spring IOC源码学习笔记

1.spring概述

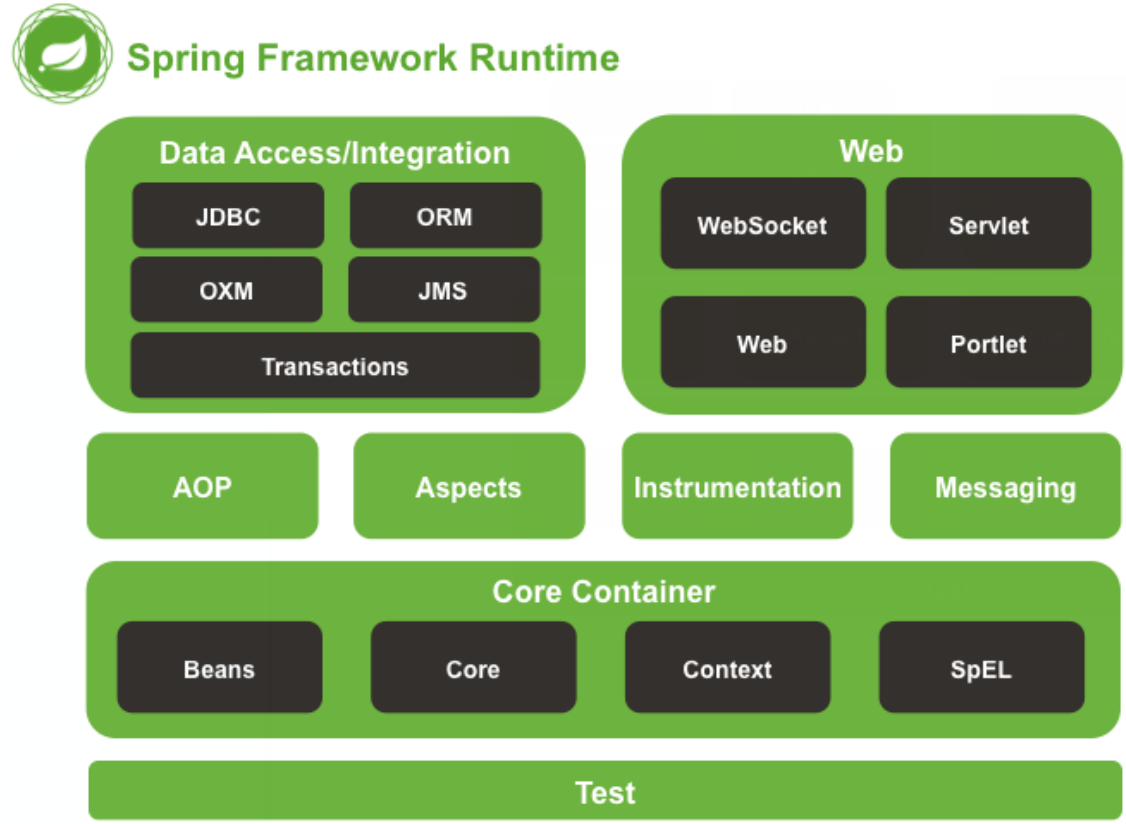
1.1 spring是什么

Spring是分层的Java SE/EE应用 full-stack轻量级开源框架，以IoC（Inverse Of Control：反转控制）和AOP（Aspect Oriented Programming：面向切面编程）为内核，提供了展现层Spring MVC和持久层Spring JDBC以及业务层事务管理等众多的企业级应用技术，还能整合开源世界众多著名的第三方框架和类库，逐渐成为使用最多的Java EE企业应用开源框架。

1.2 spring的优势

- 方便解耦，简化开发
  通过Spring提供的IoC容器，可以将对象间的依赖关系交由Spring进行控制，避免硬编码所造成的过度程序耦合。用户也不必再为单例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用
- AOP编程的支持
  通过Spring的AOP功能，方便进行面向切面的编程，许多不容易用传统OOP实现的功能可以通过AOP轻松应付。
- 声明式事务的支持
  可以将我们从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活的进行事务的管理，提高开发效率和质量。
- 方便程序的测试
  可以用非容器依赖的编程方式进行几乎所有的测试工作，测试不再是昂贵的操作，而是随手可做的事情。
- 方便集成各种优秀框架
  Spring可以降低各种框架的使用难度，提供了对各种优秀框架（Struts、Hibernate、Hessian、Quartz等）的直接支持。
- 降低JavaEE API的使用难度
  Spring对JavaEE API（如JDBC、JavaMail、远程调用等）进行了薄薄的封装层，使这些API的使用难度大为降低。
- Java源码是经典学习范例
  Spring的源代码设计精妙、结构清晰、匠心独用，处处体现着大师对Java设计模式灵活运用以及对Java技术的高深造诣。它的源代码无意是Java技术的最佳实践的范例。

1.3 spring的体系结构



2 spring IOC

2.1 程序耦合

耦合性(Coupling)，也叫耦合度，是对模块间关联程度的度量。耦合的强弱取决于模块间接口的复杂性、调用模块的方式以及通过界面传送数据的多少。模块间的耦合度是指模块之间的依赖关系，包括控制关系、调用关系、数据传递关系。模块间联系越多，其耦合性越强，同时表明其独立性越差(降低耦合性，可以提高其独立性)。耦合性存在于各个领域，而非软件设计中独有的，但是我们只讨论软件工程中的耦合。 在软件工程中，耦合指的就是就是对象之间的依赖性。对象之间的耦合越高，维护成本越高。因此对象的设计应使类和构件之间的耦合最小。软件设计中通常用耦合度和内聚度作为衡量模块独立程度的标准。划分模块的一个准则就是高内聚低耦合。 它有如下分类：

（1） 内容耦合。当一个模块直接修改或操作另一个模块的数据时，或一个模块不通过正常入口而转入另一个模块时，这样的耦合被称为内容耦合。内容耦合是最高程度的耦合，应该避免使用之。

（2） 公共耦合。两个或两个以上的模块共同引用一个全局数据项，这种耦合被称为公共耦合。在具有大量公共耦合的结构中，确定究竟是哪个模块给全局变量赋了一个特定的值是十分困难的。

（3） 外部耦合 。一组模块都访问同一全局简单变量而不是同一全局数据结构，而且不是通过参数表传递该全局变量的信息，则称之为外部耦合。

（4） 控制耦合 。一个模块通过接口向另一个模块传递一个控制信号，接受信号的模块根据信号值而进行适当的动作，这种耦合被称为控制耦合。

（5） 标记耦合 。若一个模块A通过接口向两个模块B和C传递一个公共参数，那么称模块B和C之间存在一个标记耦合。

（6） 数据耦合。模块之间通过参数来传递数据，那么被称为数据耦合。数据耦合是最低的一种耦合形式，系统中一般都存在这种类型的耦合，因为为了完成一些有意义的功能，往往需要将某些模块的输出数据作为另一些模块的输入数据。

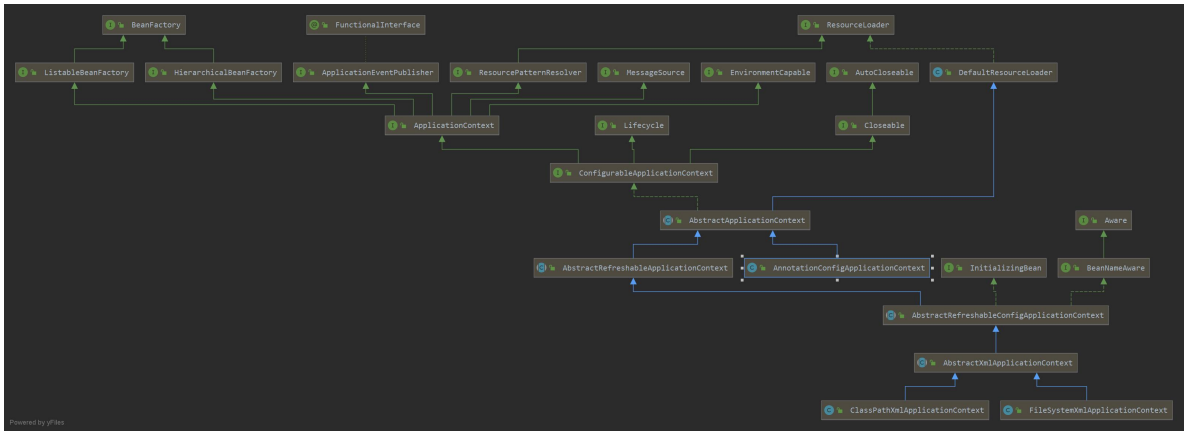
（7） 非直接耦合 。两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的。

总结： 耦合是影响软件复杂程度和设计质量的一个重要因素，在设计上我们应采用以下原则：如果模块间必须存在耦合，就尽量使用数据耦合，少用控制耦合，限制公共耦合的范围，尽量避免使用内容耦合。 内聚与耦合 内聚标志一个模块内各个元素彼此结合的紧密程度，它是信息隐蔽和局部化概念的自然扩展。内聚是从功能角度来度量模块内的联系，一个好的内聚模块应当恰好做一件事。它描述的是模块内的功能联系。耦合是软件结构中各模块之间相互连接的一种度量，耦合强弱取决于模块间接口的复杂程度、进入或访问一个模块的点以及通过接口的数据。 程序讲究的是低耦合，高内聚。就是同一个模块内的各个元素之间要高度紧密，但是各个模块之间的相互依存度却要不那么紧密。 内聚和耦合是密切相关的，同其他模块存在高耦合的模块意味着低内聚，而高内聚的模块意味着该模块同其他模块之间是低耦合。在进行软件设计时，应力争做到高内聚，低耦合。

2.2 什么是spring IOC

控制反转（Inversion of Control，缩写为**IoC**），是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。其中最常见的方式叫做**依赖注入**（Dependency Injection，简称**DI**），还有一种方式叫“依赖查找”（Dependency Lookup）。通过控制反转，对象在被创建的时候，由一个调控系统内所有对象的外界实体将其所依赖的对象的引用传递给它。也可以说，依赖被注入到对象中。控制反转是一种通过描述（在Java中可以是XML 或者注解）并通过第三方去产生或获取特定对象的方式。

Spring IoC 容器的设计主要是基于BeanFactory 和ApplicationContext 两个接口，其中ApplicationContext 是BeanFactory 的子接口之一，换句话说BeanFactory 是Spring IoC 容器所定义的最底层接口，而ApplicationContext 是其高级接口之一，并且对BeanFactory 功能做了许多有用的扩展，所以在绝大部分的工作场景下， 都会使用ApplicationContext 作为Spring IoC 容器， 其下图所示，图中展示的是Spring 相关的IoC 容器接口的主要设计。



2.3 spring IOC容器的初始化和依赖注入

Bean 的初始化和依赖注入在Spring IoC 容器中是两大步骤，bean是在初始化之后，才会进行依赖注入。

Bean 的初始化分为3 步：
1） Resource 定位，这步是Spring IoC 容器根据开发者的配置，进行资源定位，在Spring的开发中，通过XML 或者注解都是十分常见的方式，定位的内容是由开发者所提供的。
2） BeanDefinition 的载入，这个过程就是Spring 根据开发者的配置获取对应的POJO,用以生成对应实例的过程。
3） BeanDefinition 的注册，这个步骤就相当于把之前通过BeanDefinition 载入的POJO往Spring IoC 容器中注册，这样就可以使得开发和测试人员都可以通过描述从中得到Spring IoC 容器的Bean 了。
做完了这3 步， Bean 就在Spring IoC 容器中得到了初始化，但是没有完成依赖注入，也就是没有注入其配置的资源给B ean ，那么它还不能完全使用。对于依赖注入， Spri ng Bean还有一个配置选项——lazy-init ， 其含义就是是否初始化Spring Bean 。在没有任何配置的情况下，它的默认值为default ，实际值为false ， 也就是Spring IoC 默认会自动初始化Bean 。如果将其设置为true ，那么只有当我们使用Spring IoC 容器的getBean 方法获取它时，它才会进行初始化， 完成依赖注入。

- **2.3.1 bean的创建方式**

  1）使用默认构造函数创建

```
<!-- 第一种方式：使用默认构造函数创建。
            在spring的配置文件中使用bean标签，配以id和class属性之后，且没有其他属性和
标签时。
            采用的就是默认构造函数创建bean对象，此时如果类中没有默认构造函数，则对象无法
创建。
-->
<bean id="accountService"
class="com.shepherd.service.impl.AccountServiceImpl"></bean>
```

  2）使用普通工厂中的方法创建对象（使用某个类中的方法创建对象，并存入spring容器）

```
/**
 * 模拟一个工厂类（该类可能是存在于jar包中的，我们无法通过修改源码的方式来提供默认构造函
数）
 */
public class InstanceFactory {

    public IAccountService getAccountService(){
        return new AccountServiceImpl();
    }
}
```

```
<!-- 第二种方式：  使用普通工厂中的方法创建对象（使用某个类中的方法创建对象，并存入spring
容器）
  -->
<bean id="instanceFactory" class="com.shepherd.factory.InstanceFactory">
</bean>
<bean id="accountService" factory-bean="instanceFactory" factory-
method="getAccountService"></bean>
```

