# 设计模式

目录

[设计模式 1](#_Toc27660)

[第一章 概述 6](#_Toc30866)

[1.1. 设计模式的六大原则 6](#_Toc30253)

[1.1.1. 开闭原则（Open Close Principle） 6](#_Toc8708)

[1.1.2. 里氏代换原则（Liskov Substitution Principle） 6](#_Toc17914)

[1.1.3. 依赖倒转原则（Dependence Inversion Principle） 6](#_Toc25428)

[1.1.4. 接口隔离原则（Interface Segregation Principle） 7](#_Toc12419)

[1.1.5. 迪米特法则（最少知道原则）（Demeter Principle） 7](#_Toc22126)

[1.1.6. 合成复用原则（Composite Reuse Principle） 7](#_Toc1208)

[1.2. 泛化、实现、依赖、关联、聚合、组合 7](#_Toc109)

[1.2.1. 泛化Generalization【extends is-a】 7](#_Toc1414)

[1.2.2. 实现Realization【implements is-a】 8](#_Toc10686)

[1.2.3. 依赖Dependency【use】 8](#_Toc3579)

[1.2.4. 关联Association【has】 9](#_Toc27722)

[1.2.5. 聚合Aggregation【has-a】 9](#_Toc30526)

[1.2.6. 组合Composition【contains-a】 10](#_Toc20463)

[1.2.7. 总结 11](#_Toc22437)

[1.3. 设计模式的分类 12](#_Toc20439)

[第二章 单例模式 13](#_Toc11061)

[2.1. 简介 13](#_Toc14187)

[2.2. 懒汉式单例 13](#_Toc14662)

[2.2.1. 线程不安全的懒汉式单例 13](#_Toc31730)

[2.2.2. 在getInstance方法上加同步 14](#_Toc26660)

[2.2.3. 双重校验锁 15](#_Toc8224)

[2.2.4. 静态内部类 15](#_Toc30435)

[2.2.5. 总结 16](#_Toc22816)

[2.3. 饿汉式单例 16](#_Toc5835)

[2.4. 枚举 17](#_Toc10455)

[2.5. 总结 17](#_Toc26871)

[第三章 工厂模式 18](#_Toc32638)

[3.1. 简介 18](#_Toc18977)

[3.2. 工厂方法模式 18](#_Toc18113)

[3.3. 抽象工厂模式 18](#_Toc26040)

[3.4. Spring中的工厂模式 19](#_Toc2811)

[第四章 建造者模式 21](#_Toc4781)

[第五章 原型模式 22](#_Toc23379)

[第六章 代理模式 23](#_Toc20379)

[6.1. 简介 23](#_Toc32709)

[6.2. 静态代理 23](#_Toc8500)

[6.3. 动态代理 24](#_Toc23636)

[6.3.1. JDK动态代理 24](#_Toc27930)

[6.3.2. CGLIB动态代理 26](#_Toc21249)

[第七章 装饰者模式 29](#_Toc31892)

[7.1. 简介 29](#_Toc16949)

[7.2. 装饰器模式的应用场景 30](#_Toc28085)

[第八章 适配器模式 31](#_Toc27583)

[8.1. 简介 31](#_Toc20208)

[8.2. 类的适配 32](#_Toc12448)

[8.3. 对象的适配 33](#_Toc16439)

[8.4. 接口的适配 34](#_Toc12333)

[8.5. 三种适配的区别 35](#_Toc3597)

[第九章 外观模式 35](#_Toc3411)

[第十章 桥接模式 36](#_Toc7738)

[第十一章 组合模式 36](#_Toc29450)

[第十二章 享元模式 36](#_Toc764)

[第十三章 策略模式 36](#_Toc31023)

[第十四章 管道模式 36](#_Toc28909)

[第十五章 模板方法 36](#_Toc21854)

[第十六章 观察者模式 36](#_Toc29946)

[16.1. 简介 36](#_Toc24460)

[16.2. 针对的问题 36](#_Toc29461)

[16.3. 角色组成 37](#_Toc28777)

[16.4. 举例说明 38](#_Toc31636)

[16.5. 优点 42](#_Toc4048)

[16.6. 缺点 43](#_Toc13394)

[第十七章 迭代子模式 44](#_Toc23513)

[第十八章 责任链模式 44](#_Toc15375)

[第十九章 命令模式 44](#_Toc5440)

[第二十章 备忘录模式 44](#_Toc24996)

[第二十一章 状态模式 44](#_Toc25664)

[第二十二章 访问者模式 44](#_Toc5638)

[第二十三章 中介者模式 44](#_Toc16662)

[第二十四章 解释器模式 44](#_Toc30509)

[第二十五章 并发型模式 44](#_Toc1476)

[第二十六章 线程池模式 44](#_Toc853)

[第二十七章 Spring中的设计模式 44](#_Toc6867)

[27.1. 单例模式 44](#_Toc16185)

[27.2. 工厂模式 48](#_Toc32537)

[27.3. 适配器模式 48](#_Toc12086)

[27.4. 代理模式 51](#_Toc10966)

[27.5. 装饰者模式 51](#_Toc17019)

[27.6. 策略模式 52](#_Toc4268)

[27.6.1. 定义 52](#_Toc2218)

[27.6.2. Spring中策略模式的应用 52](#_Toc21294)

[27.7. 观察者模式 54](#_Toc3481)

[27.7.1. ApplicationEvent 54](#_Toc15393)

[27.7.2. ApplicationListener 55](#_Toc21531)

[27.7.3. ApplicationContext 55](#_Toc10925)

[27.7.4. ApplicationEventMulticaster 57](#_Toc28652)

[27.7.5. 举例说明 60](#_Toc6144)

[第二十八章 总结 62](#_Toc63)

[28.1. 装饰者模式和代理模式区别 62](#_Toc3645)

[28.2. 适配器、装饰者、代理模式的区别 63](#_Toc6017)

[第二十九章 Java基础知识 64](#_Toc2987)

[29.1. JavaWeb监听器 64](#_Toc16959)

[29.1.1. JavaWeb监听器概述 64](#_Toc23671)

[29.1.2. 创建与销毁对象的监听器--生命周期监听 64](#_Toc11875)

[29.1.3. 操作域属性的监听器--属性监听 65](#_Toc1940)

[29.1.4. HttpSession的监听器--感知监听 67](#_Toc15641)

[第三十章 Spring基础知识 70](#_Toc3954)

[30.1. 介绍 70](#_Toc23836)

[30.2. Spring多个配置文件组合方法 71](#_Toc3500)

[30.3. Spring配置文件bean的配置规则 71](#_Toc14403)

[30.4. Spring依赖注入3种方式 72](#_Toc10457)

[30.5. Spring通过注入方式注入依赖bean的方法 73](#_Toc22137)

[30.6. Spring的集合注入 73](#_Toc7711)

[30.7. Java注解(Annotation)简单介绍 74](#_Toc10641)

[30.8. 基于注解的Spring配置准备条件 76](#_Toc9281)

[30.9. 基于注解的spring配置 76](#_Toc20622)

[30.10. 额外的初始化和销毁 76](#_Toc23614)

[30.11. 使用 @Required 进行 Bean 的依赖检查 77](#_Toc5720)

[30.12. Spring的自动装配 78](#_Toc25529)

[30.12.1. 使用 @Resource、@Autowired 和 @Qualifier 指定 Bean 的自动装配策略 79](#_Toc3834)

[30.12.2. 使用@Autowired和@Qualifier注解执行自动装配 80](#_Toc32700)

[30.12.3. 使用@Resource和@Qualifier注解执行自动装配 82](#_Toc4052)

[30.13. Spring的自动扫描 82](#_Toc19106)

[30.14. 使用@Configuration和@Bean进行Bean的声明 83](#_Toc25196)

[30.15. Spring中的几个点 85](#_Toc26142)

[30.15.1. id和name 85](#_Toc20167)

[30.15.2. 配置是否允许 Bean 覆盖、是否允许循环依赖 86](#_Toc31425)

[30.15.3. profile 87](#_Toc16742)

[30.15.4. 工厂模式生成Bean 88](#_Toc22637)

[30.15.5. FactoryBean 89](#_Toc12899)

[30.15.6. 初始化Bean的回调 91](#_Toc26514)

[30.15.7. 销毁Bean的回调 92](#_Toc24129)

[30.15.8. ConversionService 93](#_Toc22830)

[30.15.9. Bean继承 94](#_Toc9834)

[30.15.10. 方法注入 95](#_Toc13279)

[30.15.11. BeanPostProcessor 98](#_Toc24178)

[第三十一章 Spring源码学习 99](#_Toc3732)

[31.1. Bean加载流程 99](#_Toc13460)

[31.2. 入口 99](#_Toc3474)

[31.3. ClassPathXmlApplicationContext 101](#_Toc5899)

[31.4. refresh方法 103](#_Toc7411)

[31.4.1. 创建Bean容器前的准备工作：prepareRefresh方法 104](#_Toc17713)

[31.4.2. 创建Bean容器，加载并注册Bean ：obtainFreshBeanFactory方法 105](#_Toc31688)

[31.4.3. <bean>中不定义id及id重复场景Spring处理方式 128](#_Toc29317)

[31.4.4. 准备Bean容器：prepareBeanFactory方法 131](#_Toc1400)

[31.4.5. postProcessBeanFactory方法 133](#_Toc21263)

[31.4.6. invokeBeanFactoryPostProcessors方法 134](#_Toc21286)

[31.4.7. registerBeanPostProcessors方法 134](#_Toc25532)

[31.4.8. initMessageSource方法 134](#_Toc23832)

[31.4.9. initApplicationEventMulticaster方法 134](#_Toc5790)

[31.4.10. onRefresh方法 134](#_Toc17520)

[31.4.11. registerListeners方法 134](#_Toc2619)

[31.4.12. 初始化所有的singleton beans：finishBeanFactoryInitialization方法 134](#_Toc28614)

[31.4.13. finishRefresh方法 155](#_Toc27730)

# 概述

## 设计模式的六大原则

### 开闭原则（Open Close Principle）

开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

### 里氏代换原则（Liskov Substitution Principle）

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

### 依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。

按理说，高层组件应该依赖于低层组件，低层组件为高层组件提供一些最基础的服务，但是工厂模式倒置了这一依赖现象，让低层组件反而要依赖于统一的抽象接口。

工厂模式让高层组件（WebCrawler）和低层组件（ATypeClient|BTypeClient|……）都依赖于共同的接口（HttpClient），这倒置了原本的依赖模型，解除了高层组件和低层组件之间的强依赖关系。

### 接口隔离原则（Interface Segregation Principle）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

### 迪米特法则（最少知道原则）（Demeter Principle）

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

### 合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。

## 泛化、实现、依赖、关联、聚合、组合

### 泛化Generalization【extends is-a】

泛化是一种继承关系，如果一个类A的所有属性和操作能被另一个类B所继承，则类B不仅可以包含自己独有的属性，而且可以包含类A的属性和操作。继承是类与类或者类与接口之间最常见的关系。

父类 父类实例=new 子类()；

泛化中子类直接继承了父类的属性和操作所以它的关系最强。

class Animal{}     
class Tiger extends Animal{}     
public class Test     
{     
    public void test()     
     {     
         Animal a=new Tiger();     
     }     
}

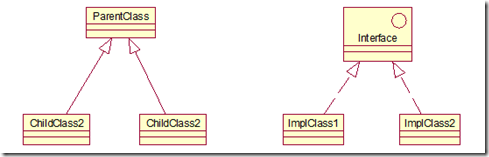
指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力，继承是类与类或者接口与接口之间最常见的关系。

UML中用带空心箭头的实线表示，箭头指向一般个体（父类）。

### 实现****Realization【implements is-a】****

实现关系和泛化是同等的强度。

指的是一个class类实现interface接口（可以是多个）的功能；实现是类与接口之间最常见的关系。



UML中用空心箭头和虚线表示，箭头指向定义约定的元素(一般指接口)。

### 依赖Dependency【use】

对于两个相对独立的对象，当一个对象负责构造另一个对象的实例时，或者依赖另一个对象的服务时，这两个对象之间主要体现为依赖关系

  与关联关系不同的是，依赖关系是以参数变量的形式传入到依赖类中。

  依赖是一种弱关联，只要一个类用到另一个类，但是和另一个类的关系不太明显的时候（可以说是“use”了那个类），可以把这种关系看成是依赖。

IMG_256

依赖是单向的。

    依赖关系表现在局部变量，方法的参数，以及对静态方法的调用。

public class Person{     
    /\*\* 拧螺丝 \*/    
    public void screw(Screwdriver screwdriver){     
         screwdriver.screw();     
     }     
}

UML中用带箭头的虚线表示，箭头指向被依赖元素。

### 关联Association【has】

对于两个相对独立的对象，当一个对象的实例与另一个对象的一些特定实例存在固定的对应关系时，这两个对象之间为关联关系。

IMG_256

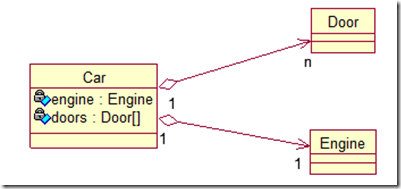
关联关系时使用实例变量来实现的。

public class Company{     
    private Employee employee;     
    public Employee getEmployee(){     
        return employee;     
     }     
    public void setEmployee(Employee employee){     
        this.employee=employee;     
     }     
    //公司运作     
    public void run(){     
         employee.startWorking();     
     }     
}

UML中用实线表示Association关系，箭头指向被依赖元素。

### 聚合Aggregation【has-a】

当对象A被加入到对象B中，成为对象B的组成部分时，对象B和A之间为聚集关系。聚合是关联关系的一种，是较强的关联关系，强调整体与部分之间的关系。



聚合关系时使用实例变量来实现的。

public class Computer{     
    private CPU cpu;     
    public CPU getCPU(){     
        return cpu;     
     }     
    public void setCPU(CPU cpu){     
        this.cpu=cpu;     
     }     
    //开启电脑     
    public void start(){     
        //cpu运作     
         cpu.run();     
     }     
}

