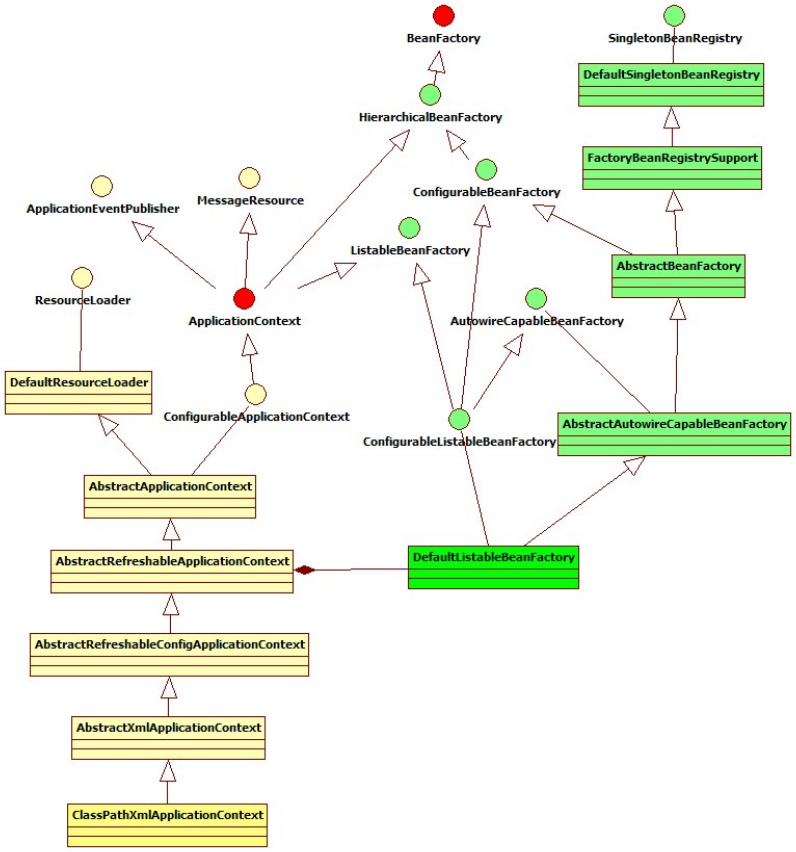
SpringIOC 核心源码学习：容器初始化过程

ClassPathXmlApplicationContext 类体系结构

以下是 ClassPathXmlApplicationContext 的类继承体系结构，理解这个结构有助于后面的代码理解。



左边黄色部分是 ApplicationContext 体系继承结构，右边是 BeanFactory 的结构体系,两个结构是典型模板方法设计模式的使用。
从该继承体系可以看出：

1. BeanFactory 是一个 bean 工厂的最基本定义，里面包含了一个 bean 工厂的几个最基本的方法，getBean(…) 、 containsBean(…) 等 ,是一个很纯粹的bean工厂，不关注资源、资源位置、事件等。ApplicationContext 是一个容器的最基本接口定义，它继承了 BeanFactory, 拥有工厂的基本方法。同时继承了 ApplicationEventPublisher 、 MessageSource 、 ResourcePatternResolver 等接口，使其 定义了一些额外的功能，如资源、事件等这些额外的功能。
2. AbstractBeanFactory 和 AbstractAutowireCapableBeanFactory 是两个模板抽象工厂类。AbstractBeanFactory 提供了 bean 工厂的抽象基类，同时提供了 ConfigurableBeanFactory 的完整实现。 AbstractAutowireCapableBeanFactory 是继承了 AbstractBeanFactory 的抽象工厂，里面提供了bean 创建的支持，包括 bean 的创建、依赖注入、检查等等功能，是一个核心的 bean 工厂基类。
3. ClassPathXmlApplicationContext之 所以拥有 bean 工厂的功能是通过持有一个真正的 bean 工厂DefaultListableBeanFactory 的实例，并通过 代理 该工厂完成。
4. ClassPathXmlApplicationContext 的初始化过程是对本身容器的初始化同时也是对其持有的DefaultListableBeanFactory 的初始化。