  3）使用工厂中的静态方法创建对象（使用某个类中的静态方法创建对象，并存入spring容器）

```
/**
 * 模拟一个工厂类（该类可能是存在于jar包中的，我们无法通过修改源码的方式来提供默认构造函
数）
 */
public class StaticFactory {
    public static IAccountService getAccountService(){
        return new AccountServiceImpl();
    }
}
```

```
<!-- 第三种方式：使用工厂中的静态方法创建对象（使用某个类中的静态方法创建对象，并存入
spring容器)
 -->
<bean id="accountService" class="com.shepherd.factory.StaticFactory"
factory-method="getAccountService"></bean>
```

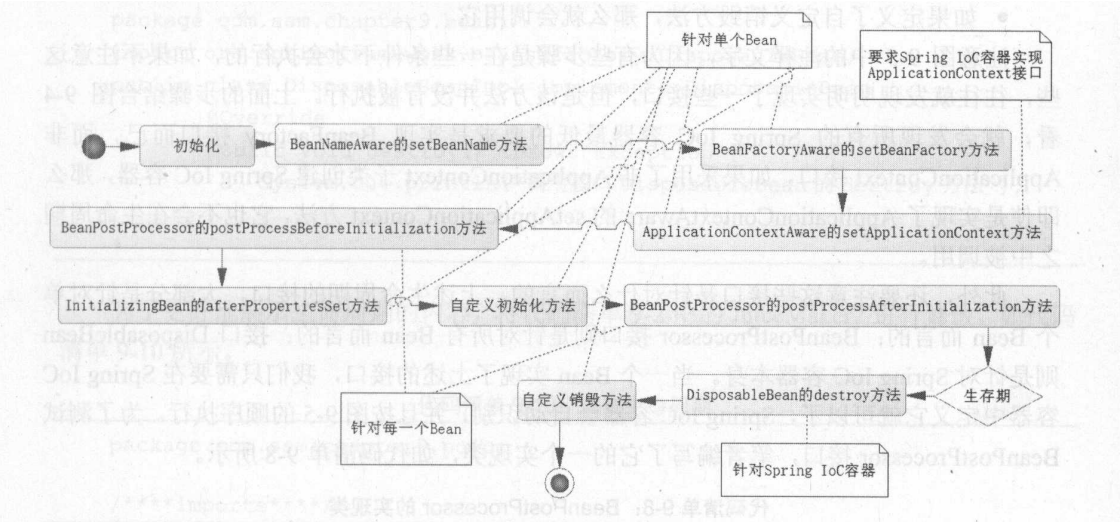
- **2.3.2 bean的作用范围**

```
<!-- bean的作用范围调整
        bean标签的scope属性：
            作用：用于指定bean的作用范围
            取值：  常用的就是单例的和多例的
                singleton：单例的（默认值）
                prototype：多例的
                request：作用于web应用的请求范围
                session：作用于web应用的会话范围
                global-session：作用于集群环境的会话范围（全局会话范围），当不是集群
环境时，它就是session
-->

<bean id="accountService"
class="com.shepherd.service.impl.AccountServiceImpl" scope="prototype">
</bean>
```

- **2.3.3 bean的生命周期**

```
<!-- bean对象的生命周期
            单例对象
                出生：当容器创建时对象出生
                活着：只要容器还在，对象一直活着
                死亡：容器销毁，对象消亡
                总结：单例对象的生命周期和容器相同
            多例对象
                出生：当我们使用对象时spring框架为我们创建
                活着：对象只要是在使用过程中就一直活着。
                死亡：当对象长时间不用，且没有别的对象引用时，由Java的垃圾回收器回收
--><bean id="accountService"
class="com.shepherd.service.impl.AccountServiceImpl"
          scope="singleton" init-method="init" destroy-method="destroy">
</bean>
```

生命周期主要是为了了解Spring IoC 容器初始化和销毁Bean 的过程，通过对它的学习就可以知道如何在初始化和销毁的时候加入自定义的方法，以满足特定的需求。图下图展示了Spring IoC 容器初始化和销毁Bean 的过程。



从上图可以看到， Spring IoC 容器对Bean 的管理还是比较复杂的， Spring IoC 容器在执行了初始化和依赖注入后，会执行一定的步骤来完成初始化，通过这些步骤我们就能自定义初始化，而在Spring IoC 容器正常关闭的时候，它也会执行一定的步骤来关闭容器，释放资源。除需要了解整个生命周期的步骤外，还要知道这些生命周期的接口是针对什么而言的， 首先介绍生命周期的步骤。

- 如果Bean 实现了接口BeanNameAware 的setBeanName 方法，那么它就会调用这个方法。
- 如果Bean 实现了接口BeanFactory Aware 的setBeanFactory 方法，那么它就会调用这个方法。
- 如果Bean 实现了接口App licationContextAware 的setApplicationContext 方法，且Spring IoC 容器也必须是一个ApplicationContext 接口的实现类，那么才会调用这个方法，否则是不调用的。
- 如果Bean 实现了接口BeanPostProcessor 的postProcessBeforelnitialization 方法，那么它就会调用这个方法。
- 如果Bean 实现了接口BeanF actory PostProcessor 的a企erProperti 已sS已t 方法，那么它就会调用这个方法。
- 如果Bean 自定义了初始化方法，它就会调用己定义的初始化方法。如果Bean 实现了接口BeanPostProcessor 的postProcessA武erlnitialization 方法，完成了这些调用，这个时候Bean 就完成了初始化，那么Bean 就生存在S pring IoC 的容器中了，使用者就可以从中获取Bean 的服务。

当服务器正常关闭，或者遇到其他关闭Spring IoC 容器的事件，它就会调用对应的方法完成Bean 的销毁，其步骤如下：

- 如果Bean 实现了接口DisposableBean 的destroy 方法，那么就会调用它。
- 如果定义了自定义销毁方法， 那么就会调用它。

- **2.3.4 依赖注入(DI)**

  依赖注入：Dependency Injection。它是spring框架核心ioc的具体实现。 我们的程序在编写时，通过控制反转，把对象的创建交给了spring，但是代码中不可能出现没有依赖的情况。ioc解耦只是降低他们的依赖关系，但不会消除。例如：我们的业务层仍会调用持久层的方法。 那这种业务层和持久层的依赖关系，在使用spring之后，就让spring来维护了。 简单的说，就是坐等框架把持久层对象传入业务层，而不用我们自己去获取。

  1）构造函数注入

```
<!--构造函数注入：
        使用的标签:constructor-arg
        标签出现的位置：bean标签的内部
        标签中的属性
```

```
            type：用于指定要注入的数据的数据类型，该数据类型也是构造函数中某个或某些参数的类型
            index：用于指定要注入的数据给构造函数中指定索引位置的参数赋值。索引的位置是从0开始
            name：用于指定给构造函数中指定名称的参数赋值
            常用的
            =============以上三个用于指定给构造函数中哪个参数赋值
            value：用于提供基本类型和String类型的数据
            ref：用于指定其他的bean类型数据。它指的就是在spring的Ioc核心容器中出现过的bean对象
        优势：
            在获取bean对象时，注入数据是必须的操作，否则对象无法创建成功。
        弊端：
            改变了bean对象的实例化方式，使我们在创建对象时，如果用不到这些数据，也必须提供。
    -->
    <bean id="user" class="com.shepherd.entity.User">
        <constructor-arg name="name" value="泰斯特"></constructor-arg>
        <constructor-arg name="age" value="18"></constructor-arg>
        <constructor-arg name="birthday" ref="now"></constructor-arg>
     </bean>

    <!-- 配置一个日期对象 -->
    <bean id="now" class="java.util.Date"></bean>
```

2）set方法注入

```
    <!-- set方法注入                    更常用的方式
        涉及的标签：property
        出现的位置：bean标签的内部
        标签的属性
            name：用于指定注入时所调用的set方法名称
            value：用于提供基本类型和String类型的数据
            ref：用于指定其他的bean类型数据。它指的就是在spring的Ioc核心容器中出现过的bean对象
        优势：
            创建对象时没有明确的限制，可以直接使用默认构造函数
        弊端：
            如果有某个成员必须有值，则获取对象是有可能set方法没有执行。
    -->
    <bean id="user2" class="com.shepherd.entity.User">
        <property name="name" value="TEST" ></property>
        <property name="age" value="21"></property>
        <property name="birthday" ref="now"></property>
    </bean>
```

3）复杂类型(集合类型)注入

```
    <!-- 复杂类型的注入/集合类型的注入
        用于给List结构集合注入的标签：
            list array set
        用于个Map结构集合注入的标签：
            map  props
        结构相同，标签可以互换
    -->
    <bean id="user3" class="com.shepherd.entity.User">
```

```
        <property name="arrays">
            <set>
                <value>AAA</value>
                <value>BBB</value>
                <value>CCC</value>
            </set>
        </property>

        <property name="list">
            <array>
                <value>AAA</value>
                <value>BBB</value>
                <value>CCC</value>
            </array>
        </property>

        <property name="set">
            <list>
                <value>AAA</value>
                <value>BBB</value>
                <value>CCC</value>
            </list>
        </property>

        <property name="map">
            <props>
                <prop key="testC">ccc</prop>
                <prop key="testD">ddd</prop>
            </props>
        </property>

        <property name="properties">
            <map>
                <entry key="testA" value="aaa"></entry>
                <entry key="testB">
                    <value>BBB</value>
                </entry>
            </map>
        </property>
    </bean>
```

2.4 spring IOC注解实现

- **@Component**

```
作用： 把资源让spring来管理。相当于在xml中配置一个bean。
属性： value：指定bean的id。如果不指定value属性，默认bean的id是当前类的类名。首字母小写
```

- **@Controller @Service @Repository**

下面三个注解都是针对一个的衍生注解，他们的作用及属性都是一模一样的。他们只不过是提供了更加明确的语义化。以下三个注解他们的作用和属性与Component是一模一样。是spring框架为我们提供明确的三层使用的注解，使我们的三层对象更加清晰

@Controller：一般用于表现层的注解。

@Service：一般用于业务层的注解。

@Repository：一般用于持久层的注解。

细节：如果注解中有且只有一个属性要赋值时，且名称是value，value在赋值是可以不写。

- **@Autowired**

自动按照类型注入。当使用注解注入属性时，set方法可以省略。它只能注入其他bean类型。当有多个类型匹配时，使用要注入的对象变量名称作为bean的id，在spring容器查找，找到了也可以注入成功。找不到就报错。

- **@Qualifier**

作用：在按照类中注入的基础之上再按照名称注入。它在给类成员注入时不能单独使用。但是在给方法参数注入时可以（稍后我们讲）

属性：value：用于指定注入bean的id。

- **@Resource**

作用： 直接按照Bean的id注入。它也只能注入其他bean类型。

属性： name：指定bean的id。

- **@Value**

作用： 注入基本数据类型和String类型数据的

属性： value：用于指定值

```
@Value("${jdbc.driver}")
private String driver;

@Value("${jdbc.url}")
private String url;

@Value("${jdbc.username}")
private String username;

@Value("${jdbc.password}")
private String password;
```

- **@Scope**

作用： 指定bean的作用范围。

属性： value：指定范围的值。

取值：singleton prototype request session globalsession

- **@PostConstruct**

用于指定初始化方法

- **@PreDestroy**

  > 用于指定销毁方法

以上注解示例代码如下：

```
package com.shepherd.service.impl;

import com.shepherd.dao.IAccountDao;
import com.shepherd.service.IAccountService;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.annotation.Resource;

/**
 * @author fjZheng
 * @version 1.0
 * @date 2021/1/26 17:41
*/

/**
 * 账户的业务层实现类
 *
 * 曾经XML的配置：
 *  <bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl"
 *         scope=""  init-method="" destroy-method="">
 *      <property name=""  value="" | ref=""></property>
 *  </bean>
 *
 * 用于创建对象的
 *      他们的作用就和在XML配置文件中编写一个<bean>标签实现的功能是一样的
 *      Component:
 *          作用：用于把当前类对象存入spring容器中
 *          属性：
 *              value：用于指定bean的id。当我们不写时，它的默认值是当前类名，且首字母改小
写。
 *      Controller：一般用在表现层
 *      Service：一般用在业务层
 *      Repository：一般用在持久层
 *      以上三个注解他们的作用和属性与Component是一模一样。
 *      他们三个是spring框架为我们提供明确的三层使用的注解，使我们的三层对象更加清晰
 *
 *
 * 用于注入数据的
 *      他们的作用就和在xml配置文件中的bean标签中写一个<property>标签的作用是一样的
 *      Autowired:
 *          作用：自动按照类型注入。只要容器中有唯一的一个bean对象类型和要注入的变量类型匹
配，就可以注入成功
 *                  如果ioc容器中没有任何bean的类型和要注入的变量类型匹配，则报错。
 *                  如果Ioc容器中有多个类型匹配时：
 *          出现位置：
 *              可以是变量上，也可以是方法上
 *          细节：
 *              在使用注解注入时，set方法就不是必须的了。
 *
```

```
 *      Qualifier:
 *          作用：在按照类中注入的基础之上再按照名称注入。它在给类成员注入时不能单独使用。但是在给方法参数注入时可以（稍后我们讲）
 *          属性：
 *              value：用于指定注入bean的id。
 *      Resource
 *          作用：直接按照bean的id注入。它可以独立使用
 *          属性：
 *              name：用于指定bean的id。
 *      以上三个注入都只能注入其他bean类型的数据，而基本类型和String类型无法使用上述注解实现。
 *      另外，集合类型的注入只能通过XML来实现。
 *
 *      Value
 *          作用：用于注入基本类型和String类型的数据
 *          属性：
 *              value：用于指定数据的值。它可以使用spring中SpEL(也就是spring的el表达式）
 *                      SpEL的写法：${表达式}
 *
 * 用于改变作用范围的
 *      他们的作用就和在bean标签中使用scope属性实现的功能是一样的
 *      Scope
 *          作用：用于指定bean的作用范围
 *          属性：
 *              value：指定范围的取值。常用取值：singleton prototype
 *
 * 和生命周期相关 了解
 *      他们的作用就和在bean标签中使用init-method和destroy-methode的作用是一样的
 *      PreDestroy
 *          作用：用于指定销毁方法
 *      PostConstruct
 *          作用：用于指定初始化方法
 */
@Service("accountService")
//@Scope("prototype")
public class AccountServiceImpl implements IAccountService {

//    @Autowired
//    @Qualifier("accountDao1")
    @Resource(name = "accountDao2")
    private IAccountDao accountDao = null;

    @PostConstruct
    public void  init(){
        System.out.println("初始化方法执行了");
    }

    @PreDestroy
    public void  destroy(){
        System.out.println("销毁方法执行了");
    }

    public void  saveAccount(){
        accountDao.saveAccount();
    }
}
```