UML中用带空心菱形头的实线表示，菱形头指向整体。

### 组合Composition【contains-a】

  组合关系也是聚合关系的一种，是比聚合关系更强的关系。组合关系是不能共享的。例如人有四肢、头等

   表示类之间整体和部分的关系，组合中部分和整体具有统一的生存周期。一旦整体对象不存在，部分对象也将不存在。部分对象和整体对象之间具有共生死的感觉。

IMG_256

组合关系时使用实例变量来实现的。

class Leg{};

class Arm{};

class Person

{

Leg mLeg;

Arm mArm;

}

UML中用带实心菱形头的实线表示Composition关系，菱形头指向整体。

### 总结

 实现关系和泛化是同等的强度。

#### 关联和依赖的区别

（1）关联关系中，体现的是两个类、或者类与接口之间语义级别的一种强依赖关系，比如我和我的朋友；这种关系比依赖更强、不存在依赖关系的偶然性、关系也不是临时性的，一般是长期性的，而且双方的关系一般是平等的。

（2）依赖关系中，可以简单的理解，就是一个类A使用到了另一个类B，而这种使用关系是具有偶然性的、临时性的、非常弱的，但是B类的变化会影响到A。

#### 关联与聚合的区别

   在语法上是难以区分的，在语义上才能更好的区分两者的区别。

    区别：关联关系所涉及的两个对象是处在同一个层次上的。比如人和自行车就是一种关联关系，而不是聚合关系，因为人不是自行车的组成部分。

         聚合关系涉及的两个对象处于不平等的层次上，一个代表整体，一个代表部分。比如：电脑和它的显示器、键盘、主板和内存就是聚集关系。

#### 聚合和组合的区别

1. 聚合和组合都是一种结合关系，只是额外具有整体部分的含义
2. 部件的生命周期不同

     聚合关系中，整体不会拥有部件的生命周期，所以整体删除时，部件不会被删除。再者，多个整体可以共享同一个部件。

     组合关系中，整体拥有部分的生命周期，所以整体删除时，部件一定会跟着删除。而且，多个整体不可以同时间共享一个部件。

1. 聚合关系是“has-a”关系，组合关系是“contain-a”关系

所以它们的关系是：组合>聚合>关联。

综上所述它们之间的耦合度是：泛化=实现>组合>聚合>关联>依赖。

## 设计模式的分类

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

其实还有两类：并发型模式和线程池模式。

**要想正确理解设计模式，首先必须明确它是为了解决什么问题而提出来的。**

# 单例模式

## 简介

单例模式具有以下特点：   
1、单例类只能有一个实例。   
2、单例类必须自己创建自己的唯一实例。   
3、单例类必须给所有其他对象提供这一实例。

单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。

单例模式的写法有好几种，这里主要介绍三种：懒汉式单例、饿汉式单例、静态内部类、枚举和双重校验锁。

懒汉式单例模式：在类加载时不初始化。

饿汉式单例模式：在类加载时就完成了初始化，所以类加载比较慢，但获取对象的速度快。

## 懒汉式单例

### 线程不安全的懒汉式单例

//*懒汉式单例类.在第一次调用的时候实例化自己*public class Singleton {  
 private Singleton() {  
 }  
 private static Singleton *singleton*;  
 //静态工厂方法  
 public static Singleton getInstance() {  
 if (*singleton* == null) {  
 *singleton* = new Singleton();  
 }  
 return *singleton*;  
 }  
}

Singleton通过将构造方法限定为private避免了类在外部被实例化，在同一个虚拟机范围内，Singleton的唯一实例只能通过getInstance()方法访问。

（事实上，通过Java反射机制是能够实例化构造方法为private的类的，那基本上会使所有的Java单例实现失效。此问题在此处不做讨论。）

但是以上懒汉式单例的实现没有考虑线程安全问题，它是线程不安全的，并发环境下很可能出现多个Singleton实例，要实现线程安全，有以下三种方式，都是对getInstance这个方法改造，保证了懒汉式单例的线程安全。

特点：延时加载；线程不安全，多线程下不能正常工作。

### 在getInstance方法上加同步

//静态工厂方法  
public static synchronized Singleton getInstance() {  
 if (*singleton* == null) {  
 *singleton* = new Singleton();  
 }  
 return *singleton*;  
}

这种写法在getInstance()方法中加入了synchronized锁。能够在多线程中很好的工作，而且看起来它也具备很好的lazy loading，但是效率很低（因为锁），并且大多数情况下不需要同步。

### 双重校验锁

public static Singleton getInstance() {  
 if (*singleton* == null) {  
 synchronized (Singleton.class) {  
 if (*singleton* == null) {  
 *singleton* = new Singleton();  
 }  
 }  
 }  
 return *singleton*;  
}

### 静态内部类

public class Singleton {

*/\*\*  
 \* 类级的内部类，也就是静态的成员式内部类，该内部类的实例与外部类的实例没有绑定关系，而且只有被调用到时才会装载(装在过程是由jvm保证线程安全)  
 \* ，从而实现了延迟加载  
 \*/*  
 private static class LazyHolder {

*/\*\*  
 \* 静态初始化器，由JVM来保证线程安全  
 \*/*  
 private static final Singleton *INSTANCE* = new Singleton();  
 }  
 private Singleton (){}

*/\*\*  
 这个模式的优势在于：getInstance方法并没有被同步，并且只是执行一个域的访问，因此延迟初始化并没有增加任何访问成本   
 \*/*  
 public static final Singleton getInstance() {  
 return LazyHolder.*INSTANCE*;  
 }  
}

这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，它跟饿汉式单例不同的是（很细微的差别）：饿汉式单例是只要Singleton类被装载了，那么instance就会被实例化（没有达到lazy loading效果），而这种方式是Singleton类被装载了，instance不一定被初始化。因为LazyHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载LazyHolder类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载，另外一方面，我不希望在Singleton类加载时就实例化，因为我不能确保Singleton类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化instance显然是不合适的。这个时候，这种方式相比饿汉式方法就显得更合理。

### 总结

第2.2.2种：加synchronized锁，在方法调用上加了同步，虽然线程安全了，但是每次都要同步，会影响性能，毕竟99%的情况下是不需要同步的，

第2.2.3种：双重校验锁，在getInstance中做了两次null检查，确保了只有第一次调用单例的时候才会做同步，这样也是线程安全的，同时避免了每次都同步的性能损耗。不过不推荐这种方式。

第2.2.4种：静态内部类，利用了classloader的机制来保证初始化instance时只有一个线程，所以也是线程安全的，同时没有性能损耗，所以一般我倾向于使用这一种。

## 饿汉式单例

public class Singleton {  
 private static Singleton *instance* = new Singleton();  
 private Singleton(){}  
 public static Singleton getInstance(){  
 return *instance*;  
 }  
}

这种方式基于classloder机制避免了多线程的同步问题，不过，instance在类装载时就实例化，这时候初始化instance显然没有达到lazy loading的效果。

以下为饿汉式单例的变种，和上面的一样，都是在类装载时实例化。

饿汉式单例天生就是线程安全的。

特点：可以通过反射机制攻击；线程安全（多个类加载器除外）。

private static Singleton *instance* = null;  
static {  
 *instance* = new Singleton();  
}

## 枚举

public enum Singleton implements Serializable{  
 *INSTANCE*,  
 *INSTANCE2*}

Singleton singleton=Singleton.*INSTANCE*;

这种方式是Effective Java作者Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象。

枚举实现线程安全的单例模式特点：

JVM会保证enum不能被反射并且构造器方法只执行一次。

## 总结

饿汉式在类创建的同时就实例化一个静态对象出来，不管之后会不会使用这个单例，都会占据一定的内存，但是相应的，在第一次调用时速度也会更快，因为其资源已经初始化完成，

而懒汉式顾名思义，会延迟加载，在第一次使用该单例的时候才会实例化对象出来，第一次调用时要做初始化，如果要做的工作比较多，性能上会有些延迟，之后就和饿汉式一样了。

什么是线程安全？

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。

# 工厂模式

## 简介

**工厂模式：父类定义了创建对象的接口，但是由子类来具体实现，工厂方法让类把实例化的动作推迟到了子类当中。**

也就是说，父类知道什么时候该去创建这个对象，也知道拿到这个对象之后应该对这个对象做什么事情，但是不知道如何去创建这个对象，对象的创建由子类来完成。

## 工厂方法模式

**工厂模式无法解决产品族和产品等级结构的问题。**

public interface IFoodFactory {  
 IFood produceFood();  
}

public class NoodleFactory implements IFoodFactory {  
 @Override  
 public IFood produceFood() {  
 System.*out*.println("---面条厂---");  
 return new Noodle();  
 }  
}

## 抽象工厂模式

抽象工厂模式中，一个工厂生产多个产品，它们是一个产品族，不同的产品族的产品派生于不同的抽象产品（或产品接口）

**工厂模式中，一个工厂生产一个产品，所有产品派生于同一个抽象产品（或产品接口）；而抽象工厂模式中，一个工厂生产多个产品，它们是一个产品族，不同的产品族的产品派生于不同的抽象产品（或产品接口）。**

public interface IDiningRoomFactory {  
 IFood produceFood();  
 IDrink produceDrink();  
}

public class DiningRoomFactoryA implements IDiningRoomFactory {  
 @Override  
 public IFood produceFood() {  
 System.*out*.println("---食堂A---");  
 return new Noodle();  
 }  
  
 @Override  
 public IDrink produceDrink() {  
 System.*out*.println("---食堂A---");  
 return new Water();  
 }  
}

## Spring中的工厂模式

// AbstractFactoryBean.java  
// 继承了FactoryBean，工厂Bean的主要作用是为了实现getObject()返回Bean实例  
public abstract class AbstractFactoryBean<T> implements FactoryBean<T>, BeanClassLoaderAware, BeanFactoryAware, InitializingBean, DisposableBean {  
  
 // 定义了获取对象的前置判断工作，创建对象的工作则交给了一个抽象方法  
// 这里判断了Bean是不是单例并且是否已经被加载过了（未初始化但加载过了，这个问题涉及到Spring处理循环依赖，以后会讨论到）  
 public final T getObject() throws Exception {  
 return this.isSingleton()?(this.initialized?this.singletonInstance:this.getEarlySingletonInstance()):this.createInstance();  
 }  
 // 由子类负责具体创建对象  
 protected abstract T createInstance() throws Exception;  
}

之所以这么写是因为这种写法带来了两个好处:

（1） 保证了创建Bean的方式的多样性   
 Bean工厂有很多种，它们负责创建各种各样不同的Bean，比如Map类型的Bean，List类型的Bean，Web服务Bean，子类们不需要关心单例或非单例情况下是否需要额外操作，只需要关心如何创建Bean，并且创建出来的Bean是多种多样的。

（2） 严格规定了Bean创建前后的其它动作   
 虽然子类可以自由的去创建Bean，但是创建Bean之前的准备工作以及创建Bean之后对Bean的处理工作是AbstractFactoryBean设定好了的，子类不需要关心，也没权力关心，在这个例子中父类只负责一些前置判断工作。

工厂方法模式非常的有趣，它给了子类创建实例的自由，又严格的规定了实例创建前后的业务流程。

# 建造者模式

# 原型模式

# 代理模式

## 简介

代理模式是常用的java设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。   
　　按照代理的创建时期，代理类可以分为两种。   
　　静态代理：由程序员创建或特定工具自动生成源代码，再对其编译。在程序运行前，代理类的.class文件就已经存在了。   
　　动态代理：在程序运行时，运用反射机制动态创建而成。

　　为什么使用动态代理？因为动态代理可以对请求进行任何处理。  
　　哪些地方需要动态代理？不允许直接访问某些类；对访问要做特殊处理等。

public interface IHello {  
 void sayHello(String name);  
}

public class Hello implements IHello {  
 @Override  
 public void sayHello(String name) {  
 System.*out*.println("你好："+name);  
 }  
}

## 静态代理

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-04-25 16:25  
 \* 与核心业务有关的辅助性操作  
 \*/*public class StaticProxy implements IHello {  
 private Logger logger = LoggerFactory.*getLogger*(this.getClass());  
 private IHello iHello;  
  
 public StaticProxy(IHello iHello){  
 this.iHello=iHello;  
 }  
  
 @Override  
 public void sayHello(String name) {  
 logger.info(System.*currentTimeMillis*()+"：开始！");  
 iHello.sayHello(name);  
 logger.info(System.*currentTimeMillis*()+"：结束！");  
 }  
}

## 动态代理

### JDK动态代理

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-04-25 16:24  
 \* \* 只要你是采用面向接口编程,那么,你的任何对象的方法执行之前要加上记录日志的操作都是可以的.  
 \* 他(DynaPoxyHello)自动去代理执行被代理对象(Hello)中的每一个方法,  
 \* 一个java.lang.reflect.InvocationHandler接口就把我们的代理对象和被代理对象解藕了.  
 \*  
 \*InvocationHandler是JDK动态代理的核心，生成的代理对象的方法调用都会委托到InvocationHandler.invoke()方法  
 \* JdkDynamicAopProxy是通过接口实现动态代理类，主要方法是getProxy(ClassLoader classLoader)，代理类生成之后再调用目标方法时就会调用invoke方法。  
 \*  
 \* 实现InvocationHandler接口，该接口定义了一个 invoke(Object proxy, Method method, Object[] args)的方法，  
 \* 1,proxy是最终生成的代理实例，一般不会用到；  
 \* 2,method是被代理目标实例的某个具体方法，通过它可以发起目标实例方法的反射调用；  
 \* 3,args是通过被代理实例某一个方法的入参，在方法反射调用时使用。  
 \*/*public class JDKDynamicProxy implements InvocationHandler {  
 private Logger logger = LoggerFactory.*getLogger*(this.getClass());  
  
 //被代理类的对象  
 private Object delegate;  
  