下面通过源码着重介绍一个容器的初始化过程，并重点理解 bean 的创建过程。

容器初始化过程

了解一个容器的大概过程是：

applicationContext.xml BeanDefinition Map Beans

整个过程可以理解为是容器的初始化过程。第一个过程是 ApplicationContext 的职责范围，第二步是BeanFactory 的职责范围。可以看出 ApplicationContext 是一个运行时的容器需要提供不容资源环境的支持，屏蔽不同环境的差异化。而 BeanDifinition 是内部关于 bean 定义的基本结构。 Bean 的创建就是基于它，回头会介绍一下改结构的定义。下面看一下整个容器的初始化过程。

容器的初始化是通过调用 refresh() 来实现。该方法是非常重要的一个方法，定义在AbstractApplicationContext 接口里。 AbstractApplicationContext 是容器的最基础的一个抽象父类。也就是说在该里面定义了一个容器初始化的基本流程，流程里的各个方法有些有提供了具体实现，有些是抽象的(因为不同的容器实例不一样) ，由继承它的每一个具体容器完成定制。看看 refresh 的基本流程：

Java代码

```
public void refresh() throws BeansException, IllegalStateException {
      synchronized (this.startupShutdownMonitor) {
         // Prepare this context for refreshing.
         prepareRefresh();
         // Tell the subclass to refresh the internal bean factory.
         ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
         // Prepare the bean factory for use in this context.
         prepareBeanFactory(beanFactory);
         try {
            // Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);
            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);
            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);
            // Initialize message source for this context.
            initMessageSource();
            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();
            // Initialize other special beans in specific context subclasses.
            onRefresh();
            // Check for listener beans and register them.
            registerListeners();
            // Instantiate all remaining (non-lazy-init) singletons.
            finishBeanFactoryInitialization(beanFactory);
            // Last step: publish corresponding event.
            finishRefresh();
         }
         catch (BeansException ex) {
            // Destroy already created singletons to avoid dangling resources.
            beanFactory.destroySingletons();
            // Reset 'active' flag.
            cancelRefresh(ex);
            // Propagate exception to caller.
            throw ex;
         }
      }
   }
```

解释如下：

Bean 的创建过程

Bean的创建过程基本是BeanFactory所要完成的事情.

根据以上过程，将会重点带着以下两个个问题来理解核心代码：

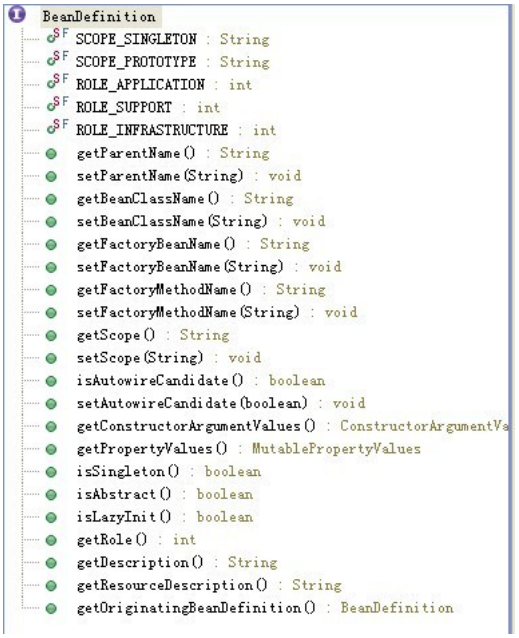
1.Bean 的创建时机

bean 是在什么时候被创建的，有哪些规则。

2.Bean 的创建过程

bean 是怎么创建的，会选择哪个构造函数？依赖如何注入？ InitializingBean 的 set 方法什么时候被调用？实现 ApplicationContextAware, BeanFactoryAware,BeanNameAware, ResourceLoaderAware 这些接口的 bean 的 set 方法何时被调用？

在解释这两个问题前，先看一下 BeanDefinition 接口的定义。



从该接口定义可以看出，通过 bean 定义能够得到 bean 的详细信息，如类名子、工厂类名称、 scope 、是否单例、是否抽象、是否延迟加载等等。基于此，来看一下以下两个问题：