- **@Configuration**

  作用： 用于指定当前类是一个spring配置类，当创建容器时会从该类上加载注解。获取容器时需要使用AnnotationApplicationContext(有@Configuration注解的类.class)。
  属性： value：用于指定配置类的字节码

- **@ComponentScan**

  作用： 用于指定spring在初始化容器时要扫描的包。作用和在spring的xml配置文件中的：<context:component-scan base-package="com.itheima"/>是一样的。
  属性： basePackages：用于指定要扫描的包。和该注解中的value属性作用一样。

- **@Bean**

  作用： 该注解只能写在方法上，表明使用此方法创建一个对象，并且放入spring容器。
  属性： name：给当前@Bean注解方法创建的对象指定一个名称(即bean的id）。

- **@PropertySource**

  作用：用于加载.properties文件中的配置。例如我们配置数据源时，可以把连接数据库的信息写到properties配置文件中，就可以使用此注解指定properties配置文件的位置。
  属性： value[]：用于指定properties文件位置。如果是在类路径下，需要写上classpath:

- **@Import**

  作用： 用于导入其他配置类，在引入其他配置类时，可以不用再写@Configuration注解。当然，写上也没问题。 属性： value[]：用于指定其他配置类的字节码。

以上注解示例代码：

```
package config;

import com.alibaba.druid.pool.DruidDataSource;
import org.apache.commons.dbutils.QueryRunner;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Scope;

import javax.sql.DataSource;

/**
 * 和spring连接数据库相关的配置类
 */
public class JdbcConfig {

    @Value("${jdbc.driver}")
    private String driver;

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;
```

```
    @Value("${jdbc.password}")
    private String password;

    /**
     * 用于创建一个QueryRunner对象
     * @param dataSource
     * @return
     */
    @Bean(name="runner")
    @Scope("prototype")
    public QueryRunner createQueryRunner(@Qualifier("ds") DataSource dataSource)
{
        return new QueryRunner(dataSource);
    }

    /**
     * 创建数据源对象
     * @return
     */
    @Bean(name="ds")
    public DataSource createDataSource(){
        try {
            DruidDataSource druidDataSource = new DruidDataSource();
            druidDataSource.setDriverClassName(driver);
            druidDataSource.setUrl(url);
            druidDataSource.setUsername(username);
            druidDataSource.setPassword(password);
            return druidDataSource;
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }

    @Bean(name="ds1")
    public DataSource createDataSource1(){
        try {
            DruidDataSource ds = new DruidDataSource();
            ds.setDriverClassName(driver);
            ds.setUrl("jdbc:mysql://localhost:3306/test");
            ds.setUsername(username);
            ds.setPassword(password);
            return ds;
        }catch (Exception e){
            throw new RuntimeException(e);
        }
    }
}
```

```
package config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.PropertySource;
```

```
/**
 * 该类是一个配置类，它的作用和bean.xml是一样的
 * spring中的新注解
 * Configuration
 *      作用：指定当前类是一个配置类
 *      细节：当配置类作为AnnotationConfigApplicationContext对象创建的参数时，该注解可以
不写。
 * ComponentScan
 *       作用：用于通过注解指定spring在创建容器时要扫描的包
 *       属性：
 *           value：它和basePackages的作用是一样的，都是用于指定创建容器时要扫描的包。
 *                  我们使用此注解就等同于在xml中配置了：
 *                       <context:component-scan base-package="com.itheima">
</context:component-scan>
 *  Bean
 *       作用：用于把当前方法的返回值作为bean对象存入spring的ioc容器中
 *       属性：
 *           name:用于指定bean的id。当不写时，默认值是当前方法的名称
 *       细节：
 *           当我们使用注解配置方法时，如果方法有参数，spring框架会去容器中查找有没有可用的
bean对象。
 *           查找的方式和Autowired注解的作用是一样的
 *  Import
 *       作用：用于导入其他的配置类
 *       属性：
 *           value：用于指定其他配置类的字节码。
 *                   当我们使用Import的注解之后，有Import注解的类就父配置类，而导入的都是子
配置类
 *  PropertySource
 *       作用：用于指定properties文件的位置
 *       属性：
 *           value：指定文件的名称和路径。
 *                   关键字：classpath，表示类路径下
 */
@Configuration
@ComponentScan("com.shepherd")
@Import(JdbcConfig.class)
@PropertySource("classpath:jdbcConfig.properties")
public class SpringConfiguration {


}
```

3.spring IOC源码解析

在这里我将基于xml 的配置的方式解析spring源码，虽然在实际开发中已经大部分使用注解方式，至少不是纯 xml 配置，不过从理解源码的角度来看用这种方式来说无疑是最合适的。对于注解方式的源码解析，我后续会跟进，敬请期待。

3.1 源码解析工程案例准备

1）新建maven工程，这里不介绍新建工程的过程(因为很简单)，然后引入spring的依赖，如下所示：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.0.2.RELEASE</version>
</dependency>
```

2） 我们在 resources 目录新建一个配置文件，文件名随意，如下面的bean.xml，在这里我们使用xml的配置方式

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--把对象的创建交给spring来管理-->
<bean id="accountService" class="com.shepherd.service.impl.AccountServiceImpl">
</bean>
<bean id="accountDao" class="com.shepherd.dao.impl.AccountDaoImpl"></bean>
</beans>
```

上面xml文件中全限定类名对应的类和接口，如下所示：

```
public interface AccountDao {

    void saveAccount();
}

public class AccountDaoImpl implements AccountDao {

    public  void saveAccount(){

        System.out.println("保存了账户");
    }
}


public interface IAccountService {

    void saveAccount();
}


public class AccountServiceImpl implements IAccountService {

    private AccountDao accountDao ;

    public AccountServiceImpl(){
        System.out.println("对象创建了");
    }

    public void  saveAccount(){
        accountDao.saveAccount();
    }
}
```

3）在Spring容器的设计中，有两个主要的容器系列，一个是实现BeanFactory接口的简单容器系列，这个接口实现了容器最基本的功能；另一个是ApplicationContext应用上下文，作为容器的高级形态而存在，它用于扩展BeanFactory中现有的功能。ApplicationContext和BeanFactory两者都是用于加载Bean的，但是相比之下，ApplicationContext提供了更多的扩展功能，简单一点说：ApplicationContext包含BeanFactory的所有功能。绝大多数“典型”的企业应用和系统，ApplicationContext就是你需要使用的。下面展示一下分别使用BeanFactory和ApplicationContext读取xml配置文件的方式：

```
public class Client {

    /**
     * 获取spring的Ioc核心容器，并根据id获取对象
     *
     * ApplicationContext的三个常用实现类：
     *      ClassPathXmlApplicationContext：它可以加载类路径下的配置文件，要求配置文件必须在类路径下。不在的话，加载不了。(更常用)
     *      FileSystemXmlApplicationContext：它可以加载磁盘任意路径下的配置文件(必须有访问权限)
     *
     *      AnnotationConfigApplicationContext：它是用于读取注解创建容器的，是明天的内容。
     *
     * 核心容器的两个接口引发出的问题：
     *  ApplicationContext:      单例对象适用                    采用此接口
     *      它在构建核心容器时，创建对象采取的策略是采用立即加载的方式。也就是说，只要一读取完配置文件马上就创建配置文件中配置的对象。
     *
     *  BeanFactory:             多例对象使用
     *      它在构建核心容器时，创建对象采取的策略是采用延迟加载的方式。也就是说，什么时候根据id获取对象了，什么时候才真正的创建对象。
     * @param args
     */
    public static void main(String[] args) {
        //1.获取核心容器对象
        ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");
//        ApplicationContext ac = new FileSystemXmlApplicationContext("C:\\Users\\zhy\\Desktop\\bean.xml");
//        2.根据id获取Bean对象
        IAccountService as  = (IAccountService)ac.getBean("accountService");

        IAccountDao adao = ac.getBean("accountDao",IAccountDao.class);

        System.out.println(as);
        System.out.println(adao);
        as.saveAccount();


        //--------BeanFactory----------
//        Resource resource = new ClassPathResource("bean.xml");
//        BeanFactory factory = new XmlBeanFactory(resource);
//        IAccountService as  = (IAccountService)factory.getBean("accountService");
//        IAccountService as1  = (IAccountService)factory.getBean("accountService");
//        System.out.println(as);
```

```
    }
}
```

3.2 通过debug解析源码

下面笔者将根据代码debug执行顺序进行解析

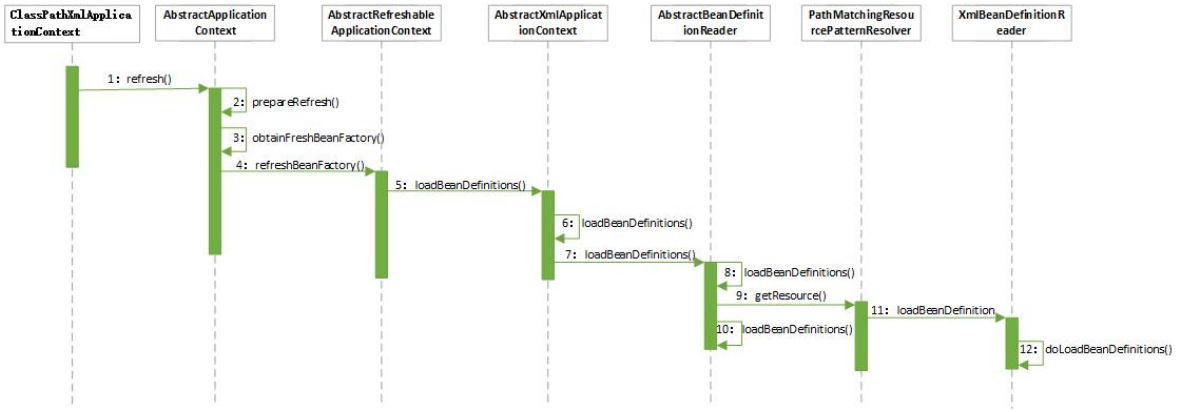
首先进入ClassPathXmlApplicationContext类

```
/**
     * Create a new ClassPathXmlApplicationContext with the given parent,
     * loading the definitions from the given XML files.
     * @param configLocations array of resource locations
     * @param refresh whether to automatically refresh the context,
     * loading all bean definitions and creating all singletons.
     * Alternatively, call refresh manually after further configuring the
context.
     * @param parent the parent context
     * @throws BeansException if context creation failed
     * @see #refresh()
     */

public ClassPathXmlApplicationContext(String[] configLocations, boolean refresh,
                          @Nullable ApplicationContext parent)throws
BeansException {

    super(parent);
    // 根据提供的路径，处理成配置文件数组(以分号、逗号、空格、tab、换行符分割)
    setConfigLocations(configLocations);
    if (refresh) {
        refresh(); //核心方法
    }
}
```

AbstractRefreshableConfigApplicationContext.class

```

    /**
     * Set the config locations for this application context.
     * <p>If not set, the implementation may use a default as appropriate.
     */
    public void setConfigLocations(@Nullable String... locations) {
        if (locations != null) {
            Assert.noNullElements(locations, "Config locations must not be
null");
            this.configLocations = new String[locations.length];
            for (int i = 0; i < locations.length; i++) {
                this.configLocations[i] = resolvePath(locations[i]).trim();
            }
        }
        else {
            this.configLocations = null;
        }
    }
```