 //绑定被代理对象  
 public Object bind(Object delegate) {  
 this.delegate = delegate;  
 //返回实现了被代理类所实现的所有接口的Object对象，即动态代理，需要强制转型  
 //创建代理对象，注意这里被代理的对象类必须实现至少一个接口  
 return Proxy.*newProxyInstance*(delegate.getClass().getClassLoader(), delegate.getClass().getInterfaces(), this);  
 }  
  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 logger.info(System.*currentTimeMillis*() + "：开始！");

// 调用原始对象的方法  
 Object object = method.invoke(this.delegate, args);  
 if (method.getName().matches("say[a-zA-Z0-9]+")) {  
 logger.info("这是一个以say开头的方法！");  
 }  
 logger.info(System.*currentTimeMillis*() + "：结束！");  
 return object;  
 }  
}

JDK自从1.3版本开始，就引入了动态代理，并且经常被用来动态地创建代理。JDK的动态代理用起来非常简单，但它有一个限制，就是使用动态代理的对象必须实现一个或多个接口。比如上面的Hello类，实现了IHello接口，所以可以用JDK的动态代理。如果想代理没有实现接口的继承的类，该怎么办？ CGLIB就是最好的选择。

### CGLIB动态代理

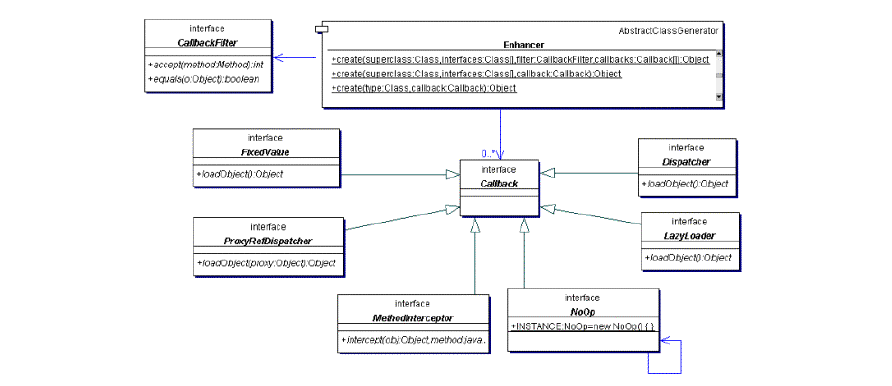
CGlib是一个强大的，高性能，高质量的Code生成类库。它可以在运行期扩展Java类与实现Java接口。其底层是通过小而快的字节码处理框架ASM（http://forge.ow2.org/projects/asm，使用BSD License）来转换字节码并生成新的类。大部分功能实际上是asm所提供的，CGlib只是封装了asm，简化了asm的操作，实现了在运行期动态生成新的class。

　　CGlib被许多AOP的框架使用，例如Spring AOP和dynaop，为他们提供方法的interception（拦截）；最流行的OR Mapping工具hibernate也使用CGLIB来代理单端single-ended（多对一和一对一）关联（对集合的延迟抓取，是采用其他机制实现的）；EasyMock和jMock是通过使用模仿（moke）对象来测试java代码的包，它们都通过使用CGLIB来为那些没有接口的类创建模仿（moke）对象。

　　CGLIB包的基本代码很少，但学起来有一定的困难，主要是缺少文档，API描述过于简单，这也是开源软件的一个不足之处。目前CGLIB的版本是cglib-2.2.jar，主要由一下部分组成：  
　　（1）net.sf.cglib.core：底层字节码处理类，他们大部分与ASM有关系。  
　　（2）net.sf.cglib.transform：编译期或运行期类和类文件的转换。  
　　（3）net.sf.cglib.proxy ：实现创建代理和方法拦截器的类。  
　　（4）net.sf.cglib.reflect ：实现快速反射和C#风格代理的类。  
　　（5）net.sf.cglib.util：集合排序工具类。  
　　（6）net.sf.cglib.beans：JavaBean相关的工具类。

CGLIB包是在ASM之上的一个高级别的层。对代理那些没有实现接口的类非常有用。本质上，它是通过动态的生成一个子类去覆盖所要代理类的不是final的方法，并设置好callback，则原有类的每个方法调用就会转变成调用用户定义的拦截方法（interceptors），这比JDK动态代理方法快多了。可见，Cglib的原理是对指定的目标类动态生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类和final方法进行代理。

下图表示Cglib常用到的几类。



创建一个具体类的代理时，通常要用到的CGLIB包的APIs：

**net.sf.cglib.proxy.Callback**接口：在CGLIB包中是一个很关键的接口，所有被net.sf.cglib.proxy.Enhancer类调用的回调（callback）接口都要继承这个接口。

**net.sf.cglib.proxy.MethodInterceptor**接口：是最通用的回调（callback）类型，它经常被AOP用来实现拦截（intercept）方法的调用。能够满足任何的拦截（interception ）需要，当对有些情况下可能过度。这个接口只定义了一个方法。

代理类的所有方法经常会用到回调（callback），当然你也可以使用net.sf.cglib.proxy.CallbackFilter 有选择的对一些方法使用回调（callback），这种考虑周详的控制特性在JDK的动态代理中是没有的。CallbackFilter可以实现不同的方法使用不同的回调方法。所以CallbackFilter称为"回调选择器"更合适一些。  
CallbackFilter中的accept方法，根据不同的method返回不同的值i，这个值是在callbacks中callback对象的序号，就是调用了callbacks[i]。

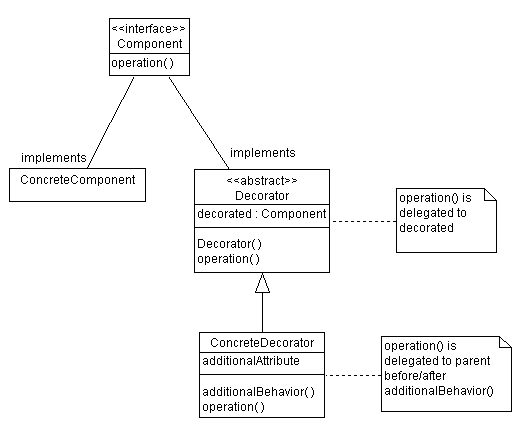
在JDK代理中，对 java.lang.reflect.InvocationHandler方法的调用对代理类的所有方法都有效。

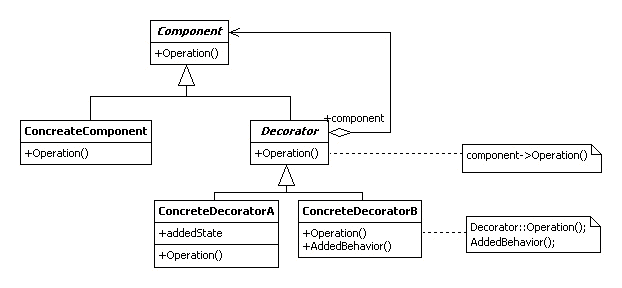
*/\*\*  
 \** ***@author*** *esther  
 \* 2018-04-25 16:24  
 \* Cglib的原理是对指定的目标类动态生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类和final方法进行代理。  
 \*  
 \* 拦截器：实现MethodInterceptor接口的类，在intercept方法中实现对代理目标类的方法拦截。  
 \* 但同时Cglib为简化和提高性能提供了一些专门的回调类型如FixedValue（可以在实现的方法loadObject中指定返回固定值，而不调用目标类函数）、  
 \* NoOp（把对回调方法的调用直接委派到这个方法的父类，即不进行拦截）  
 \*/*public class CGLIBDynamicProxy implements MethodInterceptor {  
 private Logger logger = LoggerFactory.*getLogger*(this.getClass());  
  
 private Enhancer enhancer = new Enhancer();  
  
 //　要注意的是，target类是作为产生的代理的父类传进来的。不同于JDK的动态代理，它不能在创建代理时传target对象，target对象必须被CGLIB包来创建。  
 public Object getProxy(Class targetClass) {  
 enhancer.setSuperclass(targetClass); //① 设置需要创建子类的类  
 // 设置单一回调对象，在回调中拦截对目标方法的调用  
 enhancer.setCallback(this);  
 return enhancer.create(); //②通过字节码技术动态创建子类实例  
  
 }  
  
 //③拦截父类所有方法的调用  
  
 */\*\*  
 \* \* 回调方法:在代理实例上拦截并处理目标方法的调用，返回结果  
 \*  
 \** ***@param*** *obj 代理类  
 \** ***@param*** *method 被代理的方法  
 \** ***@param*** *args 该方法的参数数组  
 \** ***@param*** *proxy  
 \** ***@return*** *\** ***@throws*** *Throwable  
 \*/* @Override  
 public Object intercept(Object obj, Method method, Object[] args,  
 MethodProxy proxy) throws Throwable {  
 PerformanceMonitor.*begin*(obj.getClass().getName() + "." + method.getName());//③-1  
 // 调用目标方法，用methodProxy,  
 // 而不是原始的method，以提高性能  
 Object result = proxy.invokeSuper(obj, args); //③-2 使用Cglib代理调用  
 PerformanceMonitor.*end*();//③-1通过代理类调用父类中的方法  
 return result;  
 }  
}

# 装饰者模式

## 简介

装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。





## 装饰器模式的应用场景

1、需要扩展一个类的功能。

2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

优点：比静态继承更具灵活性，**Avoids feature-laden classes high up in the hierarchy，**装饰对象和被装饰对象不一样。

缺点：产生过多相似的对象，不易排错！

public interface IProgrammer {  
 void coding();  
}

public class JavaProgrammer implements IProgrammer {  
 @Override  
 public void coding() {  
 System.*out*.println("Java程序员编码。");  
 }  
}

public abstract class Decorator implements IProgrammer {  
 private IProgrammer programmer;  
 public Decorator(IProgrammer programmer){  
 super();  
 this.programmer=programmer;  
 }  
 @Override  
 public void coding() {  
 programmer.coding();  
 }  
}

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-07 16:46  
 \* 装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例  
 \*/*public class DesignDecorator extends Decorator {  
 public DesignDecorator(IProgrammer programmer) {  
 super(programmer);  
 }  
  
 @Override  
 public void coding() {  
 System.*out*.println("编码前先设计。");  
 super.coding();  
 }  
}

# 适配器模式

## 简介

将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一下工作。--Gang of Four

主要分为三类：类的适配、对象的适配、接口的适配。

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 10:52  
 \* 介绍：src类: 我们有的220V电压  
 \*/*public class Voltage220 {  
 public int output220V() {  
 int src = 220;  
 System.*out*.println("我是" + src + "V");  
 return src;  
 }  
}

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 11:16  
 \* 介绍：src接口  
 \*/*public interface IVoltage220 {  
 int output220V();  
}

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 10:54  
 \* 介绍：dst接口：客户需要的5V电压  
 \*/*public interface IVoltage5 {  
 int output5V();  
}

## 类的适配

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 10:57  
 \* 介绍：Adapter类：完成220V-5V的转变  
 \* 通过继承src类，实现 dst 类接口，完成src->dst的适配。  
 \*/  
  
/\*\*  
 \* 类适配器需要继承src类这一点算是一个缺点，  
 因为这要求dst必须是接口，有一定局限性;  
 且src类的方法在Adapter中都会暴露出来，也增加了使用的成本。  
 但同样由于其继承了src类，所以它可以根据需求重写src类的方法，使得Adapter的灵活性增强了。  
 \*/*public class VoltageAdapter4Class extends Voltage220 implements IVoltage5 {  
  
 @Override  
 public int output5V() {  
 int src = output220V();  
 System.*out*.println("适配器工作开始适配电压");  
 int dst = src / 44;  
 System.*out*.println("适配完成后输出电压：" + dst);  
 return dst;  
 }  
}

## 对象的适配

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 10:56  
 \* 介绍：对象适配器模式：  
 \* 持有 src类，实现 dst 类接口，完成src->dst的适配。 。以达到解决兼容性的问题。  
 \*/*public class VoltageAdapter4Object implements IVoltage5 {  
 private Voltage220 voltage220;  
  
 public VoltageAdapter4Object(Voltage220 voltage220) {  
 this.voltage220 = voltage220;  
 }  
  
 @Override  
 public int output5V() {  
 int dst = 0;  
 if (Objects.*nonNull*(voltage220)) {  
 int src = voltage220.output220V();  
 System.*out*.println("对象适配器工作，开始适配电压");  
 dst = src / 44;  
 System.*out*.println("适配完成后输出电压：" + dst);  
 }  
 return dst;  
 }  
}

## 接口的适配

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 10:57  
 \* 缺省适配器模式：  
当不需要全部实现接口提供的方法时，可先设计一个抽象类实现接口，并为该接口中每个方法提供一个默认实现（空方法），  
那么该抽象类的子类可有选择地覆盖父类的某些方法来实现需求，它适用于一个接口不想使用其所有的方法的情况。  
 \*/*public class VoltageAdapter4Interface implements IVoltage220, IVoltage5 {  
 @Override  
 public int output220V() {  
 int src = 220;  
 System.*out*.println("我是" + src + "V");  
 return src;  
 }  
  
 @Override *public int output5V() {  
 int src = output220V();* System.*out*.println("适配器工作开始适配电压");  
 int dst = src / 44;  
 System.*out*.println("适配完成后输出电压：" + dst);  
 return dst;  
 }  
}

## 三种适配的区别

类适配器，以类给到，在Adapter里，就是将src当做类，继承，  
 对象适配器，以对象给到，在Adapter里，将src作为一个对象，持有。  
 接口适配器，以接口给到，在Adapter里，将src作为一个接口，实现。

# 外观模式

# 桥接模式

# 组合模式

# 享元模式

# 策略模式

# 管道模式

# 模板方法

# 观察者模式

## 简介

观察者模式是对象的行为模式，外号非常多...！！又叫发布-订阅(Publish/Subscribe)模式、模型-视图(Model/View)模式、源-监听器(Source/Listener)模式或从属者(Dependents)模式。