问题 1 ： Bean 的创建时机

bean 是在什么时候被创建的，有哪些规则？

容器初始化的时候会预先对单例和非延迟加载的对象进行预先初始化。其他的都是延迟加载是在第一次调用getBean 的时候被创建。

从 DefaultListableBeanFactory 的 preInstantiateSingletons 里可以看到这个规则的实现。

Java代码

```
public void preInstantiateSingletons() throws BeansException {
    if (this.logger.isInfoEnabled()) {
        this.logger.info("Pre-instantiating singletons in " + this);
    }

    synchronized (this.beanDefinitionMap) {
        for (Iterator it = this.beanDefinitionNames.iterator(); it.hasNext();) {
            String beanName = (String) it.next();
            RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
            <span style="color: #ff0000;">if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {</span>



//对非抽象、单例的和非延迟加载的对象进行实例化。
                if (isFactoryBean(beanName)) {
                    FactoryBean factory = (FactoryBean) getBean(FACTORY_BEAN_PREFIX + beanName);
                    if (factory instanceof SmartFactoryBean && ((SmartFactoryBean) factory).isEagerInit()) {
                        getBean(beanName);
                    }
                }
                else {
                    getBean(beanName);
                }
            }
        }
    }
  }
```

从上面来看对于以下配置，只有 *singletonBean 会被预先创建。*

Xml代码

```
<?xml version="1.0" encoding="GB2312"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN" "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans default-autowire="byName">
   <bean id="otherBean"         class="com.test.OtherBean" scope="prototype"/>
   <bean id="myBean"         class="com.test.MyBean" lazy-init="true"/>
   <bean id="singletonBean"         class="com.test.SingletonBean"/>
</beans>
```

问题二：Bean 的创建过程

对于 bean 的创建过程其实都是通过调用工厂的 getBean 方法来完成的。这里面将会完成对构造函数的选择、依赖注入等。

无论预先创建还是延迟加载都是调用getBean实现，AbstractBeanFactory 定义了 getBean 的过程：

Java代码

```
protected Object doGetBean(
        final String name, final Class requiredType, final Object[] args, boolean typeCheckOnly) throws BeansException {
    final String beanName = transformedBeanName(name);
    Object bean = null;
    // Eagerly check singleton cache for manually registered singletons.
    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 + "'");
            }
        }
        bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
    }
    else {
        // Fail if we're already creating this bean instance:
        // We're assumably within a circular reference.
        if (isPrototypeCurrentlyInCreation(beanName)) {
            throw new BeanCurrentlyInCreationException(beanName);
```

```
        }

        // Check if bean definition exists in this factory.
        BeanFactory parentBeanFactory = getParentBeanFactory();
        if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
            // Not found -> check parent.
            String nameToLookup = originalBeanName(name);
            if (args != null) {
                // Delegation to parent with explicit args.
                return parentBeanFactory.getBean(nameToLookup, args);
            }
            else {
                // No args -> delegate to standard getBean method.
                return parentBeanFactory.getBean(nameToLookup, requiredType);
            }
        }

        if (!typeCheckOnly) {
            markBeanAsCreated(beanName);
        }

        final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
        checkMergedBeanDefinition(mbd, beanName, args);

        // Guarantee initialization of beans that the current bean depends on.
        String[] dependsOn = mbd.getDependsOn();
        if (dependsOn != null) {
            for (int i = 0; i < dependsOn.length; i++) {
                String dependsOnBean = dependsOn[i];
                getBean(dependsOnBean);
                registerDependentBean(dependsOnBean, beanName);
            }
        }
        // Create bean instance.
        if (mbd.isSingleton()) {//单例对象创建过程,间接通过getSingleton方法来创建，里面会实现将单例对象缓存

                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);
        }

        else if (mbd.isPrototype()) {//非单例对象创建

            // It's a prototype -> create a new instance.
            Object prototypeInstance = null;
            try {
                beforePrototypeCreation(beanName);
                prototypeInstance = createBean(beanName, mbd, args);<span style="color: #ff0000;">//直接调用
createBean</span>
            }
            finally {
                afterPrototypeCreation(beanName);
            }
            bean = getObjectForBeanInstance(prototypeInstance, name, beanName, mbd);
        }

        else {
            String scopeName = mbd.getScope();
            final Scope scope = (Scope) this.scopes.get(scopeName);
```

```
            if (scope == null) {
                throw new IllegalStateException("No Scope registered for scope '" + scopeName + "'");
            }
            try {
                Object scopedInstance = scope.get(beanName, new ObjectFactory() {
                    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 singleton",
                        ex);
            }
        }
    }
    // Check if required type matches the type of the actual bean instance.
    if (requiredType != null && bean != null && !requiredType.isAssignableFrom(bean.getClass())) {
        throw new BeanNotOfRequiredTypeException(name, requiredType, bean.getClass());
    }
    return bean;
}
```

