            original = mpvs.getPropertyValueList();
        }
        else {
            original = Arrays.asList(pvs.getPropertyValues());
        }

    //获取自定义的转化器
        TypeConverter converter = getCustomTypeConverter();
        if (converter == null) {
            converter = bw;
        }
        //获取bean定义的值解析器
        BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver(this, beanName, mbd, converte

        // Create a deep copy, resolving any references for values.
        List<PropertyValue> deepCopy = new ArrayList<PropertyValue>(original.size());
        boolean resolveNecessary = false;
        //循环属性集合
        for (PropertyValue pv : original) {
           //当前的属性值被转化过 添加到
            if (pv.isConverted()) {
               保存到集合中去
                deepCopy.add(pv);
            }
            else {
               //属性名
                String propertyName = pv.getName();
                //原始属性值
```

```
                    Object originalValue = pv.getValue();
                    //就是在该方法上来解决循环依赖的 解析出来的值
                    Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);
                    //把解析出来的赋值给转换的值
                    Object convertedValue = resolvedValue;
                    boolean convertible = bw.isWritableProperty(propertyName) &&
                            !PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);
                    if (convertible) {
                // 对属性值的类型进行转换，比如将 String 类型的属性值 "123" 转为 Integer 类型的 123
                        convertedValue = convertForProperty(resolvedValue, propertyName, bw, converter);
                    }
                    // Possibly store converted value in merged bean definition,
                    // in order to avoid re-conversion for every created bean instance.
                    if (resolvedValue == originalValue) {
                        if (convertible) {
                    //// 将 convertedValue 设置到 pv 中，后续再次创建同一个 bean 时，就无需再次进行转换了
                            pv.setConvertedValue(convertedValue);
                        }
                        deepCopy.add(pv);
                    }
                    /**
                    * 如果原始值 originalValue 是 TypedStringValue，且转换后的值
            * convertedValue 不是 Collection 或数组类型，则将转换后的值存入到 pv 中。
            */
                    else if (convertible && originalValue instanceof TypedStringValue &&
                            !((TypedStringValue) originalValue).isDynamic() &&
                            !(convertedValue instanceof Collection || ObjectUtils.isArray(convertedValue))) {
                        pv.setConvertedValue(convertedValue);
                        deepCopy.add(pv);
                    }
                    else {
                        resolveNecessary = true;
                        deepCopy.add(new PropertyValue(pv, convertedValue));
                    }
                }
            }
            if (mpvs != null && !resolveNecessary) {
                mpvs.setConverted();
            }

            // Set our (possibly massaged) deep copy.
            try {
                // 将所有的属性值设置到 bean 实例中
                bw.setPropertyValues(new MutablePropertyValues(deepCopy));
            }
            catch (BeansException ex) {
                throw new BeanCreationException(
                        mbd.getResourceDescription(), beanName, "Error setting property values", ex);
            }
        }

================================valueResolver.resolveValueIfNecessary====================
Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);
        public Object resolveValueIfNecessary(Object argName, Object value) {
            //判断解析的值是不是 运行时bean的引用
            if (value instanceof RuntimeBeanReference) {
                RuntimeBeanReference ref = (RuntimeBeanReference) value;
                //解析引用
                return resolveReference(argName, ref);
            }
            //若value 是RuntimeBeanNameReference
            else if (value instanceof RuntimeBeanNameReference) {
```

```
            String refName = ((RuntimeBeanNameReference) value).getBeanName();
            refName = String.valueOf(doEvaluate(refName));
            if (!this.beanFactory.containsBean(refName)) {
                throw new BeanDefinitionStoreException(
                        "Invalid bean name '" + refName + "' in bean reference for " + argName);
            }
            return refName;
        }
        //是BeanDefinitionHolder
        else if (value instanceof BeanDefinitionHolder) {
            // Resolve BeanDefinitionHolder: contains BeanDefinition with name and aliases.
            BeanDefinitionHolder bdHolder = (BeanDefinitionHolder) value;
            return resolveInnerBean(argName, bdHolder.getBeanName(), bdHolder.getBeanDefinition());
        }
        //BeanDefinition
        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_NAME_SEPARATOR +
                    ObjectUtils.getIdentityHexString(bd);
            return resolveInnerBean(argName, innerBeanName, bd);
        }
        //处理array的
        else if (value instanceof ManagedArray) {
            // May need to resolve contained runtime references.
            ManagedArray array = (ManagedArray) value;
            Class<?> elementType = array.resolvedElementType;
            if (elementType == null) {
                String elementTypeName = array.getElementTypeName();
                if (StringUtils.hasText(elementTypeName)) {
                    try {
                        elementType = ClassUtils.forName(elementTypeName, this.beanFactory.getBeanClassLoad
                        array.resolvedElementType = elementType;
                    }
                    catch (Throwable ex) {
                        // Improve the message by showing the context.
                        throw new BeanCreationException(
                                this.beanDefinition.getResourceDescription(), this.beanName,
                                "Error resolving array type for " + argName, ex);
                    }
                }
                else {
                    elementType = Object.class;
                }
            }
            return resolveManagedArray(argName, (List<?>) value, elementType);
        }
        else if (value instanceof ManagedList) {
            // May need to resolve contained runtime references.
            return resolveManagedList(argName, (List<?>) value);
        }
        else if (value instanceof ManagedSet) {
            // May need to resolve contained runtime references.
            return resolveManagedSet(argName, (Set<?>) value);
        }
        else if (value instanceof ManagedMap) {
            // May need to resolve contained runtime references.
            return resolveManagedMap(argName, (Map<?, ?>) value);
        }
        else if (value instanceof ManagedProperties) {
            Properties original = (Properties) value;
            Properties copy = new Properties();
```