下面我们来重点看看refresh()过程



简单来说，Spring容器的初始化是由refresh()方法来启动的，这个方法标志着IOC容器的正式启动。具体来说，这里的启动包括了BeanDefinition和Resource的定位、载入和注册三个基本过程

```
    public void refresh() throws BeansException, IllegalStateException {
        // 加锁，不然 refresh() 还没结束，你又来个启动或销毁容器的操作，那不就乱套了嘛
        synchronized (this.startupShutdownMonitor) {
            // Prepare this context for refreshing.
            //准备刷新容器（上下文环境）
            // 准备工作，记录下容器的启动时间、标记“已启动”状态、处理配置文件中的占位符
            prepareRefresh();

            // Tell the subclass to refresh the internal bean factory.
            //通知子类刷新内部bean工厂，初始化BeanFactory并进行XML文件的解析、读取
            // 这步比较关键，这步完成后，配置文件就会解析成一个个 Bean 定义，注册到
BeanFactory 中，
            // 当然，这里说的 Bean 还没有初始化，只是配置信息都提取出来了，
            // 注册也只是将这些信息都保存到了注册中心(说到底核心是一个 beanName->
beanDefinition 的                 map)
            ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();

            // Prepare the bean factory for use in this context.
            //准备bean工厂在此上下文中使用，并对BeanFactory进行各种功能填充
            // 设置 BeanFactory 的类加载器，添加几个 BeanPostProcessor，手动注册几个特
殊的 bean
            prepareBeanFactory(beanFactory);

            try {
                // Allows post-processing of the bean factory in context
subclasses.
                // 允许在上下文子类中对bean工厂进行后处理
                //Bean 如果实现了此接口，那么在容器初始化以后，Spring 会负责调用里面的
                            postProcessBeanFactory 方法。
                // 这里是提供给子类的扩展点，到这里的时候，所有的 Bean 都加载、注册完成了，
但是都还没                        有初始化
                // 具体的子类可以在这步的时候添加一些特殊的 BeanFactoryPostProcessor 的
实现类或做点                        什么事
                postProcessBeanFactory(beanFactory);

                // Invoke factory processors registered as beans in the context.
```

```
                // 在上下文中调用注册并激活BeanFactory的处理器，就是这个方法中的处理器注册了bean                    // 调用 BeanFactoryPostProcessor 各个实现类的 postProcessBeanFactory(factory)                    方法
                invokeBeanFactoryPostProcessors(beanFactory);

                // Register bean processors that intercept bean creation.
                // 注册拦截bean创建的bean处理器，这里只是注册，真正的调用Bean是在getBean方法的时候
                // 注册 BeanPostProcessor 的实现类，注意看和 BeanFactoryPostProcessor 的区别
                // 此接口两个方法: postProcessBeforeInitialization 和                    postProcessAfterInitialization
                // 两个方法分别在 Bean 初始化之前和初始化之后得到执行。注意，到这里 Bean 还没初始化
                registerBeanPostProcessors(beanFactory);

                // Initialize message source for this context.
                // 初始化此上下文的消息源，及不同语言的消息体，例如国际化处理
                initMessageSource();

                // Initialize event multicaster for this context.
                // 初始化此上下文的事件多播器
                initApplicationEventMulticaster();

                // Initialize other special beans in specific context subclasses.
                // 在特定上下文子类中初始化其他特殊bean.
                // 从方法名就可以知道，典型的模板方法
                // 具体的子类可以在这里初始化一些特殊的 Bean（在初始化 singleton beans 之前）
                onRefresh();

                // Check for listener beans and register them.
                // 检查监听器bean并注册它们
                registerListeners();

                // Instantiate all remaining (non-lazy-init) singletons.
                // 实例化所有剩余（非惰性的）单例
                //重点   重点    重点
                finishBeanFactoryInitialization(beanFactory);

                // Last step: publish corresponding event.
                // 完成刷新过程，通知生命周期处理器lifecycleProcessor刷新过程，同时发出                    ContextRefreshEvent通知别人
                finishRefresh();
            }

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

                // Destroy already created singletons to avoid dangling resources.
                // 摧毁已经创建的单例以避免资源浪费
                destroyBeans();
```

```
                // Reset 'active' flag.
                // 重置'有效'标志
                cancelRefresh(ex);

                // Propagate exception to caller.
                throw ex;
            }

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

接下来我将逐步解析reflash()里面的方法：主要分为以下几个阶段进行讲解

1）创建bean容器前的准备工作：

1.AbstractApplicationContext.class

```
protected void prepareRefresh() {
   // 记录启动时间，
   // 将 active 属性设置为 true，closed 属性设置为 false，它们都是 AtomicBoolean 类型
   this.startupDate = System.currentTimeMillis();
   this.closed.set(false);
   this.active.set(true);

   if (logger.isInfoEnabled()) {
      logger.info("Refreshing " + this);
   }

   // Initialize any placeholder property sources in the context environment
   initPropertySources();

   // 校验 xml 配置文件
   getEnvironment().validateRequiredProperties();

   this.earlyApplicationEvents = new LinkedHashSet<ApplicationEvent>();
```

2）创建 Bean 容器，加载并注册 Bean。通知子类刷新内部bean工厂，初始化BeanFactory并进行XML文件的解析、读取。obtain就是指获得的含义，这个方法obtaiinFreshBeanFactory正是实现BeanFactory的地方，也就是经过这个方法，ApplicationContext就已经拥有了BeanFactory的全部功能（也就是BeanFactory包含在了Spring容器里了）。注意，这个方法是全文最重要的部分之一，这里将会初始化BeanFactory、加载 Bean、注册 Bean 等等。当然，这步结束后，Bean 并没有完成初始化。

2.AbstractApplicationContext.java

```
    /**
     * Tell the subclass to refresh the internal bean factory.
     * @return the fresh BeanFactory instance
     * @see #refreshBeanFactory()
     * @see #getBeanFactory()
```

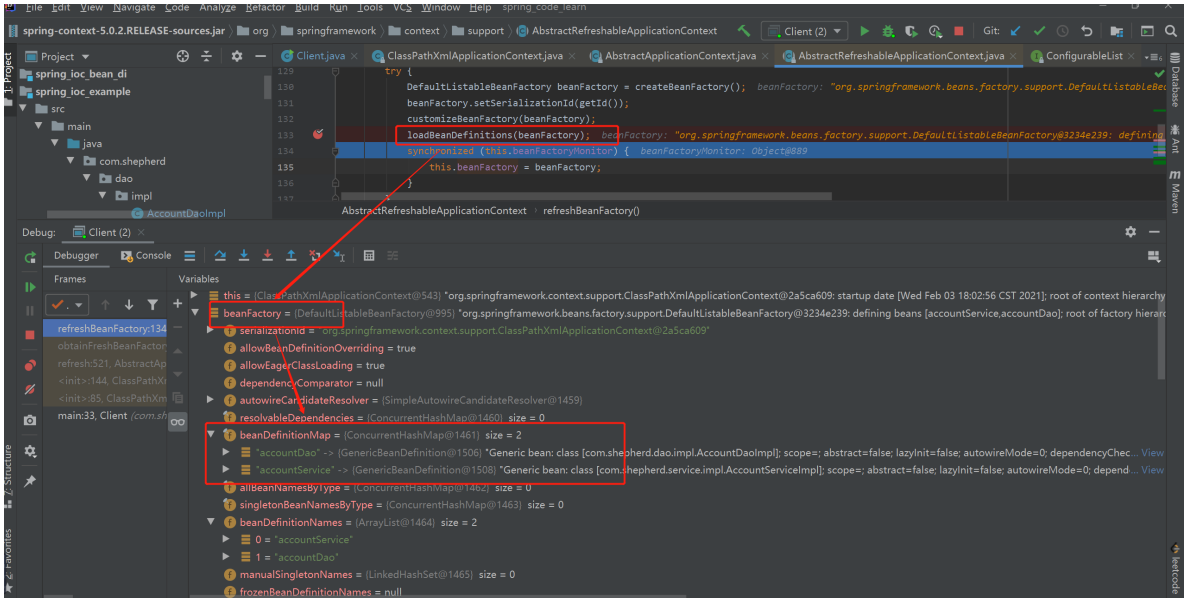
```
     */
    protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {

        // 关闭旧的 BeanFactory（如果有），创建新的 BeanFactory，加载 Bean 定义、注册
Bean 等
        refreshBeanFactory();

        // 返回刚刚创建的 BeanFactory
        ConfigurableListableBeanFactory beanFactory = getBeanFactory();
        if (logger.isDebugEnabled()) {
            logger.debug("Bean factory for " + getDisplayName() + ": " +
beanFactory);
        }
        return beanFactory;
    }
```

3.AbstractRefreshableApplicationContext .class

```
@Override
protected final void refreshBeanFactory() throws BeansException {
   // 如果 ApplicationContext 中已经加载过 BeanFactory 了，销毁所有 Bean，关闭
BeanFactory
   // 注意，应用中 BeanFactory 本来就是可以多个的，这里可不是说应用全局是否有
BeanFactory，而是当前
   // ApplicationContext 是否有 BeanFactory
   //如果有bean工厂，则关闭该工厂
   if (hasBeanFactory()) {
      destroyBeans();
      closeBeanFactory();
   }
   try {
      // 初始化一个 DefaultListableBeanFactory
      //创建一个新bean工厂，这里的DefaultListableBeanFactory就是前面笔者将的Spring核心
类，这个类真          的很重要!
      DefaultListableBeanFactory beanFactory = createBeanFactory();
      // 用于 BeanFactory 的序列化
      //为了序列化指定ID，如果需要的话，让这个BeanFactory从ID反序列化掉BeanFactory对象
      beanFactory.setSerializationId(getId());

      // 下面这两个方法很重要，别跟丢了，具体细节之后说
      // 设置 BeanFactory 的两个配置属性：是否允许 Bean 覆盖、是否允许循环引用
      //定制beanFactory，设置相关属性，包括是否允许覆盖同名称的不同定义的对象以及循环依赖以及
设置              @Autowired和@Qualifier注解解析器
QualifierAnnotationAutowireCandidateResolver
      customizeBeanFactory(beanFactory);

      // 加载 Bean 到 BeanFactory 中
      //加载bean定义信息，这一步实际上就从XML配置文件里的bean信息给读取到了Factory里了
      loadBeanDefinitions(beanFactory);
      synchronized (this.beanFactoryMonitor) {
         this.beanFactory = beanFactory;
      }
   }
   catch (IOException ex) {
      throw new ApplicationContextException("I/O error parsing bean definition
source for " + getDisplayName(), ex);
```

```
    }
}
```

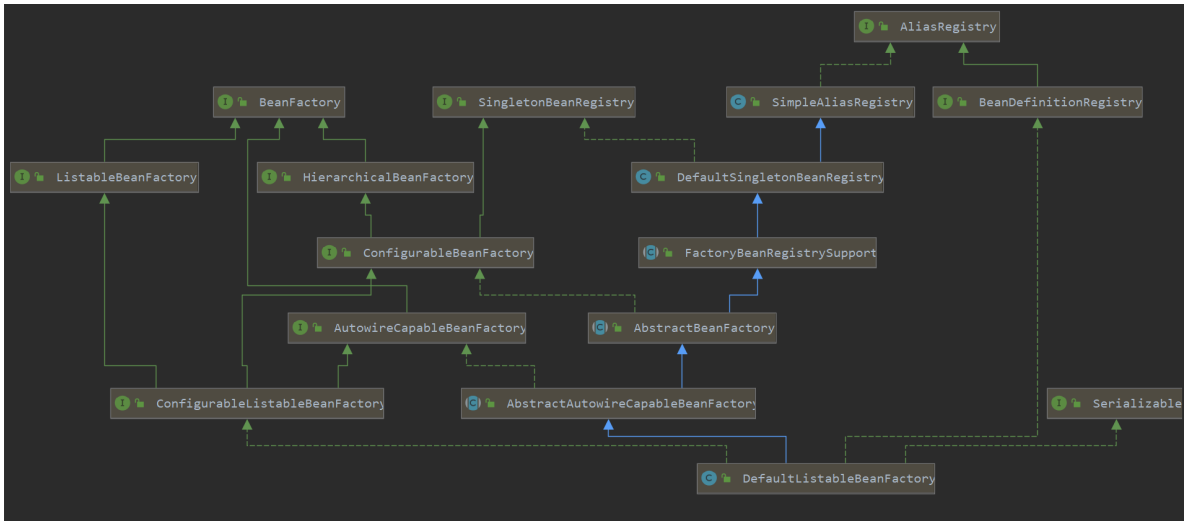
这里先看看上面代码的loadBeanDefinitions()方法运行完后的结果：



loadBeanDefinitions()方法运行完后，在beanFactory变量里面存放着一个ConcurrentHashMap变量，用于存放着beanDefinition, beanDefinition的getPropertyValues()获取bean的属性值。

看到这里的时候，我觉得读者就应该站在高处看 ApplicationContext 了，ApplicationContext 继承自 BeanFactory，但是它不应该被理解为 BeanFactory 的实现类，而是说其内部持有一个实例化的 BeanFactory（DefaultListableBeanFactory）。以后所有的 BeanFactory 相关的操作其实是给这个实例来处理的。

我们说说为什么选择实例化 DefaultListableBeanFactory？前面我们说了有个很重要的接口 ConfigurableListableBeanFactory，它实现了 BeanFactory 下面一层的所有三个接口，如下所示：



我们可以看到 ConfigurableListableBeanFactory 只有一个实现类 DefaultListableBeanFactory，而且实现类 DefaultListableBeanFactory 还通过实现右边的 AbstractAutowireCapableBeanFactory 通吃了右路。所以结论就是，最底下这个家伙 DefaultListableBeanFactory 基本上是最牛的 BeanFactory 了，这也是为什么这边会使用这个类来实例化的原因。

接下来先了解 BeanDefinition。**我们说BeanFactory是 Bean 容器，那么 Bean 又是什么呢?**

这里的 BeanDefinition 就是我们所说的 Spring 的 Bean，我们自己定义的各个 Bean 其实会转换成一个个 BeanDefinition 存在于 Spring 的 BeanFactory 中。

所以，如果有人问你 Bean 是什么的时候，你要知道 Bean 在代码层面上是 BeanDefinition 的实例。

BeanDefinition 中保存了我们的 Bean 信息，比如这个 Bean 指向的是哪个类、是否是单例的、是否懒加载、这个 Bean 依赖了哪些 Bean 等等。

BeanDefinition 接口定义 如下：