     观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

## 针对的问题

被观察者对象在状态变化时，通知所有观察者对象，使它们能够自动更新自己。保证整体的数据一致性。此种模式通常被用来实现事件处理系统。

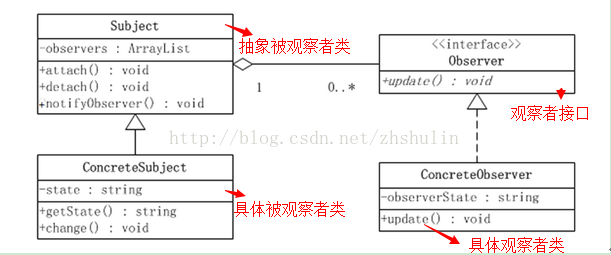
适用性：

1）当一个抽象模型有两个方面,其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

2）当对一个对象的改变需要同时改变其它对象,而不知道具体有多少对象有待改变。

3）当一个对象必须通知其它对象，而它又不能假定其它对象是谁。

## 角色组成



**Ø 抽象主题(Subject)角色：**

  抽象主题角色把所有对观察者对象的引用保存在一个聚集（比如ArrayList对象）里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象，抽象主题角色又叫做抽象被观察者(Observable)角色。

**Ø 具体主题(ConcreteSubject)角色：**

  将有关状态存入具体观察者对象；在具体主题的内部状态改变时，给所有登记过的观察者发出通知。具体主题角色又叫做具体被观察者(Concrete Observable)角色。

**Ø 抽象观察者(Observer)角色：**

  为所有的具体观察者定义一个接口，在得到主题的通知时更新自己，这个接口叫做更新接口。

**Ø 具体观察者(ConcreteObserver)角色：**

  存储与主题的状态自恰的状态。具体观察者角色实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态协调。如果需要，具体观察者角色可以保持一个指向具体主题对象的引用。

## 举例说明

Subject：抽象被观察者

1. **package** org.zsl.designmodel.observer;
2. **import** java.util.ArrayList;
3. **import** java.util.List;
4. /\*\*
5. \* 被观察者角色
6. \*/
7. **public** **abstract** **class** Subject {
8. **private** List<Observer> list = **new** ArrayList<Observer>();
9. /\*\*
10. \* 注册观察者对象
11. \* @param observer  观察者
12. \*/
13. **public** **void** registerObserver(Observer observer){
14. list.add(observer);
15. System.out.println("注册了一个观察者角色！");
16. }
17. /\*\*
18. \* 删除观察者对象
19. \* @param observer  观察者
20. \*/
21. **public** **void** removeObserver(Observer observer){
22. list.remove(observer);
23. }
24. /\*\*
25. \* 通知所有注册的观察者对象，我的状态改变咯
26. \* @param newState
27. \*/
28. **public** **void** notifyAllObservers(String newState){
29. **for**(Observer observer : list){
30. observer.update(newState);
31. }
32. }
33. }

Observer ：观察者接口，只有一个update()方法

1. **package** org.zsl.designmodel.observer;
2. /\*\*
3. \* 观察者接口
4. \*/
5. **public** **interface** Observer {
6. /\*\*
7. \* 更新接口
8. \* @param state 更新的状态
9. \*/
10. **public** **void** update(String state);
11. }

Heater：热水器，一个具体被观察者

1. **package** org.zsl.designmodel.observer;
2. **public** **class** Heater **extends** Subject {
3. **private** **int** temperature;
4. **public** **int** getTemperature() {
5. **return** temperature;
6. }
7. **public** **void** setTemperature(**int** temperature) {
8. **this**.temperature = temperature;
9. }
10. **public** **void** boilWater(){
11. **for**(**int** i=95;i<105;i++){
12. temperature = i;
13. **this**.setTemperature(temperature);   //设置新的温度
14. **this**.notifyAllObservers(Integer.toString(temperature)); //通知所有注册的观察者
15. }
16. }
17. }

Display：既是观察者，又是被观察者

1. **package** org.zsl.designmodel.observer;
2. /\*\*
3. \* 显示器，既是观察者又是被观察者
4. \*/
5. **public** **class** Display **extends** Subject **implements** Observer {
6. **private** **boolean** isBoiled = **false**;   //表示水是否烧开了
7. **private** **int** displayTemperature;     //定义显示器的温度
8. **public** **boolean** isBoiled() {
9. **return** isBoiled;
10. }
11. **public** **void** setBoiled(**boolean** isBoiled) {
12. **this**.isBoiled = isBoiled;
13. }
14. @Override
15. **public** **void** update(String state) {
16. displayTemperature = Integer.parseInt(state);
17. System.out.println("当前显示器显示的温度是："+displayTemperature);
18. **this**.displayTemperature(displayTemperature);
19. }
20. **private** **void** displayTemperature(**int** temperature){
21. **if**(temperature>100){
22. **this**.setBoiled(**true**);
23. **this**.notifyAllObservers(Boolean.toString(isBoiled));
24. }
25. }
26. }

Alerm：只是观察者

1. **package** org.zsl.designmodel.observer;
2. **public** **class** Alarm **implements** Observer {
3. **private** String observerState;       //定义观察者的状态
4. @Override
5. **public** **void** update(String state) {
6. Boolean flag = Boolean.parseBoolean(state);
7. **if**(flag){
8. System.out.println("报警器响了，水温超过100度了。");
9. }
10. }
11. }

测试

1. **package** org.zsl.designmodel.observer;
2. **public** **class** Test {
3. **public** **static** **void** main(String[] args) {
4. Heater heater = **new** Heater();
5. Display display = **new** Display();
6. Alarm alarm = **new** Alarm();
7. heater.registerObserver(display);
8. display.registerObserver(alarm);
9. heater.boilWater();
10. }
11. }

结果：



## 优点

a、支持松耦合和减少依赖性

客户端不再依赖于观察器，因为通过使用主体和 Observer 接口对客户端进行了隔离。 许多框架具有此优点，在这些框架中的应用程序组件可以注册为当（低级）框架事件发 生时得到通知。结果，框架将调用应用程序组件，但不会依赖于它。

b、提高了应用程序的可维护性和重用性

面向对象设计的一个原则是：系统中的每个类将重点放在某一个功能上，而不是其他方面。一个对象只做一件事情，并且将他做好。观察者模式在模块之间划定了清晰的界限，提高了应用程序的可维护性和重用性。

c、观察器数目可变

观察器可以在运行时附加和分离，因为主体对于观察器数目没有任何假定。此功能在这样的情况下是很有用的：观察器数在设计时是未知的。例如，如果用户在应用程序中打开的每个窗口都需要一个观察器。

综合来讲，使用观察者模式最大的好处就是减少死循环式的轮循带来的资源无端消耗，并且有着良好的可扩展性。

## 缺点

a、性能降低。

     在许多实现中，观察器的 update() 方法可能与主体在同一线程中执行。如果观察器列表很长，则执行 Notify() 方法可能需要很长时间。抽取对象依赖性并不意味着添加观察器对应用程序没有任何影响。

b、内存泄漏。

      在 Observer 中使用的回调机制（当对象注册为以后调用时）会产生一个常见的错误，从而导致内存泄漏，甚至是在托管的 C# 代码中。假定观察器超出作用范围，但忘记取消对主体的订阅，那么主体仍然保留对观察器的引用。此引用防止垃圾收集在主体对象也被破坏之前重新分配与观察器关联的内存。如果观察器的生存期比主体的生存期短得多（通常是这种情况），则会导致严重的内存泄漏。

c、隐藏的依赖项。

      观察器的使用将显式依赖性（通过方法调用）转变为隐式依赖性（通过观察器）。如果在整个应用程序中广泛地使用观察器，则开发人员几乎不可能通过查看源代码来了解所发生的事情。这样，就使得了解代码更改的含意非常困难。此问题随传播级别急剧增大（例如，充当 Subject 的观察器）。因此，应该仅在少数定义良好的交互（如 Model-View-Controller 模式中模型和视图之间的交互）中使用观察器。最好不要在域对象之间使用观察器。

d、测试 / 调试困难。

    尽管松耦合是一项重大的体系结构功能，但是它可以使开发更困难。将两个对象去耦的情况越多，在查看源代码或类的关系图时了解它们之间的依赖性就越难因此，仅当可以安全地忽略两个对象之间的关联时才应该将它们松耦合（例如，如果观察器没有副作用）。

# 迭代子模式

# 责任链模式

# 命令模式

# 备忘录模式

# 状态模式

# 访问者模式

# 中介者模式

# 解释器模式

# 并发型模式

# 线程池模式

# Spring中的设计模式

## 单例模式

在Spring中，bean可以被定义为两种模式：prototype(多例)和singleton(单例)。

Singleton(单例)：只有一个共享的实例存在，所有对这个bean的请求都会返回这个唯一的实例。

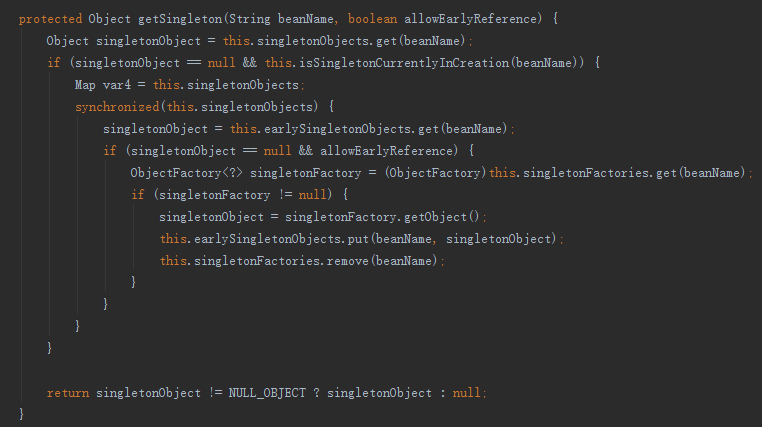
Prototype(多例)：对这个bean的每次请求都会创建一个新的bean实例，类似于new。

Spring依赖注入Bean实例默认是单例的，我们由此展开。

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



通过将工厂函数传入getSingleton函数中，就可以获得一个Bean单例。单例的生成是通过修改createBean函数的参数实现的，其中mbd是一个RootBeanDefinition类，它存储了生成Bean实例所需要的信息。在createBean之中的代码里，程序调用实例化Bean的函数initializeBean。

 从上面代码可以看到，spring依赖注入时，使用了**双重判断加锁**的单例模式，首先从缓存singletonObjects（实际上是一个map）中获取bean实例，如果为null，对缓存singletonObjects加锁，然后再从缓存中获取bean，如果继续为null，就创建一个bean。

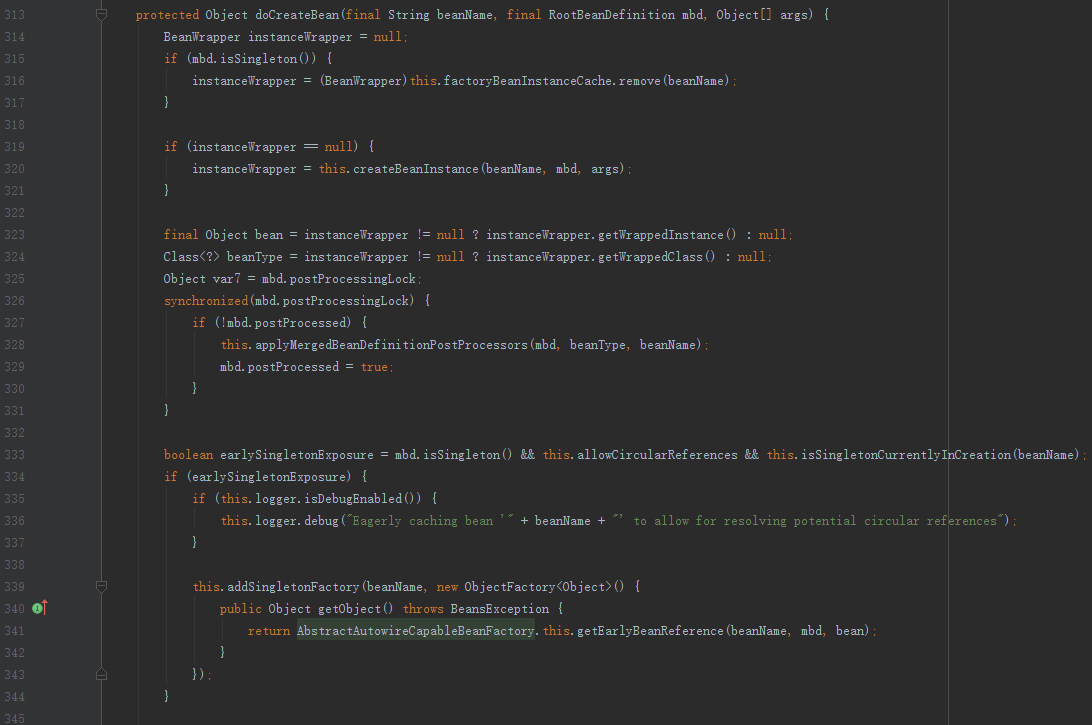
**这样双重判断，能够避免在加锁的瞬间，有其他依赖注入引发bean实例的创建，从而造成重复创建的结果。**

     在这里Spring并没有使用私有构造方法来创建bean，而是通过singletonFactory.getObject()返回具体beanName对应的ObjectFactory来创建bean。

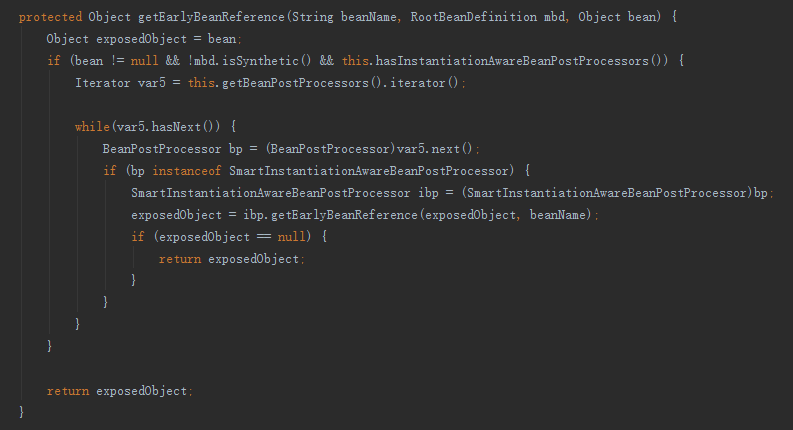
我们一路跟踪下去，发现实际上是调用了AbstractAutowireCapableBeanFactory的doCreateBean方法，返回了BeanWrapper包装并创建的bean实例。