GetBean 的大概过程：

1. 先试着从单例缓存对象里获取。
2. 从父容器里取定义，有则由父容器创建。
3. 如果是单例，则走单例对象的创建过程：在 spring 容器里单例对象和非单例对象的创建过程是一样的。都会调用父类 AbstractAutowireCapableBeanFactory 的 createBean 方法。 不同的是单例对象只创建一次并且需要缓存起来。 DefaultListableBeanFactory 的父类 DefaultSingletonBeanRegistry 提供了对单例对象缓存等支持工作。所以是单例对象的话会调用 DefaultSingletonBeanRegistry 的 getSingleton 方法，它会间接调用 AbstractAutowireCapableBeanFactory 的 createBean 方法。如果是 Prototype 多例则直接调用父类 AbstractAutowireCapableBeanFactory 的 createBean 方法。

bean的创建是由AbstractAutowireCapableBeanFactory来定义：

Java代码

```
protected Object createBean(final String beanName, final RootBeanDefinition mbd, final Object[] args)
        throws BeanCreationException {
    AccessControlContext acc = AccessController.getContext();
    return AccessController.doPrivileged(new PrivilegedAction() {
        public Object run() {
            if (logger.isDebugEnabled()) {
                logger.debug("Creating instance of bean '" + beanName + "'");
            }
            // Make sure bean class is actually resolved at this point.
            resolveBeanClass(mbd, beanName);
            // Prepare method overrides.
            try {
                mbd.prepareMethodOverrides();
            }
            catch (BeanDefinitionValidationException ex) {
                throw new BeanDefinitionStoreException(mbd.getResourceDescription(),
                        beanName, "Validation of method overrides failed", ex);
```

```
            }
            try {
                // Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.
                Object bean = resolveBeforeInstantiation(beanName, mbd);
                if (bean != null) {
                    return bean;
                }
            }
            catch (Throwable ex) {
                throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                        "BeanPostProcessor before instantiation of bean failed", ex);
            }
            Object beanInstance = doCreateBean(beanName, mbd, args);
            if (logger.isDebugEnabled()) {
                logger.debug("Finished creating instance of bean '" + beanName + "'");
            }
            return beanInstance;
        }
    }, acc);
}
```

createBean 会调用 doCreateBean 方法：

Java代码

```
protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[] args) {
    // Instantiate the bean.
    BeanWrapper instanceWrapper = null;
    if (mbd.isSingleton()) {
        instanceWrapper = (BeanWrapper) this.factoryBeanInstanceCache.remove(beanName);
    }
    if (instanceWrapper == null) {
        instanceWrapper = createBeanInstance(beanName, mbd, args);
    }
    final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);
    Class beanType = (instanceWrapper != null ? instanceWrapper.getWrappedClass() : null);

    // Allow post-processors to modify the merged bean definition.
    synchronized (mbd.postProcessingLock) {
        if (!mbd.postProcessed) {
            applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
            mbd.postProcessed = true;
        }
    }

    // Eagerly cache singletons to be able to resolve circular references
    // even when triggered by lifecycle interfaces like BeanFactoryAware.
    boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
            isSingletonCurrentlyInCreation(beanName));
    if (earlySingletonExposure) {
        if (logger.isDebugEnabled()) {
            logger.debug("Eagerly caching bean '" + beanName +
                    "' to allow for resolving potential circular references");
        }
        addSingletonFactory(beanName, new ObjectFactory() {
            public Object getObject() throws BeansException {
                return getEarlyBeanReference(beanName, mbd, bean);
            }
        });
    }
    // Initialize the bean instance.
    Object exposedObject = bean;
    try {
        populateBean(beanName, mbd, instanceWrapper);
        exposedObject = initializeBean(beanName, exposedObject, mbd);
    }
    catch (Throwable ex) {
        if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {
            throw (BeanCreationException) ex;
```

```
            }
            else {
                throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);
            }
        }
        if (earlySingletonExposure) {
            Object earlySingletonReference = getSingleton(beanName, false);
            if (earlySingletonReference != null) {
                if (exposedObject == bean) {
                    exposedObject = earlySingletonReference;
                }
                else if (!this.allowRawInjectionDespiteWrapping && hasDependentBean(beanName)) {
                    String[] dependentBeans = getDependentBeans(beanName);
                    Set actualDependentBeans = new LinkedHashSet(dependentBeans.length);
                    for (int i = 0; i < dependentBeans.length; i++) {
                        String dependentBean = dependentBeans[i];
                        if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {                    actualDependentBeans.add(de

                        }
                    }
                    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.
        registerDisposableBeanIfNecessary(beanName, bean, mbd);
        return exposedObject;
    }
```