```
public interface BeanDefinition extends AttributeAccessor, BeanMetadataElement {

    // 我们可以看到，默认只提供 sington 和 prototype 两种，
    // 很多读者都知道还有 request, session, globalSession, application, websocket 这几种，
    // 不过，它们属于基于 web 的扩展。
    String SCOPE_SINGLETON = ConfigurableBeanFactory.SCOPE_SINGLETON;
    String SCOPE_PROTOTYPE = ConfigurableBeanFactory.SCOPE_PROTOTYPE;

    // 比较不重要，直接跳过吧
    int ROLE_APPLICATION = 0;
    int ROLE_SUPPORT = 1;
    int ROLE_INFRASTRUCTURE = 2;

    // 设置父 Bean，这里涉及到 bean 继承，不是 java 继承。请参见附录介绍
    void setParentName(String parentName);

    // 获取父 Bean
    String getParentName();

    // 设置 Bean 的类名称
    void setBeanClassName(String beanClassName);

    // 获取 Bean 的类名称
    String getBeanClassName();


    // 设置 bean 的 scope
    void setScope(String scope);

    String getScope();

    // 设置是否懒加载
    void setLazyInit(boolean lazyInit);

    boolean isLazyInit();

    // 设置该 Bean 依赖的所有的 Bean，注意，这里的依赖不是指属性依赖(如 @Autowire 标记的)，
    // 是 depends-on="" 属性设置的值。
    void setDependsOn(String... dependsOn);

    // 返回该 Bean 的所有依赖
    String[] getDependsOn();

    // 设置该 Bean 是否可以注入到其他 Bean 中，只对根据类型注入有效，
    // 如果根据名称注入，即使这边设置了 false，也是可以的
    void setAutowireCandidate(boolean autowireCandidate);

    // 该 Bean 是否可以注入到其他 Bean 中
    boolean isAutowireCandidate();
```

```
    // 主要的。同一接口的多个实现，如果不指定名字的话，Spring 会优先选择设置 primary 为 true
的 bean
    void setPrimary(boolean primary);

    // 是否是 primary 的
    boolean isPrimary();

    // 如果该 Bean 采用工厂方法生成，指定工厂名称。对工厂不熟悉的读者，请参加附录
    void setFactoryBeanName(String factoryBeanName);
    // 获取工厂名称
    String getFactoryBeanName();
    // 指定工厂类中的 工厂方法名称
    void setFactoryMethodName(String factoryMethodName);
    // 获取工厂类中的 工厂方法名称
    String getFactoryMethodName();

    // 构造器参数
    ConstructorArgumentValues getConstructorArgumentValues();

    // Bean 中的属性值，后面给 bean 注入属性值的时候会说到
    MutablePropertyValues getPropertyValues();

    // 是否 singleton
    boolean isSingleton();

    // 是否 prototype
    boolean isPrototype();

    // 如果这个 Bean 原生是抽象类，那么不能实例化
    boolean isAbstract();

    int getRole();
    String getDescription();
    String getResourceDescription();
    BeanDefinition getOriginatingBeanDefinition();
}
```

customizeBeanFactory(beanFactory) 比较简单，就是配置是否允许 BeanDefinition 覆盖、是否允许循环引用。

```
protected void customizeBeanFactory(DefaultListableBeanFactory beanFactory) {
    if (this.allowBeanDefinitionOverriding != null) {
        // 是否允许 Bean 定义覆盖

 beanFactory.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOverriding
);
    }
    if (this.allowCircularReferences != null) {
        // 是否允许 Bean 间的循环依赖
        beanFactory.setAllowCircularReferences(this.allowCircularReferences);
    }
}
```

BeanDefinition 的覆盖问题大家也许会碰到，就是在配置文件中定义 bean 时使用了相同的 id 或 name，默认情况下，allowBeanDefinitionOverriding 属性为 null，如果在同一配置文件中重复了，会抛错，但是如果不是同一配置文件中，会发生覆盖。

循环引用也很好理解：A 依赖 B，而 B 依赖 A。或 A 依赖 B，B 依赖 C，而 C 依赖 A。

默认情况下，Spring 允许循环依赖，当然如果你在 A 的构造方法中依赖 B，在 B 的构造方法中依赖 A 是不行的。

加载 Bean: loadBeanDefinitions ()

4.AbstractXmlApplicationContext .class

```
/** 我们可以看到，此方法将通过一个 XmlBeanDefinitionReader 实例来加载各个 Bean。*/
@Override
protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory)
throws BeansException, IOException {
   // 给这个 BeanFactory 实例化一个 XmlBeanDefinitionReader
   XmlBeanDefinitionReader beanDefinitionReader = new
XmlBeanDefinitionReader(beanFactory);

   // Configure the bean definition reader with this context's
   // resource loading environment.
   beanDefinitionReader.setEnvironment(this.getEnvironment());
   beanDefinitionReader.setResourceLoader(this);
   beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

   // 初始化 BeanDefinitionReader，其实这个是提供给子类覆写的，
   // 我看了一下，没有类覆写这个方法，我们姑且当做不重要吧
   initBeanDefinitionReader(beanDefinitionReader);
   // 重点来了，继续往下
   loadBeanDefinitions(beanDefinitionReader);
}


protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws
BeansException, IOException {
    Resource[] configResources = getConfigResources();
    if (configResources != null) {
        reader.loadBeanDefinitions(configResources);
    }
    String[] configLocations = getConfigLocations();
    if (configLocations != null) {
        reader.loadBeanDefinitions(configLocations);
    }
}
```

首先在refreshBeanFactory()方法中已经初始化了DefaultListableBeanFactory，对于读取XML配置文件，还需要使用XmlBeanDefinitionReader。所以在上述loadBeanDefinitions()中就需要初始化XmlBeanDefinitionReader。在DefaultListableBeanFactory和XmlBeanDefinitionReader后就可以进行配置文件的读取了。要注意的地方时，在XmlBeanDefinitionReader初始化时就已经把DefaultListableBeanFactory给注册进去了，所以在XmlBeanDefinitionReader读取的BeanDefinition都会注册到DefaultListableBeanFactory中，也就是经过上述的loadingBeanDefinitions()，类型DefaultListableBeanFactory的变量beanFactory就已经包含了所有解析好的配置了。

5.AbstractBeanDefinitionReader.class

```
@Override
public int loadBeanDefinitions(String... locations) throws
BeanDefinitionStoreException {
    Assert.notNull(locations, "Location array must not be null");
```

```
    int count = 0;
    for (String location : locations) {
        count += loadBeanDefinitions(location);
    }
    return count;
}

/*
从指定的资源位置加载bean定义,他的位置也可以是位置模式，前提是此bean定义读取器的
ResourceLoader是ResourcePatternResolver。
*/
public int loadBeanDefinitions(String location, @Nullable Set<Resource>
actualResources) throws BeanDefinitionStoreException {
    ResourceLoader resourceLoader = getResourceLoader();
    if (resourceLoader == null) {
        ...
    }

    if (resourceLoader instanceof ResourcePatternResolver) {
        // 资源模式匹配可用
        try {
            Resource[] resources = ((ResourcePatternResolver)
resourceLoader).getResources(location);
            int count = loadBeanDefinitions(resources);
            if (actualResources != null) {
                Collections.addAll(actualResources, resources);
            }
            if (logger.isTraceEnabled()) {
                logger.trace("Loaded " + count + " bean definitions from
location pattern [" + location + "]");
            }
            return count;
        }
        catch (IOException ex) {
            ...
        }
    }
    else {
        // 只能通过绝对URL加载单个资源
        Resource resource = resourceLoader.getResource(location);
        int count = loadBeanDefinitions(resource);
        if (actualResources != null) {
            actualResources.add(resource);
        }
        ...
    return count;
}
```

6.PathMatchingResourcePatternResolver.class

```
@Override
public Resource[] getResources(String locationPattern) throws IOException {
    Assert.notNull(locationPattern, "Location pattern must not be null");
    if (locationPattern.startsWith(CLASSPATH_ALL_URL_PREFIX)) {
        // 类路径资源（可能有多个同名资源）
```

```
        if
(getPathMatcher().isPattern(locationPattern.substring(CLASSPATH_ALL_URL_PREFIX.l
ength()))) {
            // 类路径资源模式
            return findPathMatchingResources(locationPattern);
        }
        else {
            // 具有给定名称的所有类路径资源
            return
findAllClassPathResources(locationPattern.substring(CLASSPATH_ALL_URL_PREFIX.len
gth()));
        }
    }
    else {
        // 通常只在前缀后面查找一个模式，并且在Tomcat之后只有“* /”分隔符之后的“war：”协议。
        int prefixEnd = (locationPattern.startsWith("war:") ?
locationPattern.indexOf("*/") + 1 :
                locationPattern.indexOf(':') + 1);
        if (getPathMatcher().isPattern(locationPattern.substring(prefixEnd))) {
            // 文件模式
            return findPathMatchingResources(locationPattern);
        }
        else {
            // 具有给定名称的单个资源
            return new Resource[]
{getResourceLoader().getResource(locationPattern)};
        }
    }
}
```

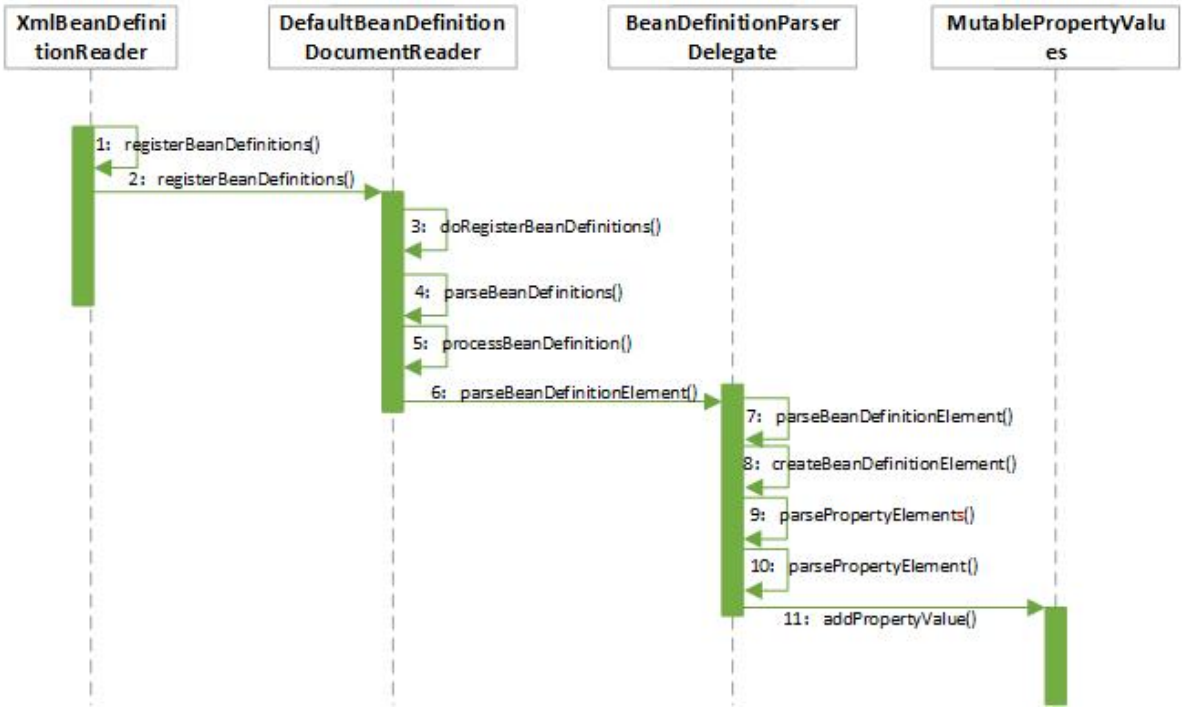
7.XmlBeanDefinitionReader.class

```
/*
    从XML配置文件中获取bean定义信息
*/
public int loadBeanDefinitions(EncodedResource encodedResource) throws
BeanDefinitionStoreException {
    Assert.notNull(encodedResource, "EncodedResource must not be null");
    ...
    Set<EncodedResource> currentResources =
this.resourcesCurrentlyBeingLoaded.get();
    if (currentResources == null) {
        currentResources = new HashSet<>(4);
        this.resourcesCurrentlyBeingLoaded.set(currentResources);
    }
    ...
    try {
        InputStream inputStream =
encodedResource.getResource().getInputStream();
        try {
            InputSource inputSource = new InputSource(inputStream);
            if (encodedResource.getEncoding() != null) {
                inputSource.setEncoding(encodedResource.getEncoding());
            }
            //获取到读取xml配置文件的InputStream流后，进行BeanDefinitions的加载
            return doLoadBeanDefinitions(inputSource,
encodedResource.getResource());
```