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

见如下代码：

320行createBeanInstance()创建bean实例返回给BeanWrapper

339行addSingletonFactory()增加beanName和ObjectFactory的键值对应关系。



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

## 工厂模式

// AbstractFactoryBean.java  
// 继承了FactoryBean，工厂Bean的主要作用是为了实现getObject()返回Bean实例  
public abstract class AbstractFactoryBean<T> implements FactoryBean<T>, BeanClassLoaderAware, BeanFactoryAware, InitializingBean, DisposableBean {  
  
 // 定义了获取对象的前置判断工作，创建对象的工作则交给了一个抽象方法  
// 这里判断了Bean是不是单例并且是否已经被加载过了（未初始化但加载过了，这个问题涉及到Spring处理循环依赖，以后会讨论到）  
 public final T getObject() throws Exception {  
 return this.isSingleton()?(this.initialized?this.singletonInstance:this.getEarlySingletonInstance()):this.createInstance();  
 }  
 // 由子类负责具体创建对象  
 protected abstract T createInstance() throws Exception;  
}

之所以这么写是因为这种写法带来了两个好处:

（1） 保证了创建Bean的方式的多样性   
 Bean工厂有很多种，它们负责创建各种各样不同的Bean，比如Map类型的Bean，List类型的Bean，Web服务Bean，子类们不需要关心单例或非单例情况下是否需要额外操作，只需要关心如何创建Bean，并且创建出来的Bean是多种多样的。

（2） 严格规定了Bean创建前后的其它动作   
 虽然子类可以自由的去创建Bean，但是创建Bean之前的准备工作以及创建Bean之后对Bean的处理工作是AbstractFactoryBean设定好了的，子类不需要关心，也没权力关心，在这个例子中父类只负责一些前置判断工作。

工厂方法模式非常的有趣，它给了子类创建实例的自由，又严格的规定了实例创建前后的业务流程。

## 适配器模式

public interface Controller {  
}

public class AnnotationController implements Controller {  
 public void doAnnotationHandler() {  
 System.*out*.println("执行AnnotationController.doAnnotationHandler()");  
 }  
}

public class HttpController implements Controller {  
 public void doHttpHandler() {  
 System.*out*.println("执行HttpController.doHttpHandler()");  
 }  
}

public class SimpleController implements Controller {  
 public void doSimplerHandler(){  
 System.*out*.println("执行SimpleController.doSimplerHandler()");  
 }  
}

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-04 14:13  
 \* 定义一个Adapter接口  
 \*/*public interface HandlerAdapter {  
 public boolean supports(Object handler);  
 public void handle(Object handler);  
}

public class AnnotationHandlerAdapter implements HandlerAdapter {  
 @Override  
 public boolean supports(Object handler) {  
 return handler instanceof AnnotationController;  
 }  
  
 @Override  
 public void handle(Object handler) {  
 ((AnnotationController)handler).doAnnotationHandler();  
 }  
}

public class HttpHandlerAdapter implements HandlerAdapter {  
 @Override  
 public boolean supports(Object handler) {  
 return handler instanceof HttpController;  
 }  
  
 @Override  
 public void handle(Object handler) {  
 ((HttpController)handler).doHttpHandler();  
 }  
}

public class SimpleHandlerAdapter implements HandlerAdapter {  
 @Override  
 public boolean supports(Object handler) {  
 return (handler instanceof SimpleController);  
 }  
  
 @Override  
 public void handle(Object handler) {  
 ((SimpleController) handler).doSimplerHandler();  
 }  
}

Client类：

public class DispatcherServlet {  
 public static List<HandlerAdapter> *handlerAdapters* = new ArrayList<>();  
  
 public DispatcherServlet() {  
 *handlerAdapters*.add(new AnnotationHandlerAdapter());  
 *handlerAdapters*.add(new SimpleHandlerAdapter());  
 *handlerAdapters*.add(new HttpHandlerAdapter());  
 }  
  
 public void doDispatch(Controller controller) {  
 //此处模拟SpringMVC从request取handler的对象，仅仅new出，可以出，  
 //不论实现何种Controller，适配器总能经过适配以后得到想要的结果  
// HttpController controller = new HttpController();  
// AnnotationController controller = new AnnotationController();  
// SimpleController controller = new SimpleController();  
 //得到对应适配器  
 HandlerAdapter handlerAdapter = getHandlerAdapter(controller);  
 //通过适配器执行对应的controller对应方法  
 handlerAdapter.handle(controller);  
 }  
  
 public HandlerAdapter getHandlerAdapter(Controller controller) {  
 for (HandlerAdapter handlerAdapter : *handlerAdapters*) {  
 if (handlerAdapter.supports(controller)) {  
 return handlerAdapter;  
 }  
 }  
 return null;  
 }  
}

## 代理模式

## 装饰者模式

通过使用修饰模式，可以在运行时扩充一个类的功能。原理是：增加一个修饰类包裹原来的类，包裹的方式一般是通过在将原来的对象作为修饰类的构造函数的参数。装饰类实现新的功能，但是，在不需要用到新功能的地方，它可以直接调用原来的类中的方法。修饰类必须和原来的类有相同的接口。

装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例。

Spring中类中带有Wrapper的都是包装类，如下创建bean就是典型的装饰器模式。



## 策略模式

### 定义

在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。策略模式实际就是一堆算法族的封装。

### Spring中策略模式的应用

当bean需要访问资源配置文件时，Spring有两种方式

* 代码中获取Rescource实例
* 依赖注入   
  第一种方式需要获取rescource资源的位置，代码中耦合性太高，而今我们一直使用注解，依赖注入的方式去获取。这样的话就无需修改程序，只改配置文件即可。

<beans>

<bean id="test" class="com.example.Test">

<!-- 注入资源 -->

<property name="tmp" value="classpath:book.xml"/>

</bean>

</beans>

在依赖注入的过程中，Spring会调用ApplicationContext 来获取Resource的实例。然而，Resource 接口封装了各种可能的资源类型，包括了：UrlResource，ClassPathResource，FileSystemResource等，Spring需要针对不同的资源采取不同的访问策略。在这里，Spring让ApplicationContext成为了资源访问策略的“决策者”。在资源访问策略的选择上，Spring采用了策略模式。当 Spring 应用需要进行资源访问时，它并不需要直接使用 Resource 实现类，而是调用 ApplicationContext 实例的 getResource() 方法来获得资源，ApplicationContext 将会负责选择 Resource 的实现类，也就是确定具体的资源访问策略，从而将应用程序和具体的资源访问策略分离开来。

ApplicationContext ctx = new Class PathXmlApplicationContext("bean.xml");

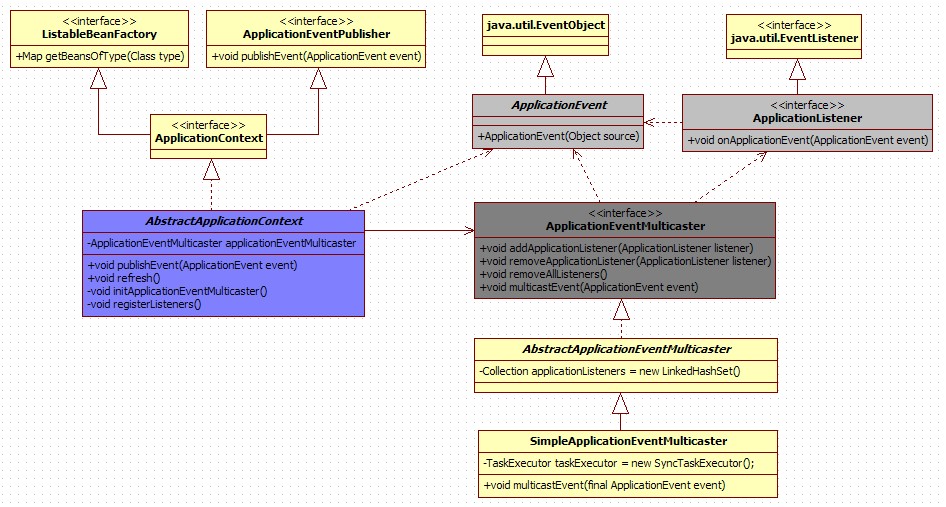
Resource res = ctx.getResource("book.xml");

上面的代码中，Spring 将采用和 ApplicationContext 相同的策略来访问资源。即： ApplicationContext 是 ClassPathXmlApplicationContext，则res 就是 ClassPathResource 实例。若将代码改为：

ApplicationContext ctx = new Class FileSystemXmlApplicationContext("bean.xml");

则再次调用ctx.getResource时，res 就是 ClassPathResource 实例。

## 观察者模式



### ApplicationEvent

package org.springframework.context;  
import java.util.EventObject;  
public abstract class ApplicationEvent extends EventObject {  
 private static final long serialVersionUID = 7099057708183571937L;  
 private final long timestamp = System.currentTimeMillis();  
  
 public ApplicationEvent(Object source) {  
 super(source);  
 }  
 public final long getTimestamp() {  
 return this.timestamp;  
 }  
}

ApplicationEvent继承自jdk的EventObject,所有的事件都需要继承ApplicationEvent,并且通过source得到事件源。该类的实现类ApplicationContextEvent表示ApplicaitonContext的容器事件。

### ApplicationListener

package org.springframework.context;  
import java.util.EventListener;  
public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {  
 void onApplicationEvent(E var1);  
}

ApplicationListener继承自jdk的EventListener,所有的监听器都要实现这个接口,这个接口只有一个onApplicationEvent()方法,该方法接受一个ApplicationEvent或其子类对象作为参数,在方法体中,可以通过对Event类的判断来进行相应的处理。当事件触发时所有的监听器都会收到消息,如果你需要对监听器的接收顺序有要求,可是实现该接口的一个实现SmartApplicationListener,通过这个接口可以指定监听器接收事件的顺序。

### ApplicationContext

事件机制的实现需要三个部分：事件源,事件,事件监听器。在上面介绍的ApplicationEvent就相当于事件,ApplicationListener相当于事件监听器,这里的事件源说的就是applicaitonContext。

　　ApplicationContext是spring中的全局容器,翻译过来是"应用上下文"的意思,它用来负责读取bean的配置文档,管理bean的加载,维护bean之间的依赖关系,可以说是负责bean的整个生命周期,再通俗一点就是我们平时所说的IOC容器。

     Application作为一个事件源,需要显示的调用publishEvent方法,传入一个ApplicationEvent的实现类对象作为参数,每当ApplicationContext发布ApplicationEvent时,所有的ApplicationListener就会被自动的触发。

ApplicationContext接口实现了ApplicationEventPublisher接口,后者有一个很重要的方法:

package org.springframework.context;  
public interface ApplicationEventPublisher {  
 void publishEvent(ApplicationEvent var1);  
 void publishEvent(Object var1);  
}

我们常用的ApplicationContext都继承了AbstractApplicationContext,像我们平时常见的ClassPathXmlApplicationContext、XmlWebApplicationContex也都是继承了它。AbstractApplicationcontext是ApplicationContext接口的抽象实现类,在该类中实现了publishEvent方法。

protected void publishEvent(Object event, ResolvableType eventType) {  
 Assert.notNull(event, "Event must not be null");  
 if (this.logger.isTraceEnabled()) {  
 this.logger.trace("Publishing event in " + this.getDisplayName() + ": " + event);  
 }  
  
 Object applicationEvent;  
 if (event instanceof ApplicationEvent) {  
 applicationEvent = (ApplicationEvent)event;  
 } else {  
 applicationEvent = new PayloadApplicationEvent(this, event);  
 if (eventType == null) {  
 eventType = ((PayloadApplicationEvent)applicationEvent).getResolvableType();  
 }  
 }  
  
 if (this.earlyApplicationEvents != null) {  
 this.earlyApplicationEvents.add(applicationEvent);  
 } else {  
 this.getApplicationEventMulticaster().multicastEvent((ApplicationEvent)applicationEvent, eventType);  
 }  
  
 if (this.parent != null) {  
 if (this.parent instanceof AbstractApplicationContext) {  
 ((AbstractApplicationContext)this.parent).publishEvent(event, eventType);  
 } else {  
 this.parent.publishEvent(event);  
 }  
 }  
  
}

在这个方法中,我们看到了一个getApplicationEventMulticaster().这就要牵扯到另一个类ApplicationEventMulticaster。

### ApplicationEventMulticaster

属于事件广播器,它的作用是把Applicationcontext发布的Event广播给所有的监听器。

public interface ApplicationEventMulticaster {  
 void addApplicationListener(ApplicationListener<?> var1);  
  
 void addApplicationListenerBean(String var1);  
  
 void removeApplicationListener(ApplicationListener<?> var1);  
  
 void removeApplicationListenerBean(String var1);  
  
 void removeAllListeners();  
  
 void multicastEvent(ApplicationEvent var1);  
  
 void multicastEvent(ApplicationEvent var1, ResolvableType var2);  
}

ApplicationEventMulticaster的实现类SimpleApplicationEventMulticaster：里面实现了multicastEvent()的具体方法。

public void multicastEvent(final ApplicationEvent event, ResolvableType eventType) {  
 ResolvableType type = eventType != null ? eventType : this.resolveDefaultEventType(event);  
 Iterator var4 = this.getApplicationListeners(event, type).iterator();  
  
 while(var4.hasNext()) {  
 final ApplicationListener<?> listener = (ApplicationListener)var4.next();  
 Executor executor = this.getTaskExecutor();  
 if (executor != null) {  
 executor.execute(new Runnable() {  
 public void run() {  
 SimpleApplicationEventMulticaster.this.invokeListener(listener, event);  
 }  
 });  
 } else {  
 this.invokeListener(listener, event);  
 }  
 }  
  