doCreateBean 的流程：

1. 会创建一个 BeanWrapper 对象 用于存放实例化对象。
2. 如果没有指定构造函数，会通过反射拿到一个默认的构造函数对象，并赋予beanDefinition.resolvedConstructorOrFactoryMethod 。
3. 调用 spring 的 BeanUtils 的 instantiateClass 方法，通过反射创建对象。
4. applyMergedBeanDefinitionPostProcessors
5. populateBean(beanName, mbd, instanceWrapper); 根据注入方式进行注入。根据是否有依赖检查进行依赖检查。

执行 bean 的注入里面会选择注入类型：

Java代码

```
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.
        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_NAME) {
            autowireByName(beanName, mbd, bw, newPvs);
        }<span style="color: #ff0000;">//根据名字注入</span>




        // Add property values based on autowire by type if applicable.
        if (mbd.getResolvedAutowireMode() == RootBeanDefinition.AUTOWIRE_BY_TYPE) {
            autowireByType(beanName, mbd, bw, newPvs);
        }<span style="color: #ff0000;">//根据类型注入</span>
```

```
            pvs = newPvs;
        }
```

6. initializeBean(beanName, exposedObject, mbd);

判断是否实现了 BeanNameAware 、 BeanClassLoaderAware 等 spring 提供的接口，如果实现了，进行默认的注入。同时判断是否实现了 InitializingBean 接口，如果是的话，调用 afterPropertySet 方法。

Java代码

```
protected Object initializeBean(String beanName, Object bean, RootBeanDefinition mbd) {
        if (bean instanceof BeanNameAware) {
            ((BeanNameAware) bean).setBeanName(beanName);
        }
        if (bean instanceof BeanClassLoaderAware) {
            ((BeanClassLoaderAware) bean).setBeanClassLoader(getBeanClassLoader());
        }
        if (bean instanceof BeanFactoryAware) {
            ((BeanFactoryAware) bean).setBeanFactory(this);
        }
        Object wrappedBean = bean;
        if (mbd == null || !mbd.isSynthetic()) {
            wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
        }
<span style="color: #ff0000;">      try {
            invokeInitMethods(beanName, wrappedBean, mbd);
        }</span>




        catch (Throwable ex) {
            throw new BeanCreationException(
                    (mbd != null ? mbd.getResourceDescription() : null),
                    beanName, "Invocation of init method failed", ex);
        }

        if (mbd == null || !mbd.isSynthetic()) {
            wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
        }
        return wrappedBean;
    }
```

其中invokeInitMethods实现如下：

Java代码

```
protected void invokeInitMethods(String beanName, Object bean, RootBeanDefinition mbd)
        throws Throwable {

    boolean isInitializingBean = (bean instanceof InitializingBean);
    if (isInitializingBean && (mbd == null || !mbd.isExternallyManagedInitMethod("afterPropertiesSet"))) {
        if (logger.isDebugEnabled()) {
            logger.debug("Invoking afterPropertiesSet() on bean with name '" + beanName + "'");
        }
        ((InitializingBean) bean).afterPropertiesSet();<span style="color: #ff0000;">//调用afterPropertiesSet方法</span>




    }

    String initMethodName = (mbd != null ? mbd.getInitMethodName() : null);
    if (initMethodName != null && !(isInitializingBean && "afterPropertiesSet".equals(initMethodName)) &&
            !mbd.isExternallyManagedInitMethod(initMethodName)) {
        invokeCustomInitMethod(beanName, bean, initMethodName, mbd.isEnforceInitMethod());
    }
}
```

总结

以上基本描述了 spring 容器的初始化过程和 bean 的创建过程。主要还是从大处着眼，很多细节没有涉及到。代码阅读总体感觉就是 spring 的代码抽象很好，结合结构读起来还是蛮顺利的。后面的学习将从细节着手。