```
            }
            finally {
                inputStream.close();
            }
        }
        catch (IOException ex) {
            ...
        }
        finally {
            currentResources.remove(encodedResource);
            if (currentResources.isEmpty()) {
                this.resourcesCurrentlyBeingLoaded.remove();
            }
        }
    }

    /*
        真正从xml配置文件中加载Bean定义信息
    */
    protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
            throws BeanDefinitionStoreException {

        try {
            //获取xml的配置信息并封装为Document对象
            Document doc = doLoadDocument(inputSource, resource);
            return this.registerBeanDefinitions(doc, resource);
        }
        catch (BeanDefinitionStoreException ex) {
            ...
        }
    }
```

经过漫长的链路，一个配置文件终于转换为一颗 DOM 树了，注意，这里指的是其中一个配置文件，不是所有的，读者可以看到上面有个 for 循环的。下面从根节点开始解析

下面，继续深入registerBeanDefinitions方法。



```
/*
    注册给定DOM文档中包含的bean定义
*/
public int registerBeanDefinitions(Document doc, Resource resource) throws
BeanDefinitionStoreException {
    BeanDefinitionDocumentReader documentReader =
this.createBeanDefinitionDocumentReader();
    int countBefore = this.getRegistry().getBeanDefinitionCount();
    documentReader.registerBeanDefinitions(doc,
this.createReaderContext(resource));
    return this.getRegistry().getBeanDefinitionCount() - countBefore;
}
```

8.DefaultBeanDefinitionDocumentReader.class

```
/*
    此实现根据“spring-beans”XSD解析bean定义
*/
public void registerBeanDefinitions(Document doc, XmlReaderContext
readerContext) {
    this.readerContext = readerContext;
    this.logger.debug("Loading bean definitions");
    Element root = doc.getDocumentElement();
    this.doRegisterBeanDefinitions(root);
}

/*
    任何嵌套的<beans>元素都将导致此方法的递归。为了正确传播和保留beans的default属性，需要跟
踪当前（父）委托，该委托可以为null。创建新的（子）委托时，需要引用父项以进行回退，然后最终将
this.delegate重置为其原始（父）引用。
*/
protected void doRegisterBeanDefinitions(Element root) {
    BeanDefinitionParserDelegate parent = this.delegate;
    this.delegate = this.createDelegate(this.getReaderContext(), root, parent);
    if (this.delegate.isDefaultNamespace(root)) {
        String profileSpec = root.getAttribute("profile");
        if (StringUtils.hasText(profileSpec)) {
            String[] specifiedProfiles =
StringUtils.tokenizeToStringArray(profileSpec, ",; ");
            if
(!this.getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {
                if (this.logger.isInfoEnabled()) {
                    this.logger.info("Skipped XML bean definition file due to
specified profiles [" + profileSpec + "] not matching: " +
this.getReaderContext().getResource());
                }

                return;
            }
        }
    }

    this.preProcessXml(root);
    //解析beanDefinitions信息，经过这个方法，beanFactory中就会保存从xml配置文件中解析而来
的信息
    this.parseBeanDefinitions(root, this.delegate);
```

```
    this.postProcessXml(root);
    this.delegate = parent;
}

/*
    解析文档中根级别的元素：import、alias、bean
*/
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate
delegate) {
    if (delegate.isDefaultNamespace(root)) {
        NodeList nl = root.getChildNodes();
        for(int i = 0; i < nl.getLength(); ++i) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                Element ele = (Element)node;
                if (delegate.isDefaultNamespace(ele)) {
                    //解析默认元素
                    this.parseDefaultElement(ele, delegate);
                } else {
                    delegate.parseCustomElement(ele);
                }
            }
        }
    } else {
        delegate.parseCustomElement(root);
    }

}

private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate
delegate) {
    if (delegate.nodeNameEquals(ele, "import")) {
        this.importBeanDefinitionResource(ele);
    } else if (delegate.nodeNameEquals(ele, "alias")) {
        this.processAliasRegistration(ele);
    } else if (delegate.nodeNameEquals(ele, "bean")) {
        //读取到xml配置文件的<bean>节点
        this.processBeanDefinition(ele, delegate);
    } else if (delegate.nodeNameEquals(ele, "beans")) {
        this.doRegisterBeanDefinitions(ele);
    }
}


protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate
delegate) {
   // 将 <bean /> 节点转换为 BeanDefinitionHolder，就是上面说的一堆
   BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
   if (bdHolder != null) {
      // 如果有自定义属性的话，进行相应的解析，先忽略
      bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
      try {
         // 我们把这步叫做 注册Bean 吧
         BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
      }
      catch (BeanDefinitionStoreException ex) {
```

```
            getReaderContext().error("Failed to register bean definition with name
'" +
                    bdHolder.getBeanName() + "'", ele, ex);
        }
        // 注册完成后，发送事件，本文不展开说这个
        getReaderContext().fireComponentRegistered(new
BeanComponentDefinition(bdHolder));
    }
```

9.BeanDefinitionParserDelegate.class

```
/*
    解析bean定义本身，而不考虑名称或别名，如果解析期间出错则返回null。
*/
@Nullable
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, @Nullable
BeanDefinition containingBean) {
    String id = ele.getAttribute("id");
    String nameAttr = ele.getAttribute("name");
    List<String> aliases = new ArrayList();
    if (StringUtils.hasLength(nameAttr)) {
        String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr, ",; ");
        aliases.addAll(Arrays.asList(nameArr));
    }

    String beanName = id;
    if (!StringUtils.hasText(id) && !aliases.isEmpty()) {
        beanName = (String)aliases.remove(0);
        ...
    }

    if (containingBean == null) {
        this.checkNameUniqueness(beanName, aliases, ele);
    }

    //终于，这里要解析beanDefinition了
    AbstractBeanDefinition beanDefinition = this.parseBeanDefinitionElement(ele,
beanName, containingBean);
    if (beanDefinition != null) {
        if (!StringUtils.hasText(beanName)) {
            try {
                if (containingBean != null) {
                    beanName =
BeanDefinitionReaderUtils.generateBeanName(beanDefinition,
this.readerContext.getRegistry(), true);
                } else {
                    beanName =
this.readerContext.generateBeanName(beanDefinition);
                    String beanClassName = beanDefinition.getBeanClassName();
                    if (beanClassName != null &&
beanName.startsWith(beanClassName) && beanName.length() > beanClassName.length()
&& !this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
                        aliases.add(beanClassName);
                    }
                }
                ...
```

```
            } catch (Exception var9) {
                this.error(var9.getMessage(), ele);
                return null;
            }
        }

        //别名数组
        String[] aliasesArray = StringUtils.toStringArray(aliases);
        return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
    } else {
        return null;
    }
}

@Nullable
public AbstractBeanDefinition parseBeanDefinitionElement(Element ele, String
beanName, @Nullable BeanDefinition containingBean) {
    this.parseState.push(new BeanEntry(beanName));
    String className = null;
    if (ele.hasAttribute("class")) {
        className = ele.getAttribute("class").trim();
    }

    String parent = null;
    if (ele.hasAttribute("parent")) {
        parent = ele.getAttribute("parent");
    }

    try {
        //创建BeanDefinition
        AbstractBeanDefinition bd = this.createBeanDefinition(className,
parent);
        this.parseBeanDefinitionAttributes(ele, beanName, containingBean, bd);
        bd.setDescription(DomUtils.getChildElementValueByTagName(ele,
"description"));
        this.parseMetaElements(ele, bd);
        this.parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
        this.parseReplacedMethodSubElements(ele, bd.getMethodOverrides());

        //通过构造器解析参数值
        this.parseConstructorArgElements(ele, bd);
        //通过property的value解析值吗，本文的程序xml就是通过property属性设置bean的值的，
最终被这一方法所解析出来。
        this.parsePropertyElements(ele, bd);

        this.parseQualifierElements(ele, bd);
        bd.setResource(this.readerContext.getResource());
        bd.setSource(this.extractSource(ele));
        AbstractBeanDefinition var7 = bd;
        return var7;
    } catch (ClassNotFoundException var13) {
        ...
    }
    return null;
}


public void parsePropertyElements(Element beanEle, BeanDefinition bd) {
```

```
    NodeList nl = beanEle.getChildNodes();
    for(int i = 0; i < nl.getLength(); ++i) {
        Node node = nl.item(i);
        if (this.isCandidateElement(node) && this.nodeNameEquals(node,
"property")) {
            //解析出参数值来，这里就真正的属性值解析出来并防止在一个组装的类里面存放着。因为这
里有两个bean，所以要循环调用两次parsePropertyElement()方法
            this.parsePropertyElement((Element)node, bd);
        }
    }
}

public void parsePropertyElement(Element ele, BeanDefinition bd) {
    String propertyName = ele.getAttribute("name");
    if (!StringUtils.hasLength(propertyName)) {
        this.error("Tag 'property' must have a 'name' attribute", ele);
    } else {
        this.parseState.push(new PropertyEntry(propertyName));

        try {
            if (!bd.getPropertyValues().contains(propertyName)) {
                Object val = this.parsePropertyValue(ele, bd, propertyName);
                PropertyValue pv = new PropertyValue(propertyName, val);
                this.parseMetaElements(ele, pv);
                pv.setSource(this.extractSource(ele));

                //就是这一步，属性值注入
                bd.getPropertyValues().addPropertyValue(pv);
                return;
            }
        } finally {
            this.parseState.pop();
        }

    }
}
```

产生了一个 BeanDefinitionHolder 的实例，这个实例里面也就是一个 BeanDefinition 的实例和它的 beanName、aliases 这三个信息，注意，我们的关注点始终在 BeanDefinition 上：

```
public class BeanDefinitionHolder implements BeanMetadataElement {

  private final BeanDefinition beanDefinition;

  private final String beanName;

  private final String[] aliases;
  ...
}
```

注册 Bean

10.BeanDefinitionReaderUtils.class

```
public static void registerBeanDefinition(
      BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry)
      throws BeanDefinitionStoreException {

   String beanName = definitionHolder.getBeanName();
   // 注册这个 Bean
   registry.registerBeanDefinition(beanName,
definitionHolder.getBeanDefinition());

   // 如果还有别名的话，也要根据别名统统注册一遍，不然根据别名就找不到 Bean 了，这我们就不开心
了
   String[] aliases = definitionHolder.getAliases();
   if (aliases != null) {
      for (String alias : aliases) {
         // alias -> beanName 保存它们的别名信息，这个很简单，用一个 map 保存一下就可以
了，
         // 获取的时候，会先将 alias 转换为 beanName，然后再查找
         registry.registerAlias(beanName, alias);
      }
   }
```

9.DefaultListableBeanFactory .class

```
@Override
public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
      throws BeanDefinitionStoreException {

   Assert.hasText(beanName, "Bean name must not be empty");
   Assert.notNull(beanDefinition, "BeanDefinition must not be null");

   if (beanDefinition instanceof AbstractBeanDefinition) {
      try {
         ((AbstractBeanDefinition) beanDefinition).validate();
      }
      catch (BeanDefinitionValidationException ex) {
         throw new BeanDefinitionStoreException(...);
      }
   }

   // old? 还记得“允许 bean 覆盖”这个配置吗？allowBeanDefinitionOverriding
   BeanDefinition oldBeanDefinition;

   // 之后会看到，所有的 Bean 注册后会放入这个 beanDefinitionMap 中
   oldBeanDefinition = this.beanDefinitionMap.get(beanName);

   // 处理重复名称的 Bean 定义的情况
   if (oldBeanDefinition != null) {
      if (!isAllowBeanDefinitionOverriding()) {
         // 如果不允许覆盖的话，抛异常
         throw new
BeanDefinitionStoreException(beanDefinition.getResourceDescription()...
      }
      else if (oldBeanDefinition.getRole() < beanDefinition.getRole()) {
         // log...用框架定义的 Bean 覆盖用户自定义的 Bean
      }
```

```
      else if (!beanDefinition.equals(oldBeanDefinition)) {
         // log...用新的 Bean 覆盖旧的 Bean
      }
      else {
         // log...用同等的 Bean 覆盖旧的 Bean，这里指的是 equals 方法返回 true 的 Bean
      }
      // 覆盖
      this.beanDefinitionMap.put(beanName, beanDefinition);
   }
   else {
      // 判断是否已经有其他的 Bean 开始初始化了.
      // 注意，"注册Bean" 这个动作结束，Bean 依然还没有初始化，我们后面会有大篇幅说初始化过
程，
      // 在 Spring 容器启动的最后，会 预初始化 所有的 singleton beans
      if (hasBeanCreationStarted()) {
         // Cannot modify startup-time collection elements anymore (for stable
iteration)
         synchronized (this.beanDefinitionMap) {
            this.beanDefinitionMap.put(beanName, beanDefinition);
            List<String> updatedDefinitions = new ArrayList<String>
(this.beanDefinitionNames.size() + 1);
            updatedDefinitions.addAll(this.beanDefinitionNames);
            updatedDefinitions.add(beanName);
            this.beanDefinitionNames = updatedDefinitions;
            if (this.manualSingletonNames.contains(beanName)) {
               Set<String> updatedSingletons = new LinkedHashSet<String>
(this.manualSingletonNames);
               updatedSingletons.remove(beanName);
               this.manualSingletonNames = updatedSingletons;
            }
         }
      }
      else {
         // 最正常的应该是进到这里。