}  
  
private ResolvableType resolveDefaultEventType(ApplicationEvent event) {  
 return ResolvableType.forInstance(event);  
}  
  
protected void invokeListener(ApplicationListener listener, ApplicationEvent event) {  
 ErrorHandler errorHandler = this.getErrorHandler();  
 if (errorHandler != null) {  
 try {  
 listener.onApplicationEvent(event);  
 } catch (Throwable var6) {  
 errorHandler.handleError(var6);  
 }  
 } else {  
 try {  
 listener.onApplicationEvent(event);  
 } catch (ClassCastException var5) {  
 LogFactory.getLog(this.getClass()).debug("Non-matching event type for listener: " + listener, var5);  
 }  
 }  
}

　　在AbstractApplicationcontext中有一个applicationEventMulticaster的成员变量,提供了监听器Listener的注册方法。

public abstract class AbstractApplicationContext extends DefaultResourceLoader implements ConfigurableApplicationContext, DisposableBean {

private ApplicationEventMulticaster applicationEventMulticaster;

protected void registerListeners() {  
 Iterator var1 = this.getApplicationListeners().iterator();  
  
 while(var1.hasNext()) {  
 ApplicationListener<?> listener = (ApplicationListener)var1.next();  
 this.getApplicationEventMulticaster().addApplicationListener(listener);  
 }  
  
 String[] listenerBeanNames = this.getBeanNamesForType(ApplicationListener.class, true, false);  
 String[] var7 = listenerBeanNames;  
 int var3 = listenerBeanNames.length;  
  
 for(int var4 = 0; var4 < var3; ++var4) {  
 String listenerBeanName = var7[var4];  
 this.getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);  
 }  
  
 Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;  
 this.earlyApplicationEvents = null;  
 if (earlyEventsToProcess != null) {  
 Iterator var9 = earlyEventsToProcess.iterator();  
  
 while(var9.hasNext()) {  
 ApplicationEvent earlyEvent = (ApplicationEvent)var9.next();  
 this.getApplicationEventMulticaster().multicastEvent(earlyEvent);  
 }  
 }  
}

### 举例说明

事件：

public class DIYEvent extends ApplicationEvent {  
 private static final long *serialVersionUID* = 7099057708183571977L;  
  
 public DIYEvent(String source) {  
 super(source);  
 }  
}

监听器：

@Component  
public class DIYListener implements ApplicationListener<DIYEvent> {  
 @Override  
 public void onApplicationEvent(DIYEvent diyEvent) {  
 System.*out*.println("自定义监听器执行");  
 System.*out*.println(diyEvent.getSource());  
 }  
}

有顺序的监听器：

@Component  
public class DIYOrderdListener implements SmartApplicationListener {  
 @Override  
 public boolean supportsEventType(Class<? extends ApplicationEvent> aClass) {  
 return aClass==DIYEvent.class;  
 }  
  
 @Override  
 public boolean supportsSourceType(Class<?> aClass) {  
 return true;  
 }  
  
 @Override  
 public void onApplicationEvent(ApplicationEvent applicationEvent) {  
 if (applicationEvent instanceof DIYEvent) {  
 System.*out*.println("自定义顺序监听器执行....DIYOrderdListener");  
 System.*out*.println(applicationEvent.getSource());  
 }  
 else{  
 System.*out*.println("没有满足的监听器.....DIYOrderdListener");  
 }  
 }  
  
 @Override  
 public int getOrder() {  
 return 1;  
 }  
}

测试类：

public class DIYTest{  
 private ApplicationContext applicationContext=new ClassPathXmlApplicationContext("classpath:/spring/applicationContext.xml");  
  
 @Test  
 public void diyTest(){  
 applicationContext.publishEvent(new DIYEvent("测试数据"));  
 }  
}

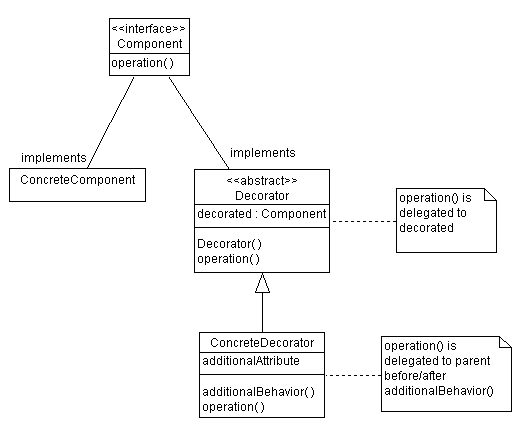
结果：

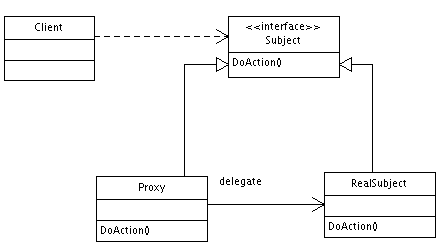
自定义监听器执行

测试数据

# 总结

## 装饰者模式和代理模式区别





  这两个图可能使我们产生困惑。这两个设计模式看起来很像。对装饰器模式来说，装饰者（decorator）和被装饰者（decoratee）都实现同一个接口。此外，不论我们使用哪一个模式，都可以很容易地在真实对象的方法前面或者后面加上自定义的方法。

       然而，实际上，在装饰器模式和代理模式之间还是有很多差别的。装饰器模式关注于在一个对象上动态的添加方法，然而代理模式关注于控制对对象的访问。换句话说，用代理模式，代理类（proxy class）可以对它的客户隐藏一个对象的具体信息。因此，当使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例。并且，当我们使用装饰器模式的时候，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器。

        代理模式中，代理类对被代理的对象有控制权，决定其执行或者不执行。而装饰模式中，装饰类对代理对象没有控制权，只能为其增加一层装饰，以加强被装饰对象的功能，仅此而已。

Spring中AOP用到了代理模式和反射机制。

JDK中的java.io.\*包大量使用了装饰者模式，如

OutputStream out = new DataOutputStream(new FileOutputStream("test.txt"));

## 适配器、装饰者、代理模式的区别

适配器模式，一个适配允许通常因为接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包裹在一个已存在的类中。

装饰器模式，原有的不能满足现有的需求，对原有的进行增强。

代理模式，同一个类而去调用另一个类的方法，不对这个方法进行直接操作。

在使用适配器模式的时候，我们必须同时持有原对象，适配对象，目标对象。

装饰器模式特点在于增强，他的特点是被装饰类和所有的装饰类必须实现同一个接口，而且必须持有被装饰的对象，可以无限装饰。

代理模式的特点在于隔离，隔离调用类和被调用类的关系，通过一个代理类去调用。

总的来说就是如下三句话：

 1 适配器模式是将一个类(a)通过某种方式转换成另一个类(b).  
 2 装饰模式是在一个原有类(a)的基础之上增加了某些新的功能变成另一个类(b).  
 3 代理模式是将一个类(a)转换成具体的操作类(b).

# Java基础知识

## JavaWeb监听器

### JavaWeb监听器概述

在JavaWeb被监听的事件源为：ServletContext、HttpSession、ServletRequest，即三大域对象。

* 监听域对象“创建”与“销毁”的监听器；
* 监听域对象“操作域属性”的监听器；
* 监听HttpSession的监听器。

### 创建与销毁对象的监听器--生命周期监听

**创建与销毁监听器**一共有三个：

1. ServletContextListener：Tomcat启动和关闭时调用下面两个方法

Ø  public void contextInitialized(ServletContextEvent evt)：ServletContext对象被创建后调用；

Ø  public void contextDestroyed(ServletContextEvent evt)：ServletContext对象被销毁前调用；

1. HttpSessionListener：开始会话和结束会话时调用下面两个方法

Ø  public void sessionCreated(HttpSessionEvent evt)：HttpSession对象被创建后调用；

Ø  public void sessionDestroyed(HttpSessionEvent evt)：HttpSession对象被销毁前调用；

1. ServletRequestListener：开始请求和结束请求时调用下面两个方法

Ø  public void requestInitiallized(ServletRequestEvent evt)：ServletRequest对象被创建后调用；

Ø  public void requestDestroyed(ServletRequestEvent evt)：ServletRequest对象被销毁前调用。

**事件对象**

1. ServletContextEvent：ServletContextgetServletContext()；
2. HttpSeessionEvent：HttpSessiongetSession()；
3. ServletRequestEvent：

Ø  ServletRequest getServletRequest()

Ø  ServletContext getServletContext()

**编写测试：**

1. 编写MyServletContextListener类，实现ServletContextListener接口；
2. 在web.xml文件中部署监听器；

public class MyServletRequestListener implements ServletRequestListener {  
 @Override  
 public void requestInitialized(ServletRequestEvent servletRequestEvent) {  
 System.*out*.println("---------------------------------ServletRequestListener启动了。。。request请求时执行----------------------");  
 }  
  
 @Override  
 public void requestDestroyed(ServletRequestEvent servletRequestEvent) {  
 System.*out*.println("--------------------------------ServletRequestListener停止了。。。响应完毕后执行对应的销毁方法------------------");  
 }  
}

web.xml文件：

<listener>  
 <description>监听ServletRequest的变化:ServletRequestListener</description>  
 <listener-class>com.esther.code.web.context.MyServletRequestListener</listener-class>  
</listener>

### 操作域属性的监听器--属性监听

当对域属性进行增、删、改时，执行的监听器一共有三个：

1. ServletContextAttributeListener：在ServletContext域进行增、删、改属性时调用下面方法。

Ø  public void attributeAdded(ServletContextAttributeEvent evt)

Ø  public void attributeRemoved(ServletContextAttributeEvent evt)

Ø  public void attributeReplaced(ServletContextAttributeEvent evt)

1. HttpSessionAttributeListener：在HttpSession域进行增、删、改属性时调用下面方法

Ø  public void attributeAdded(HttpSessionBindingEvent evt)

Ø  public void attributeRemoved (HttpSessionBindingEvent evt)

Ø  public void attributeReplaced (HttpSessionBindingEvent evt)

1. ServletRequestAttributeListener：在ServletRequest域进行增、删、改属性时调用下面方法

Ø  public void attributeAdded(ServletRequestAttributeEvent evt)

Ø  public void attributeRemoved (ServletRequestAttributeEvent evt)

Ø  public void attributeReplaced (ServletRequestAttributeEvent evt)

下面对这三个监听器的事件对象功能进行介绍：

1. ServletContextAttributeEvent

Ø  String getName()：获取当前操作的属性名；

Ø  Object getValue()：获取当前操作的属性值；

Ø  ServletContext getServletContext()：获取ServletContext对象。

1. HttpSessionBindingEvent

Ø  String getName()：获取当前操作的属性名；

Ø  Object getValue()：获取当前操作的属性值；

Ø  HttpSession getSession()：获取当前操作的session对象。

1. ServletRequestAttributeEvent

Ø  String getName()：获取当前操作的属性名；

Ø  Object getValue()：获取当前操作的属性值；

Ø  ServletContext getServletContext()：获取ServletContext对象；

Ø  ServletRequest getServletRequest()：获取当前操作的ServletRequest对象。

public class MyServletRequestAttributeListener implements ServletRequestAttributeListener {  
 @Override  
 public void attributeAdded(ServletRequestAttributeEvent servletRequestAttributeEvent) {  
 System.*out*.println(MessageFormat.*format*("ServletRequest域对象中添加了属性:{0}，属性值是:{1}",  
 servletRequestAttributeEvent.getName(), servletRequestAttributeEvent.getValue()));  
 }  
  
 @Override  
 public void attributeRemoved(ServletRequestAttributeEvent servletRequestAttributeEvent) {  
 System.*out*.println(MessageFormat.*format*("ServletRequest域对象中删除了属性:{0}，属性值是:{1}",  
 servletRequestAttributeEvent.getName(), servletRequestAttributeEvent.getValue()));  
 }  
  
 @Override  
 public void attributeReplaced(ServletRequestAttributeEvent servletRequestAttributeEvent) {  
 System.*out*.println(MessageFormat.*format*("ServletRequest域对象中替换了属性:{0}，属性值是:{1}",  
 servletRequestAttributeEvent.getName(), servletRequestAttributeEvent.getValue()));  
 }  
}

### HttpSession的监听器--感知监听

* 它用来添加到JavaBean上，而不是添加到三大域上！即让Bean类实现监听器接口，然后再把Bean对象添加到session域中。
* 这两个监听器都不需要在web.xml中注册！

#### HttpSessionBindingListener

当某个类实现了该接口后，可以感知本类对象添加到session中，以及感知从session中移除。例如让Person类实现HttpSessionBindingListener接口，那么当把Person对象添加到session中，或者把Person对象从session中移除时会调用下面两个方法：

Ø  public void valueBound([HttpSessionBindingEvent](https://blog.csdn.net/dzy21/article/details/51979035" \o "class in javax.servlet.http" \t "https://blog.csdn.net/dzy21/article/details/_blank) event)：当把监听器对象添加到session中会调用监听器对象的本方法；

Ø  public void valueUnbound([HttpSessionBindingEvent](https://blog.csdn.net/dzy21/article/details/51979035" \o "class in javax.servlet.http" \t "https://blog.csdn.net/dzy21/article/details/_blank) event)：当把监听器对象从session中移除时会调用监听器对象的本方法；

*/\*\*  
 \** ***@author*** *esther  
 \* 2018-05-21 16:25  
 \* HttpSessionBindingListener监听器，专门用于监听JavaBean对应在HttpSession中的状态情况，只需要在JavaBean中对其实现即可，不需要再xml中配置.  
 \*  
 \* \*实现了HttpSessionBindingListener接口的 JavaBean对象可以感知自己被绑定到 Session中和从Session中删除的事件。  
 \* 当对象被绑定到 HttpSession 对象中时，web 服务器调用该对象的 void valueBound(HttpSessionBindingEvent event) 方法  
 \* 当对象从 HttpSession 对象中解除绑定时，web 服务器调用该对象的 void valueUnbound(HttpSessionBindingEvent event)方法  
 \*/*public class TestBindingBean implements HttpSessionBindingListener {  
 private String name;  
 @Override  
 public void valueBound(HttpSessionBindingEvent httpSessionBindingEvent) {  
 System.*out*.println("-------------------------------" + name + "被加到session中了");  
 }  
 @Override  
 public void valueUnbound(HttpSessionBindingEvent httpSessionBindingEvent) {  
 System.*out*.println("-------------------------------" + name + "被session踢出来了");  
 }  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public TestBindingBean(String name) {  
 this.name = name;  
 }  
}