```
            for (Map.Entry<Object, Object> propEntry : original.entrySet()) {
                Object propKey = propEntry.getKey();
                Object propValue = propEntry.getValue();
                if (propKey instanceof TypedStringValue) {
                    propKey = evaluate((TypedStringValue) propKey);
                }
                if (propValue instanceof TypedStringValue) {
                    propValue = evaluate((TypedStringValue) propValue);
                }
                copy.put(propKey, propValue);
            }
            return copy;
        }
        else if (value instanceof TypedStringValue) {
            // Convert value to target type here.
            TypedStringValue typedStringValue = (TypedStringValue) value;
            Object valueObject = evaluate(typedStringValue);
            try {
                Class<?> resolvedTargetType = resolveTargetType(typedStringValue);
                if (resolvedTargetType != null) {
                    return this.typeConverter.convertIfNecessary(valueObject, resolvedTargetType);
                }
                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 {
            return evaluate(value);
        }
    }

==============================================真正的解析bean的依赖引用==========
    private Object resolveReference(Object argName, RuntimeBeanReference ref) {
        try {
           //获取bean的引用的名称
            String refName = ref.getBeanName();
            //调用值解析器来解析bean的名称
            refName = String.valueOf(doEvaluate(refName));
            //判断父容器是否能够解析
            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");
                }

                return this.beanFactory.getParentBeanFactory().getBean(refName);
            }
            else {
               //解析出来的refName 去容器中获取bean(getBean->doGetBean。。。。。。。。。。)
                Object bean = this.beanFactory.getBean(refName);
                //保存到缓存中
                this.beanFactory.registerDependentBean(refName, this.beanName);
                return bean;
```

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

i4.5>org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#initializeBean**对bean进行初始化**

```
初始化方法
protected Object initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd) {
            if (System.getSecurityManager() != null) {
                AccessController.doPrivileged(new PrivilegedAction<Object>() {
                    @Override
                    public Object run() {
                        invokeAwareMethods(beanName, bean);
                        return null;
                    }
                }, getAccessControlContext());
            }
            else {
               //调用bean实现的 XXXAware接口
                invokeAwareMethods(beanName, bean);
            }

            Object wrappedBean = bean;
            if (mbd == null || !mbd.isSynthetic()) {
               //调用bean的后置处理器的before方法
                wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
            }

            try {
               //调用initianlBean的方法和自定义的init方法
                invokeInitMethods(beanName, wrappedBean, mbd);
            }
            catch (Throwable ex) {
                throw new BeanCreationException(
                        (mbd != null ? mbd.getResourceDescription() : null),
                        beanName, "Invocation of init method failed", ex);
            }
            if (mbd == null || !mbd.isSynthetic()) {
               //调用Bean的后置处理器的post方法
                wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
            }
            return wrappedBean;
        }

i4.5.1>:org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#invokeAwareMethods 调用XXAv
        private void invokeAwareMethods(final String beanName, final Object bean) {
            if (bean instanceof Aware) { //判断bean是否实现了Aware接口
                if (bean instanceof BeanNameAware) {  //实现了BeanNameAware接口
                    ((BeanNameAware) bean).setBeanName(beanName);
                }
                if (bean instanceof BeanClassLoaderAware) { //实现了BeanClassLoaderAware接口
```

```
                ((BeanClassLoaderAware) bean).setBeanClassLoader(getBeanClassLoader());
            }
            if (bean instanceof BeanFactoryAware) { //实现了BeanFactoryAware接口
                ((BeanFactoryAware) bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);
            }
        }
    }

/
i4.5.2>applyBeanPostProcessorsBeforeInitialization 调用bean的后置处理器进行对处理
    @Override
    public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName)
            throws BeansException {

        Object result = existingBean;
        for (BeanPostProcessor processor : getBeanPostProcessors()) {
            //调用所有的后置处理器的before的方法
            result = processor.postProcessBeforeInitialization(result, beanName);
            if (result == null) {
                return result;
            }
        }
        return result;
    }

i4.5.3>org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#invokeInitMethods 对象的初始化
    protected void invokeInitMethods(String beanName, final Object bean, RootBeanDefinition mbd)
            throws Throwable {

    //判断你的bean 是否实现了 InitializingBean接口
        boolean isInitializingBean = (bean instanceof InitializingBean);
        if (isInitializingBean && (mbd == null || !mbd.isExternallyManagedInitMethod("afterPropertiesSet"))) {
            if (logger.isDebugEnabled()) {
                logger.debug("Invoking afterPropertiesSet() on bean with name '" + beanName + "'");
            }
            if (System.getSecurityManager() != null) {
                try {
                    AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {
                        @Override
                        public Object run() throws Exception {
                            ((InitializingBean) bean).afterPropertiesSet();
                            return null;
                        }
                    }, getAccessControlContext());
                }
                catch (PrivilegedActionException pae) {
                    throw pae.getException();
                }
            }
            else {
                //调用了InitializingBean的afterPropertiesSet()方法
                ((InitializingBean) bean).afterPropertiesSet();
            }
        }

    //调用自己在配置bean的时候指定的初始化方法
        if (mbd != null) {
            String initMethodName = mbd.getInitMethodName();
            if (initMethodName != null && !(isInitializingBean && "afterPropertiesSet".equals(initMethodName)) &&
                    !mbd.isExternallyManagedInitMethod(initMethodName)) {
                invokeCustomInitMethod(beanName, bean, mbd);
            }
```

```
            }
        }
```

i5>:org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#addSingleton **把创建好的实例化好的**bean**加载缓存中**

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

i6>:org.springframework.beans.factory.support.AbstractBeanFactory#getObjectForBeanInstance**对创建的**bean**进行后续的加工**