         // 将 BeanDefinition 放到这个 map 中，这个 map 保存了所有的 BeanDefinition
         this.beanDefinitionMap.put(beanName, beanDefinition);
         // 这是个 ArrayList，所以会按照 bean 配置的顺序保存每一个注册的 Bean 的名字
         this.beanDefinitionNames.add(beanName);
         // 这是个 LinkedHashSet，代表的是手动注册的 singleton bean，
         // 注意这里是 remove 方法，到这里的 Bean 当然不是手动注册的
         // 手动指的是通过调用以下方法注册的 bean ：
         //     registerSingleton(String beanName, Object singletonObject)
         //          这不是重点，解释只是为了不让大家疑惑。Spring 会在后面"手动"注册一些
Bean，如 "environment"、"systemProperties" 等 bean
         this.manualSingletonNames.remove(beanName);
      }
      // 这个不重要，在预初始化的时候会用到，不必管它。
      this.frozenBeanDefinitionNames = null;
   }

   if (oldBeanDefinition != null || containsSingleton(beanName)) {
      resetBeanDefinition(beanName);
   }
}
```

总结一下，到这里已经初始化了 Bean 容器, 配置也相应的转换为了一个个 BeanDefinition，然后注册了各个 BeanDefinition 到注册中心，并且发送了注册事件。

面的过程，就已经完成了Spring容器的初始化过程，相信读者也已经对Spring容器的初始化有了一个大致的了解。下面总结一下Spring容器的初始化：

- 第一个过程是Resource定位过程。这个Resource定位过程指的是BeanDefinition的资源定位，它由ResourceLoader通过统一的Resource接口来完成，这个Resource对各种形式的BeanDefinition的使用都提供了统一接口。这个定位过程类似于容器寻找数据的过程，就像使用水桶装水先要把水找到一样。
- 第二个过程是BeanDefinition的载入。这个载入过程是把用户定义好的Bean表示成IOC容器内部的数据结构，而这个容器内部的数据结构就是BeanDefinition。下面介绍这个数据结构的详细定义。具体来说，这个BeanDefinition实际上就是POJO对象在IOC容器的抽象，通过这个BeanDefinition定义的数据结构，使IOC能够方便地对POJO对象进行管理。
- 第三个过程是向IOC容器注册这些BeanDefinition的过程，这个过程是通过调用BeanDefinitionRegistry接口的实现来完成的。这个注册过程把载入过程中解析到的BeanDefinition向IOC容器进行注册。通过上面的分析，我们知道IOC内部将BeanDefinition注册到了ConcurrentHashMap中。

3）Bean容器实例化完成后

以上解析我们才说完refresh()中obtainFreshBeanFactory() 方法， 接下来就是要弄清楚Spring容器对bean的创建和加载了。

准备 Bean 容器: prepareBeanFactory

AbstractApplicationContext

```
/**
 * Configure the factory's standard context characteristics,
 * such as the context's ClassLoader and post-processors.
 * @param beanFactory the BeanFactory to configure
 */
protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
   // 设置 BeanFactory 的类加载器，我们知道 BeanFactory 需要加载类，也就需要类加载器，
   // 这里设置为当前 ApplicationContext 的类加载器
   beanFactory.setBeanClassLoader(getClassLoader());
   // 设置 BeanExpressionResolver
   beanFactory.setBeanExpressionResolver(new
StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
   //
   beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this,
getEnvironment()));

   // 添加一个 BeanPostProcessor，这个 processor 比较简单，
   // 实现了 Aware 接口的几个特殊的 beans 在初始化的时候，这个 processor 负责回调
   beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));

   // 下面几行的意思就是，如果某个 bean 依赖于以下几个接口的实现类，在自动装配的时候忽略它们，
   // Spring 会通过其他方式来处理这些依赖。
   beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
   beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
   beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);
   beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
   beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
   beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
```

```
    /**
     * 下面几行就是为特殊的几个 bean 赋值，如果有 bean 依赖了以下几个，会注入这边相应的值，
     * 之前我们说过，"当前 ApplicationContext 持有一个 BeanFactory"，这里解释了第一行
     * ApplicationContext 继承了 ResourceLoader、ApplicationEventPublisher、
MessageSource
     * 所以对于这几个，可以赋值为 this，注意 this 是一个 ApplicationContext
     * 那这里怎么没看到为 MessageSource 赋值呢？那是因为 MessageSource 被注册成为了一个普
通的 bean
     */
    beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);
    beanFactory.registerResolvableDependency(ResourceLoader.class, this);
    beanFactory.registerResolvableDependency(ApplicationEventPublisher.class,
this);
    beanFactory.registerResolvableDependency(ApplicationContext.class, this);

    // 这个 BeanPostProcessor 也很简单，在 bean 实例化后，如果是 ApplicationListener 的
子类，
    // 那么将其添加到 listener 列表中，可以理解成：注册事件监听器
    beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));

    // Detect a LoadTimeWeaver and prepare for weaving, if found.
    // 这里涉及到特殊的 bean，名为：loadTimeWeaver，这不是我们的重点，忽略它
    if (beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
        beanFactory.addBeanPostProcessor(new
LoadTimeWeaverAwareProcessor(beanFactory));
        // Set a temporary ClassLoader for type matching.
        beanFactory.setTempClassLoader(new
ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
    }

    /**
     * 从下面几行代码我们可以知道，Spring 往往很 "智能" 就是因为它会帮我们默认注册一些有用的
bean，
     * 我们也可以选择覆盖
     */

    // 如果没有定义 "environment" 这个 bean，那么 Spring 会 "手动" 注册一个
    if (!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
        beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME, getEnvironment());
    }
    // 如果没有定义 "systemProperties" 这个 bean，那么 Spring 会 "手动" 注册一个
    if (!beanFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {
        beanFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME,
getEnvironment().getSystemProperties());
    }
    // 如果没有定义 "systemEnvironment" 这个 bean，那么 Spring 会 "手动" 注册一个
    if (!beanFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAME)) {
        beanFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME,
getEnvironment().getSystemEnvironment());
    }
}
```

初始化所有的 singleton beans

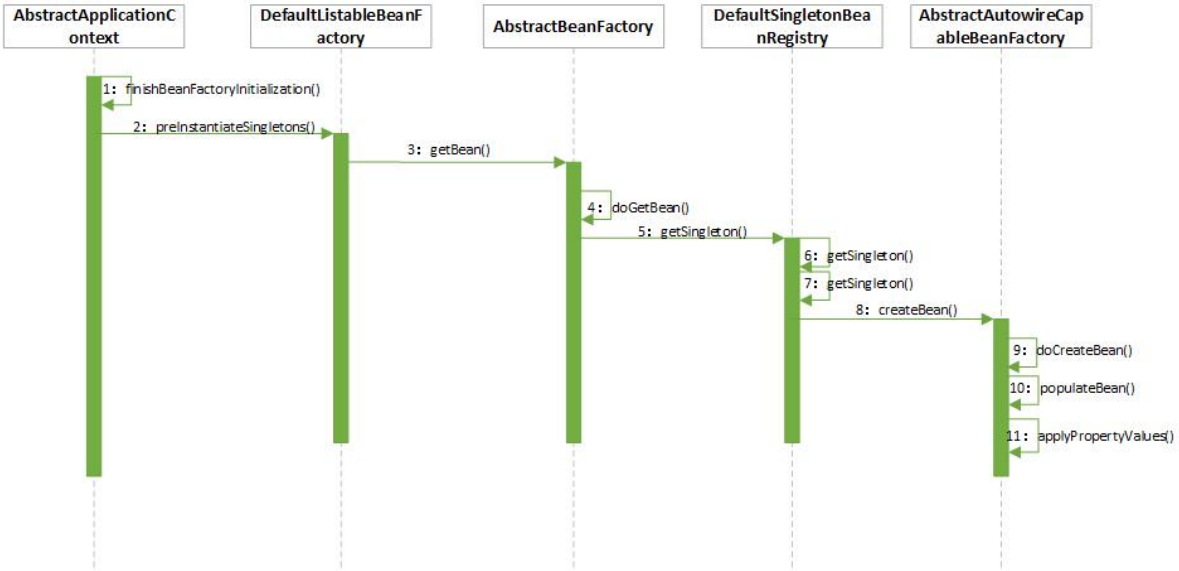
我们的重点当然是 finishBeanFactoryInitialization(beanFactory); 这个巨头了，这里会负责初始化所有的 singleton beans。我们来总结一下，到目前为止，应该说 BeanFactory 已经创建完成，并且所有的实现了 BeanFactoryPostProcessor 接口的 Bean 都已经初始化并且其中的 postProcessBeanFactory(factory) 方法已经得到执行了。所有实现了 BeanPostProcessor 接口的 Bean 也都完成了初始化。

剩下的就是初始化其他还没被初始化的 singleton beans 了，我们知道它们是单例的，如果没有设置懒加载，那么 Spring 会在接下来初始化所有的 singleton beans

下面先简单总结一下在IOC中singleton bean的生命周期：

- Bean实例的创建
- 为Bean实例设置属性
- 调用Bean的初始化方法
- 应用可以通过IOC容器使用Bean
- 当容器关闭时，调用Bean的销毁方法

先看看创建bean和初始化bean的时序图。



1.AbstractApplicationContext.class

```
// 初始化剩余的 singleton beans
protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory
beanFactory) {

   // 首先，初始化名字为 conversionService 的 Bean。
   // 什么，看代码这里没有初始化 Bean 啊！
   // 注意了，初始化的动作包装在 beanFactory.getBean(...) 中，这里先不说细节，先往下看吧
   if (beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&
         beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME,
ConversionService.class)) {
      beanFactory.setConversionService(
            beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME,
ConversionService.class));
   }

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

```
            @Override
            public String resolveStringValue(String strVal) {
                return getEnvironment().resolvePlaceholders(strVal);
            }
        });
    }

    // 先初始化 LoadTimeWeaverAware 类型的 Bean
    // 一般用于织入第三方模块，在 class 文件载入 JVM 的时候动态织入，这里不展开说
    String[] weaverAwareNames =
beanFactory.getBeanNamesForType(LoadTimeWeaverAware.class, false, false);
    for (String weaverAwareName : weaverAwareNames) {
        getBean(weaverAwareName);
    }

    // Stop using the temporary ClassLoader for type matching.
    beanFactory.setTempClassLoader(null);

    // 没什么别的目的，因为到这一步的时候，Spring 已经开始预初始化 singleton beans 了，
    // 肯定不希望这个时候还出现 bean 定义解析、加载、注册。
    beanFactory.freezeConfiguration();

    // 开始初始化剩下的
    beanFactory.preInstantiateSingletons();
}
```

2.DefaultListableBeanFactory.class

```
@Override
public void preInstantiateSingletons() throws BeansException {
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Pre-instantiating singletons in " + this);
    }

    List<String> beanNames = new ArrayList<String>(this.beanDefinitionNames);

    // 触发所有的非懒加载的 singleton beans 的初始化操作
    for (String beanName : beanNames) {

        // 合并父 Bean 中的配置，注意 <bean id="" class="" parent="" /> 中的 parent，用
的不多吧，
        // 考虑到这可能会影响大家的理解，我在附录中解释了一下 "Bean 继承"，请移步
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

        // 非抽象、非懒加载的 singletons。如果配置了 'abstract = true'，那是不需要初始化的
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
            // 处理 FactoryBean(读者如果不熟悉 FactoryBean，请移步附录区了解)
            if (isFactoryBean(beanName)) {
                // FactoryBean 的话，在 beanName 前面加上 ‘&’ 符号。再调用 getBean，
getBean 方法别急
                final FactoryBean<?> factory = (FactoryBean<?>)
getBean(FACTORY_BEAN_PREFIX + beanName);
                // 判断当前 FactoryBean 是否是 SmartFactoryBean 的实现，此处忽略，直接跳过
                boolean isEagerInit;
                if (System.getSecurityManager() != null && factory instanceof
SmartFactoryBean) {
```

```
                isEagerInit = AccessController.doPrivileged(new
PrivilegedAction<Boolean>() {
                   @Override
                   public Boolean run() {
                      return ((SmartFactoryBean<?>) factory).isEagerInit();
                   }
                }, getAccessControlContext());
             }
             else {
                isEagerInit = (factory instanceof SmartFactoryBean &&
                      ((SmartFactoryBean<?>) factory).isEagerInit());
             }
             if (isEagerInit) {

                getBean(beanName);
             }
          }
          else {
             // 对于普通的 Bean，只要调用 getBean(beanName) 这个方法就可以进行初始化了
             getBean(beanName);
          }
       }
    }


    // 到这里说明所有的非懒加载的 singleton beans 已经完成了初始化
    // 如果我们定义的 bean 是实现了 SmartInitializingSingleton 接口的，那么在这里得到回
调，忽略
    for (String beanName : beanNames) {
       Object singletonInstance = getSingleton(beanName);
       if (singletonInstance instanceof SmartInitializingSingleton) {
          final SmartInitializingSingleton smartSingleton =
(SmartInitializingSingleton) singletonInstance;
          if (System.getSecurityManager() != null) {
             AccessController.doPrivileged(new PrivilegedAction<Object>() {
                @Override
                public Object run() {
                   smartSingleton.afterSingletonsInstantiated();
                   return null;
                }
             }, getAccessControlContext());
          }
          else {
             smartSingleton.afterSingletonsInstantiated();
          }
       }
    }
}
```