#### HttpSessionActivationListener

Tomcat会在session不被使用时钝化session对象，所谓钝化session，就是把session通过序列化的方式保存到硬盘文件中。当用户再使用session时，Tomcat还会把钝化的对象再活化session，所谓活化就是把硬盘文件中的session在反序列化回内存。当session被Tomcat钝化时，session中存储的对象也被纯化，当session被活化时，也会把session中存储的对象活化。如果某个类实现了HttpSessionActiveationListener接口后，当对象随着session被钝化和活化时，下面两个方法就会被调用：

Ø  public void sessionWillPassivate([HttpSessionEvent](https://blog.csdn.net/dzy21/article/details/51979035" \o "class in javax.servlet.http" \t "https://blog.csdn.net/dzy21/article/details/_blank) se)：当对象感知被钝化时调用本方法；

Ø  public void sessionDidActivate([HttpSessionEvent](https://blog.csdn.net/dzy21/article/details/51979035" \o "class in javax.servlet.http" \t "https://blog.csdn.net/dzy21/article/details/_blank) se)：当对象感知被活化时调用本方法；

HttpSessionActivationListener监听器与HttpSessionBindingListener监听器相似，都是感知型的监听器，例如让Person类实现了HttpSessionActivationListener监听器接口，并把Person对象添加到了session中后，当Tomcat钝化session时，同时也会钝化session中的Person对象，这时Person对象就会感知到自己被钝化了，其实就是调用Person对象的sessionWillPassivate()方法。当用户再次使用session时，Tomcat会活化session，这时Person会感知到自己被活化，其实就是调用Person对象的sessionDidActivate()方法。

注意，因为钝化和活化session，其实就是使用序列化和反序列化技术把session从内存保存到硬盘，和把session从硬盘加载到内存。这说明如果Person类没有实现Serializable接口，那么当session钝化时就不会钝化Person，而是把Person从session中移除再钝化！这也说明session活化后，session中就不在有Person对象了。

在Webappt\META-INF文件夹下创建一个context.xml文件：

<?xml version="1.0" encoding="utf-8"?>  
<Context>  
 <!--其中的maxIdleSwap是钝化的时间，1分钟；  
 directory是钝化和激活的目录，只写mysession的话，就默认在tomacat的work\Catalina\localhost下产生itcast文件夹，并把session放在其中 C:\Users\201708015\.IntelliJIdea2017.2\system\tomcat\Unnamed\_kingdom\work\Catalina\localhost\kingdom\mysession -->  
 <Manager className="org.apache.catalina.session.PersistentManager" maxIdleSwap="1">  
 <Store className="org.apache.catalina.session.FileStore" directory="mysession"/>  
 </Manager>  
</Context>

等待一分钟，这时session会被钝化，也就会调用Person的sessionWillPassivate()；

刷新一下页面，这会使session活化，会调用Person的sessionDidActivate()方法。

# Spring基础知识

## 介绍

在java开发领域，Spring相对于EJB来说是一种轻量级的，非侵入性的Java开发框架，曾经有两本很畅销的书《Expert one-on-one J2EE Design and Development》和《Expert one-on-one J2EEdevelopment without EJB》是java高手进阶必看的宝典，Spring就是从这两本书的理论发展起来的。

Spring主要核心是：

(1).控制反转(IOC)：以前传统的java开发模式中，当需要一个对象时我们，我们会自己使用new或者getInstance等直接或者间接调用构造方法创建一个对象，而在Spring开发模式中，Spring容器使用了工厂模式为我们创建了所需要的对象，我们使用时不需要自己去创建，直接调用Spring为我们提供的对象即可，这就是控制反转的思想。实例化一个java对象有三种方式：使用类构造器，使用静态工厂方法，使用实例工厂方法，当使用spring时我们就不需要关心通过何种方式实例化一个对象，spring通过控制反转机制自动为我们实例化一个对象。

(2).依赖注入(DI)：Spring使用java Bean对象的Set方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程就是依赖注入的基本思想。

(3).面向切面编程(AOP)：在面向对象编程(OOP)思想中，我们将事物纵向抽象成一个个的对象。而在面向切面编程中，我们将一个个对象某些类似的方面横向抽象成一个切面，对这个切面进行一些如权限验证，事物管理，记录日志等公用操作处理的过程就是面向切面编程的思想。

在Spring中，所有管理的对象都是JavaBean对象，而BeanFactory和ApplicationContext就是spring框架的两个IOC容器，现在一般使用ApplicationnContext，其不但包含了BeanFactory的作用，同时还进行更多的扩展。

实例化Spring IOC容器的简单方法：

(1).方法1：

//如果只有一个spring配置文件可以直接传递一个string参数，不需要使用数组

ApplicationContext context = new ClassPathXmlApplicationContext(new String[]{“spring配置文件路径”});

(2).方法2：

Resource res = new FileSystemResource(“spring配置文件”);

BeanFactory factory = new XMLBeanFactory(res);

## Spring多个配置文件组合方法

很多时候，由于Spring需要管理和配置的东西比较多，如果都放在一个配置文件中，配置文件会变的比较大，同时不方便与维护，一般好的做法是按照功能模块将Spring配置文件分开，例如：DAO层配置到一个spring-dao.xml配置文件中，Service层配置到spring-service.xml文件中，Struts的action配置到spring-action.xml文件中，然后通过下面两种办法将这些分散的配置文件组合起来：

(1).方法1：在一个作为Spring总配置文件中的<bean>元素定义之前，通过<import>元素将要引入的spring其他配置文件引入，例如：

<beans>

       <import resource=”spring-dao.xml”/>

       <import resource=”spring-service.xml”/>

       <import resource=”spring-action.xml”/>

       ……

       <bean>

       </bean>

       ……

</beans>

(2).方法2：

a.对于JavaSE的工程，当使用下面方法获取ApplicationContext对象时将所有的spring配置文件通过数组传递进去，也可以使用通配符如spring-\*.xml方式。

ApplicationContext context = new ClassPathXmlApplicationContext(new String[]{“spring配置文件路径”});

b.对于JavaEE工程，在web.xml文件中指定spring配置文件时可以指定多个，中间有逗号“，”分隔，也可以使用通配符方式。

## Spring配置文件bean的配置规则

默认情况下，allowBeanDefinitionOverriding 属性为 null。如果在同一配置文件中 Bean id 或 name 重复了，会抛错，但是如果不是同一配置文件中，会发生覆盖。

(1).一个Bean可以通过一个id属性惟一指定和引用，如果spring配置文件中有两个以上相同的id时，spring会报错id冲突。

(2).一个Bean也可以通过一个name属性来引用和指定，如果spring配置文件中有两个以上相同name的Bean，则spring通过name引用时，运行时后面的会自动覆盖前面相同name的bean引用，而不会报错。

## Spring依赖注入3种方式

对于spring配置一个bean时，如果需要给该bean提供一些初始化参数，则需要通过依赖注入方式，所谓的依赖注入就是通过spring将bean所需要的一些参数传递到bean实例对象的过程，spring的依赖注入有3种方式：

(1).使用构造器注入：

<bean id=”……” class=”……”>

       <constructor-arg>构造函数需要的参数1</constructor-arg>

       <constructor-arg>构造函数需要的参数2</constructor-arg>

<!-- 可以用下面三种形式指定构造参数 -->

<constructor-arg type="int" value="1"/>

<constructor-arg name="name" value="parent"/>

<constructor-arg index="2" value="10000"/>

       ……

</bean>

使用构造器注入方式时：spring在实例化该Bean时会调用配置参数符合的构造方法。

(2).使用属性的setter注入：

<bean id=”……” class=”……”>

       <property name=”属性1” value=”……”/>

       <property name=”属性2” value=”……”/>

       ……

</bean>

使用属性的setter注入方式时，所注入的属性必须提供setter和getter方法，spring在实例化时会自动调用无参数的构造方法或者静态工厂方法，实例化之后自动调用属性的set方法将值设置进去。

(3).使用Field字段注入方式，该方式是spring2.5以后提供的基于java 注解(annotation)方式的注入，就是在java中的字段上或者setter方法上通过使用注解方式进行spring的依赖注入。

例如：

a．字段注入：

@Resource

private  UserDao dao;

b. 属性注入：

@Resource

public void setUserDao(UserDao dao){

       this.dao = dao;

}

## Spring通过注入方式注入依赖bean的方法

Spring中为一个bean注入其依赖的另一个bean时，通过使用ref来注入另一个bean，简单例子如下：

如果bean2中通过bean1属性引用了bean1则通过下面方式注入：

<beans>

       <bean id=”bean1” class=”……”>

              <property name=”属性1” value=”……”/>

              ……

       </bean>

<bean id=”bean2” class=”……”>

              <property name=”bean1” ref=”bean1”/>

<property name="child">

<ref bean="child"/>

</property>

              ……

       </bean>

</beans>

## Spring的集合注入

当需要给Bean的集合引用注入值时，spring也提供了相应的方法：

(1).Set集合注入：

<bean id=”……” class=”……”>

<set>

              <value>value1</value>

              <value>value2</value>

              ……

       </set>

</bean>

(2).List集合注入：

<bean id=”……” class=”……”>

<list>

              <value>value1</value>

              <value>value2</value>

              ……

       </list>

</bean>

(3).Map集合注入：

<bean id=”……” class=”……”>

<map>

              <entry key=”key1” value=”value1”>

              <entry key=”key2” value=”value2”>

              ……

       </map>

</bean>

(4).Properties注入：

<bean id=”……” class=”……”>

<props>

              <prop key=”key1”>value1</prop>

              <prop key=”key2”>value2</prop>

              ……

       </props>

</bean>

注意：Spring会自动将其数据类型转换，支持泛型。

## Java注解(Annotation)简单介绍

Java注解(Annotation)是通过在java源文件上添加标记字段，然后通过反射在编译或者运行时获取这些标记字段以及其标记目标，然后进行相应处理的方法，曾经风靡一时的XDoclet就是最早的java注解的工作原理说明。使用注解可以进行某些配置，大大减少xml配置文件的书写麻烦，但是也有人认为注解方式不方便理解和维护，因此对于注解VS XML配置文件一直争论不断，个人感觉没有好坏之分，合适就是最好的。

(1)、JDK内置注解：

JDK内置了3个注解：

a.      @Override：声明某个方法被重写。

b.      @Deprectated：声明某个方法过时，不推荐使用。

c.      @SuppressWarning({“unchecked”,…….})：告诉编译器对于那些警告信息忽略。

(2)、自定义java注解：

Java注解就是一种特殊的接口，自定java注解非常方便，简单例子如下：

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.CLASS)

public @interface TestAnnotation{

String value() default “”;

}

(3)、若要想自定义的注解可以被继承，则需要在自定义注解类上加上@Inherited注解。

注意和说明：

* java的注解实际上是自动继承了java.lang.annotation.Annotation接口，因此在自定义注解时不能继承其他的注解或者接口。
* Retention：告诉编译器如何处理注解，即注解运行在什么时刻。

RetentionPolicy是个枚举类型，有以下三种状态值：

1).SOURCE：该注解存储在源文件中，编译过后即废弃。

2).CLASS：该注解存储在class文件中，是其缺省的值。

3).RUNTIME：该注解存储在class文件中，并且有Java虚拟机读入解析，该类型非常重要，可以使用反射机制获取注解相关信息，可以进行程序分析处理。

* @Target：指定注解的目标使用对象。

ElementType也是个枚举类型，有以下几种状态值：

1).TYPE：该注解适用于class,interface,enum等类级别的目标对象。

2).FIELD：该注解适用于类中字段级别的目标对象。

3).METHOD：该注解适用于类中方法级别的目标对象。

4).PARAMETER：该注解适用于方法参数级别的目标对象。

5).CONSTRUCTOR：该注解适用于类构造方法。

6).LOCAL\_VARIABLE：该注解适用于局部变量。

7).ANNOTATION\_TYPE：该注解适用于annotation对象。

8).PACKAGE：该注解适用于package对象。

* 注解里面只能声明属性，不能声明方法，声明属性的方式比较特殊：

语法格式为：数据类型 属性() default 默认值(默认值是可选的); 如：Stringvalue();

使用时，“注解对象(属性=属性值)”为注解指定属性值，通过“注解对象.属性;”就可以得到注解属性的值。

* 注解的解析：使用java反射机制获得注解的目标对象就可以得到注解对象，如：通过反射得到了注解目标的Field对象field，则通过“field.getAnnotation(注解类.class);”就可以得到注解对象。

## 基于注解的Spring配置准备条件

从Spring2.5以后，Spring开始全面支持注解方式配置，所以可以不用写xml配置文件，一样可以使用spring。

* 使用spring注解方式，必须加入spring注解方式所依赖的jar包：common-annotation.jar。
* 使用注解方式时，必须在spring配置文件的schema中添加注解的命名空间如下：

xmlns:context=”http://www.springframework.org/schema/context”

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd

* 在spring配置文件中注册注解处理器：

在spring配置文件中<bean>元素之前添加：

<context:annotation-config>

## 基于注解的spring配置

Spring2.5中使用四个注解按功能来进行对bean的配置：

* @Component：泛指组件，对于一般不好归类的java Bean使用。
* @Service：用于标注配置业务层(service层)组件。
* @Controller：用于标注配置控制层的组件(如Struts中的action)。
* @Repository：用于标注配置数据访问层组件，即一般的DAO层Bean对象。