接下来，我们就进入到 getBean(beanName) 方法了，这个方法我们经常用来从 BeanFactory 中获取一个 Bean，而初始化的过程也封装到了这个方法里。

3.AbstractBeanFactory.class

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

```
}

// 我们在剖析初始化 Bean 的过程，但是 getBean 方法我们经常是用来从容器中获取 Bean 用的，注意切换思路，
// 已经初始化过了就从容器中直接返回，否则就先初始化再返回
@SuppressWarnings("unchecked")
protected <T> T doGetBean(
      final String name, final Class<T> requiredType, final Object[] args,
boolean typeCheckOnly)
      throws BeansException {
   // 获取一个“正统的” beanName，处理两种情况，一个是前面说的 FactoryBean(前面带‘&’)，
   // 一个是别名问题，因为这个方法是 getBean，获取 Bean 用的，你要是传一个别名进来，是完全可以的
   final String beanName = transformedBeanName(name);

   // 注意跟着这个，这个是返回值
   Object bean;

   // 检查下是不是已经创建过了
   Object sharedInstance = getSingleton(beanName);

   // 这里说下 args 呗，虽然看上去一点不重要。前面我们一路进来的时候都是 getBean(beanName)，
   // 所以 args 其实是 null 的，但是如果 args 不为空的时候，那么意味着调用方不是希望获取 Bean，而是创建 Bean
   if (sharedInstance != null && args == null) {
      if (logger.isDebugEnabled()) {
         if (isSingletonCurrentlyInCreation(beanName)) {
            logger.debug("...");
         }
         else {
            logger.debug("Returning cached instance of singleton bean '" +
beanName + "'");
         }
      }
      // 下面这个方法：如果是普通 Bean 的话，直接返回 sharedInstance，
      // 如果是 FactoryBean 的话，返回它创建的那个实例对象
      // (FactoryBean 知识，读者若不清楚请移步附录)
      bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
   }

   else {
      if (isPrototypeCurrentlyInCreation(beanName)) {
         // 当前线程已经创建过了此 beanName 的 prototype 类型的 bean，那么抛异常
         throw new BeanCurrentlyInCreationException(beanName);
      }

      // 检查一下这个 BeanDefinition 在容器中是否存在
      BeanFactory parentBeanFactory = getParentBeanFactory();
      if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
         // 如果当前容器不存在这个 BeanDefinition，试试父容器中有没有
         String nameToLookup = originalBeanName(name);
         if (args != null) {
            // 返回父容器的查询结果
            return (T) parentBeanFactory.getBean(nameToLookup, args);
         }
         else {
            // No args -> delegate to standard getBean method.
```

```
            return parentBeanFactory.getBean(nameToLookup, requiredType);
         }
      }

      if (!typeCheckOnly) {
         // typeCheckOnly 为 false，将当前 beanName 放入一个 alreadyCreated 的 Set 集合中。
         markBeanAsCreated(beanName);
      }

      /*
       * 稍稍总结一下：
       * 到这里的话，要准备创建 Bean 了，对于 singleton 的 Bean 来说，容器中还没创建过此 Bean；
       * 对于 prototype 的 Bean 来说，本来就是要创建一个新的 Bean。
       */
      try {
         final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
         checkMergedBeanDefinition(mbd, beanName, args);

         // 先初始化依赖的所有 Bean，这个很好理解。
         // 注意，这里的依赖指的是 depends-on 中定义的依赖
         String[] dependsOn = mbd.getDependsOn();
         if (dependsOn != null) {
            for (String dep : dependsOn) {
               // 检查是不是有循环依赖，这里的循环依赖和我们前面说的循环依赖又不一样，这里肯定是不允许出现的，不然要乱套了，读者想一下就知道了
               if (isDependent(beanName, dep)) {
                  throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                        "Circular depends-on relationship between '" + beanName + "' and '" + dep + "'");
               }
               // 注册一下依赖关系
               registerDependentBean(dep, beanName);
               // 先初始化被依赖项
               getBean(dep);
            }
         }

         // 创建 singleton 的实例
         if (mbd.isSingleton()) {
            sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {
               @Override
               public Object getObject() throws BeansException {
                  try {
                     // 执行创建 Bean，详情后面再说
                     return createBean(beanName, mbd, args);
                  }
                  catch (BeansException ex) {
                     destroySingleton(beanName);
                     throw ex;
                  }
               }
            });
            bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
```

```
            }

            // 创建 prototype 的实例
            else if (mbd.isPrototype()) {
               // It's a prototype -> create a new instance.
               Object prototypeInstance = null;
               try {
                  beforePrototypeCreation(beanName);
                  // 执行创建 Bean
                  prototypeInstance = createBean(beanName, mbd, args);
               }
               finally {
                  afterPrototypeCreation(beanName);
               }
               bean = getObjectForBeanInstance(prototypeInstance, name, beanName,
mbd);
            }

            // 如果不是 singleton 和 prototype 的话，需要委托给相应的实现类来处理
            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 {
                           // 执行创建 Bean
                           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);
               }
            }
         }
         catch (BeansException ex) {
            cleanupAfterBeanCreationFailure(beanName);
            throw ex;
         }
```

```
    }

    // 最后，检查一下类型对不对，不对的话就抛异常，对的话就返回了
    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;
}
```

4.AbstractAutowireCapableBeanFactory.class

```
/**
 * Central method of this class: creates a bean instance,
 * populates the bean instance, applies post-processors, etc.
 * @see #doCreateBean
 **/
/**
*第三个参数 args数组代表创建实例需要的参数，不就是给构造方法用的参数，或者是工厂Bean的参数嘛，
不过要注意，*在我们的初始化阶段，args 是 null。
**/
@Override
protected Object createBean(String beanName, RootBeanDefinition mbd, Object[]
args) throws BeanCreationException {
    if (logger.isDebugEnabled()) {
        logger.debug("Creating instance of bean '" + beanName + "'");
    }
    RootBeanDefinition mbdToUse = mbd;

    // 确保 BeanDefinition 中的 Class 被加载
    Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
    if (resolvedClass != null && !mbd.hasBeanClass() && mbd.getBeanClassName() !=
null) {
        mbdToUse = new RootBeanDefinition(mbd);
        mbdToUse.setBeanClass(resolvedClass);
    }

    // 准备方法覆写，这里又涉及到一个概念：MethodOverrides，它来自于 bean 定义中的
<lookup-method />
    // 和 <replaced-method />，如果读者感兴趣，回到 bean 解析的地方看看对这两个标签的解析。
    // 我在附录中也对这两个标签的相关知识点进行了介绍，读者可以移步去看看
    try {
        mbdToUse.prepareMethodOverrides();
    }
    catch (BeanDefinitionValidationException ex) {
        throw new BeanDefinitionStoreException(mbdToUse.getResourceDescription(),
                beanName, "Validation of method overrides failed", ex);
```

```
    }

    try {
        // 让 BeanPostProcessor 在这一步有机会返回代理，而不是 bean 实例，
        // 要彻底了解清楚这个，需要去看 InstantiationAwareBeanPostProcessor 接口，这里就
不展开说了
        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);
    }
    // 重头戏，创建 bean
    Object beanInstance = doCreateBean(beanName, mbdToUse, args);
    if (logger.isDebugEnabled()) {
        logger.debug("Finished creating instance of bean '" + beanName + "'");
    }
    return beanInstance;
}


/**
 * Actually create the specified bean. Pre-creation processing has already
happened
 * at this point, e.g. checking {@code postProcessBeforeInstantiation}
callbacks.
 * <p>Differentiates between default bean instantiation, use of a
 * factory method, and autowiring a constructor.
 * @param beanName the name of the bean
 * @param mbd the merged bean definition for the bean
 * @param args explicit arguments to use for constructor or factory method
invocation
 * @return a new instance of the bean
 * @throws BeanCreationException if the bean could not be created
 * @see #instantiateBean
 * @see #instantiateUsingFactoryMethod
 * @see #autowireConstructor
 */
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);
    }
    if (instanceWrapper == null) {
        // 说明不是 FactoryBean，这里实例化 Bean，这里非常关键，细节之后再说
        instanceWrapper = createBeanInstance(beanName, mbd, args);
    }
    // 这个就是 Bean 里面的 我们定义的类 的实例，很多地方我描述成 "bean 实例"
    final Object bean = (instanceWrapper != null ?
instanceWrapper.getWrappedInstance() : null);
```

```
   // 类型
   Class<?> beanType = (instanceWrapper != null ?
instanceWrapper.getWrappedClass() : null);
   mbd.resolvedTargetType = beanType;

   // 建议跳过吧，涉及接口：MergedBeanDefinitionPostProcessor
   synchronized (mbd.postProcessingLock) {
      if (!mbd.postProcessed) {
         try {
            // MergedBeanDefinitionPostProcessor，这个我真不展开说了，直接跳过吧，很少
用的
            applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
         }
         catch (Throwable ex) {
            throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
                  "Post-processing of merged bean definition failed", ex);
         }
         mbd.postProcessed = true;
      }
   }

   // Eagerly cache singletons to be able to resolve circular references
   // even when triggered by lifecycle interfaces like BeanFactoryAware.
   // 下面这块代码是为了解决循环依赖的问题，以后有时间，我再对循环依赖这个问题进行解析吧
   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<Object>() {
         @Override
         public Object getObject() throws BeansException {
            return getEarlyBeanReference(beanName, mbd, bean);
         }
      });
   }

   // Initialize the bean instance.
   Object exposedObject = bean;
   try {
      // 这一步也是非常关键的，这一步负责属性装配，因为前面的实例只是实例化了，并没有设值，这里
就是设值
      populateBean(beanName, mbd, instanceWrapper);
      if (exposedObject != null) {
         // 还记得 init-method 吗？还有 InitializingBean 接口？还有
BeanPostProcessor 接口？
         // 这里就是处理 bean 初始化完成后的各种回调
         exposedObject = initializeBean(beanName, exposedObject, mbd);
      }
   }
   catch (Throwable ex) {
      if (ex instanceof BeanCreationException &&
beanName.equals(((BeanCreationException) ex).getBeanName())) {
         throw (BeanCreationException) ex;
```

```
      }
      else {
         throw new BeanCreationException(
               mbd.getResourceDescription(), beanName, "Initialization of bean
failed", ex);
      }
   }

   if (earlySingletonExposure) {
      //
      Object earlySingletonReference = getSingleton(beanName, false);
      if (earlySingletonReference != null) {
         if (exposedObject == bean) {
            exposedObject = earlySingletonReference;
         }
         else if (!this.allowRawInjectionDespiteWrapping &&
hasDependentBean(beanName)) {
            String[] dependentBeans = getDependentBeans(beanName);
            Set<String> actualDependentBeans = new LinkedHashSet<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 {
      registerDisposableBeanIfNecessary(beanName, bean, mbd);
   }
   catch (BeanDefinitionValidationException ex) {
      throw new BeanCreationException(
            mbd.getResourceDescription(), beanName, "Invalid destruction
signature", ex);
   }

   return exposedObject;
}
```

到这里，笔者已经分析完了 doCreateBean 方法，总的来说，我们已经说完了整个初始化流程。

接下来笔者从 doCreateBean 中的选两个重要的方法解析。一个是创建 Bean 实例的 createBeanInstance 方法，一个是依赖注入的 populateBean 方法。

createBeanInstance()方法：

```
protected BeanWrapper createBeanInstance(String beanName, RootBeanDefinition
mbd, Object[] args) {
   // 确保已经加载了此 class
   Class<?> beanClass = resolveBeanClass(mbd, beanName);

   // 校验一下这个类的访问权限
   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());
   }

   if (mbd.getFactoryMethodName() != null)  {
      // 采用工厂方法实例化，不熟悉这个概念的读者请看附录，注意，不是 FactoryBean
      return instantiateUsingFactoryMethod(beanName, mbd, args);
   }

   // 如果不是第一次创建，比如第二次创建 prototype bean。
   // 这种情况下，我们可以从第一次创建知道，采用无参构造函数，还是构造函数依赖注入 来完成实例化
   boolean resolved = false;
   boolean autowireNecessary = false;
   if (args == null) {
      synchronized (mbd.constructorArgumentLock) {
         if (mbd.resolvedConstructorOrFactoryMethod != null) {
            resolved = true;
            autowireNecessary = mbd.constructorArgumentsResolved;
         }
      }
   }
   if (resolved) {
      if (autowireNecessary) {
         // 构造函数依赖注入
         return autowireConstructor(beanName, mbd, null, null);
      }
      else {
         // 无参构造函数
         return instantiateBean(beanName, mbd);
      }
   }

   // 判断是否采用有参构造函数
   Constructor<?>[] ctors =
determineConstructorsFromBeanPostProcessors(beanClass, beanName);
   if (ctors != null ||
         mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_CONSTRUCTOR ||
         mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args))  {
      // 构造函数依赖注入
      return autowireConstructor(beanName, mbd, ctors, args);
   }
```

```
    // 调用无参构造函数
    return instantiateBean(beanName, mbd);
}
```