注意：对于使用spring注解方式配置的bean对象，bean引用时默认名称为被注解名称的首字母小写形式，也可以指定名称，如：@Service(“userDao“)。

为什么 @Repository 只能标注在 DAO 类上呢？这是因为该注解的作用不只是将类识别为 Bean，同时它还能将所标注的类中抛出的数据访问异常封装为 Spring 的数据访问异常类型。 Spring 本身提供了一个丰富的并且是与具体的数据访问技术无关的数据访问异常结构，用于封装不同的持久层框架抛出的异常，使得异常独立于底层的框架。

## 额外的初始化和销毁

Spring Bean 是受 Spring IoC 容器管理，由容器进行初始化和销毁的（prototype 类型由容器初始化之后便不受容器管理），通常我们不需要关注容器对 Bean 的初始化和销毁操作，由 Spring 经过构造函数或者工厂方法创建的 Bean 就是已经初始化完成并立即可用的。然而在某些情况下，可能需要我们手工做一些额外的初始化或者销毁操作，这通常是针对一些资源的获取和释放操作。Spring 1.x 为此提供了两种方式供用户指定执行生命周期回调的方法。

第一种方式：

是实现 Spring 提供的两个接口：InitializingBean 和 DisposableBean。如果希望在 Bean 初始化完成之后执行一些自定义操作，则可以让 Bean 实现 InitializingBean 接口，该接口包含一个 afterPropertiesSet() 方法，容器在为该 Bean 设置了属性之后，将自动调用该方法；如果 Bean 实现了 DisposableBean 接口，则容器在销毁该 Bean 之前，将调用该接口的 destroy() 方法。这种方式的缺点是，让 Bean 类实现 Spring 提供的接口，增加了代码与 Spring 框架的耦合度，因此不推荐使用。

第二种方式是：

在 XML 文件中使用 <bean> 的 init-method 和 destroy-method 属性指定初始化之后和销毁之前的回调方法，代码无需实现任何接口。这两个属性的取值是相应 Bean 类中的初始化和销毁方法，方法名任意，但是方法不能有参数。示例如下：

<bean id=”userService”

class=”bookstore.service.UserService”

init-method=”init” destroy-method=”destroy”>

   …

</bean>

第三种方式：

Spring 2.5 在保留以上两种方式的基础上，提供了对 JSR-250 的支持。JSR-250 规范定义了两个用于指定声明周期方法的注解：@PostConstruct 和 @PreDestroy。这两个注解使用非常简单，只需分别将他们标注于初始化之后执行的回调方法或者销毁之前执行的回调方法上。由于使用了注解，因此需要配置相应的 Bean 后处理器，亦即在 XML 中增加如下一行：

<context:annotation-config />

比较上述三种指定生命周期回调方法的方式，第一种是不建议使用的，不但其用法不如后两种方式灵活，而且无形中增加了代码与框架的耦合度。后面两种方式开发者可以根据使用习惯选择其中一种，但是最好不要混合使用，以免增加维护的难度。

## 使用 @Required 进行 Bean 的依赖检查

依赖检查的作用是，判断给定 Bean 的相应 Setter 方法是否都在实例化的时候被调用了。而不是判断字段是否已经存在值了。Spring 进行依赖检查时，只会判断属性是否使用了 Setter 注入。如果某个属性没有使用 Setter 注入，即使是通过构造函数已经为该属性注入了值，Spring 仍然认为它没有执行注入，从而抛出异常。另外，Spring 只管是否通过 Setter 执行了注入，而对注入的值却没有任何要求，即使注入的 <null/>，Spring 也认为是执行了依赖注入。

<bean> 标签提供了 dependency-check 属性用于进行依赖检查。该属性的取值包括以下几种：

none -- 默认不执行依赖检查。可以在 <beans> 标签上使用 default-dependency-check 属性改变默认值。

simple -- 对原始基本类型和集合类型进行检查。

objects -- 对复杂类型进行检查（除了 simple 所检查类型之外的其他类型）。

all -- 对所有类型进行检查。

旧版本使用 dependency-check 在配置文件中设置，缺点是粒度较粗。使用 Spring2.0 提供的 @Required 注解，提供了更细粒度的控制。@Required 注解只能标注在 Setter 方法之上。因为依赖注入的本质是检查 Setter 方法是否被调用了，而不是真的去检查属性是否赋值了以及赋了什么样的值。如果将该注解标注在非 setXxxx() 类型的方法则被忽略。

为了让 Spring 能够处理该注解，需要激活相应的 Bean 后处理器。要激活该后处理器，只需在 XML 中增加如下一行即可：

<context:annotation-config/>

当某个被标注了 @Required 的 Setter 方法没有被调用，则 Spring 在解析的时候会抛出异常，以提醒开发者对相应属性进行设置。

## Spring的自动装配

自动装配的是指不用手动显式配置java Bean之间依赖关系，而是让spring依据某种规则自动将合适的对象注入到目标对象的过程。Spring中常用@Autowire和@Resource来进行自动装配。

* @Autowire：默认是按照对象的数据类型进行自动装配的，如

@Autowire

private UserDao userDao;

spring框架在运行时会自动将类型为UserDao的对象注入进来。

* @Resource：默认是按照名称或者Id进行自动装配的，只有当找不到匹配的名称或者Id时才按类型进行装配，如：

@Resource(name=”userDao”)

private UsreDao userDao;

spring框架在运行时会将配置名称或者id为“userDao”的对象注入进来。

注意：@Autowire和@Resource都既可以写在字段上，也可以写在set方法上。

### 使用 @Resource、@Autowired 和 @Qualifier 指定 Bean 的自动装配策略

自动装配是指，Spring 在装配 Bean 的时候，根据指定的自动装配规则，将某个 Bean 所需要引用类型的 Bean 注入进来。<bean> 元素提供了一个指定自动装配类型的 autowire 属性，该属性有如下选项：

1. no -- 显式指定不使用自动装配。
2. byName -- 如果存在一个和当前属性名字一致的 Bean，则使用该 Bean 进行注入。如果名称匹配但是类型不匹配，则抛出异常。如果没有匹配的类型，则什么也不做。
3. byType -- 如果存在一个和当前属性类型一致的 Bean ( 相同类型或者子类型 )，则使用该 Bean 进行注入。byType 能够识别工厂方法，即能够识别 factory-method 的返回类型。如果存在多个类型一致的 Bean，则抛出异常。如果没有匹配的类型，则什么也不做。
4. constructor -- 与 byType 类似，只不过它是针对构造函数注入而言的。如果当前没有与构造函数的参数类型匹配的 Bean，则抛出异常。使用该种装配模式时，优先匹配参数最多的构造函数。
5. autodetect -- 根据 Bean 的自省机制决定采用 byType 还是 constructor 进行自动装配。如果 Bean 提供了默认的构造函数，则采用 byType；否则采用 constructor 进行自动装配。

当使用 byType 或者 constructor 类型的自动装配的时候，自动装配也支持引用类型的数组或者使用了泛型的集合，这样，Spring 就会检查容器中所有类型匹配的 Bean，组成集合或者数组后执行注入。对于使用了泛型的 Map 类型，如果键是 String 类型，则 Spring 也会自动执行装配，将所有类型匹配的 Bean 作为值，Bean 的名字作为键。

我们可以给 <beans> 增加 default-autowire 属性，设置默认的自动封装策略。默认值为"no"。如果使用自动装配的同时，也指定了 property 或者 constructor-arg 标签，则显式指定的值将覆盖自动装配的值。目前的自动封装不支持简单类型，比如基本类型、String、Class，以及它们的数组类型。

在按类型匹配的时候 ( 可能是 byType、constructor、autodetect)，同一个类型可能存在多个 Bean，如果被注入的属性是数组、集合或者 Map，这可能没有问题，但是如果只是简单的引用类型，则会抛出异常。解决方法有如下几种：

1. 取消该 Bean 的自动装配特性，使用显式的注入。我们可能不希望某个 Bean 被当作其他 Bean 执行自动封装时的候选对象，我们可以给该 <bean> 增加 autowire-candidate="false"。(autowire-candidate 属性和 autowire 属性相互独立，互不相干。某个 Bean 可以将 autowire-candidate 设置为 false，同时使用 autowire 特性。) 另外，我们可以设置 <beans> 的 default-autowire-candidates 属性，可以在该属性中指定可以用于自动装配候选 Bean 的匹配模式，比如 default-autowire-candidates="\*serv,\*dao"，这表示所有名字以 serv 或者 dao 结尾的 Bean 被列为候选，其他则忽略，相当于其他 Bean 都指定为 autowire-candidate="false"，此时可以显式为 <bean> 指定 autowire-candidate="true"。在 <bean> 上指定的设置要覆盖 <beans> 上指定的设置。
2. 如果在多个类型相同的 Bean 中有首选的 Bean，那么可以将该 <bean> 的 primary 属性设置为 "true" ，这样自动装配时便优先使用该 Bean 进行装配。此时不能将 autowire-candidate 设为 false。
3. 如果使用的是 Java 5 以上版本，可以使用注解进行更细粒度的控制。

### 使用@Autowired和@Qualifier注解执行自动装配

使用 @Autowired 注解进行装配，只能是根据类型进行匹配。@Autowired 注解可以用于 Setter 方法、构造函数、字段，甚至普通方法，前提是方法必须有至少一个参数。@Autowired 可以用于数组和使用泛型的集合类型。然后 Spring 会将容器中所有类型符合的 Bean 注入进来。@Autowired 标注作用于 Map 类型时，如果 Map 的 key 为 String 类型，则 Spring 会将容器中所有类型符合 Map 的 value 对应的类型的 Bean 增加进来，用 Bean 的 id 或 name 作为 Map 的 key。

@Autowired 标注作用于普通方法时，会产生一个副作用，就是在容器初始化该 Bean 实例的时候就会调用该方法。当然，前提是执行了自动装配，对于不满足装配条件的情况，该方法也不会被执行。

当标注了 @Autowired 后，自动注入不能满足，则会抛出异常。我们可以给 @Autowired 标注增加一个 required=false 属性，以改变这个行为。另外，每一个类中只能有一个构造函数的 @Autowired.required() 属性为 true。否则就出问题了。如果用 @Autowired 同时标注了多个构造函数，那么，Spring 将采用贪心算法匹配构造函数 ( 构造函数最长 )。

@Autowired 还有一个作用就是，如果将其标注在 BeanFactory 类型、ApplicationContext 类型、ResourceLoader 类型、ApplicationEventPublisher 类型、MessageSource 类型上，那么 Spring 会自动注入这些实现类的实例，不需要额外的操作。

当容器中存在多个 Bean 的类型与需要注入的相同时，注入将不能执行，我们可以给 @Autowired 增加一个候选值，做法是在 @Autowired 后面增加一个 @Qualifier标注，提供一个 String 类型的值作为候选的Bean 的名字。举例如下：

@Autowired(required=false)

@Qualifier("ppp")

public void setPerson(person p){}

@Qualifier 甚至可以作用于方法的参数 ( 对于方法只有一个参数的情况，我们可以将 @Qualifer 标注放置在方法声明上面，但是推荐放置在参数前面 )，举例如下：

@Autowired(required=false)

 public void sayHello(@Qualifier("ppp")Person p,String name){}

我们可以在配置文件中指定某个 Bean 的 qualifier 名字，方法如下：

<bean id="person" class="footmark.spring.Person">

   <qualifier value="ppp"/>

</bean>

如果没有明确指定Bean的qualifier名字，那么默认名字就是 Bean 的名字。 通常，qualifier 应该是有业务含义的，例如 "domain"，"persistent" 等，而不应该是类似 "person" 方式。

我们还可以将 @Qualifier 标注在集合类型上，那么所有 qualifier 名字与指定值相同的 Bean 都将被注入进来。

最后，配置文件中需要指定每一个自定义注解的属性值。我们可以使用 <meta> 标签来代替 <qualifier/> 标签，如果 <meta> 标签和 <qualifier/> 标签同时出现，那么优先使用 <qualifier> 标签。如果没有 <qualifier> 标签，那么会用 <meta> 提供的键值对来封装 <qualifier> 标签。示例如下：

<bean class="footmark.HelloWorld">

<qualifier type="MovieQualifier">

<attribute key="format" value="VHS"/>

<attribute key="genre" value="Comedy"/>

</qualifier>

</bean>

<bean class="footmark.HelloWorld">

<meta key="format" value="DVD"/>

<meta key="genre" value="Action"/>

</bean>

@Autowired 注解对应的后处理注册与前面相似，只需在配置文件中增加如下一行即可：

<context:annotation-config/>

如果 @Autowired 注入的是 BeanFactory、ApplicationContext、ResourceLoader 等系统类型，那么则不需要 @Qualifier，此时即使提供了 @Qualifier 注解，也将会被忽略；而对于自定义类型的自动装配，如果使用了 @Qualifier 注解并且没有名字与之匹配的 Bean，则自动装配匹配失败。

### 使用@Resource和@Qualifier注解执行自动装配

如果希望根据 name 执行自动装配，那么应该使用 JSR-250 提供的 @Resource 注解，而不应该使用 @Autowired 与 @Qualifier 的组合。

@Resource 使用 byName 的方式执行自动封装。@Resource 标注可以作用于带一个参数的 Setter 方法、字段，以及带一个参数的普通方法上。@Resource 注解有一个 name 属性，用于指定 Bean 在配置文件中对应的名字。如果没有指定 name 属性，那么默认值就是字段或者属性的名字。@Resource 和 @Qualifier 的配合虽然仍然成立，但是 @Qualifier 对于 @Resource 而言，几乎与 name 属性等效。

如果 @Resource 没有指定 name 属性，那么使用 byName 匹配失败后，会退而使用 byType 继续匹配，如果再失败，则抛出异常。在没有为 @Resource 注解显式指定 name 属性的前提下，如果将其标注在 BeanFactory 类型、ApplicationContext 类型、ResourceLoader 类型、ApplicationEventPublisher 类型、MessageSource 类型上，那么 Spring 会自动注入这些实现类的实例，不需要额外的操作。此时 name 属性不需要指定 ( 或者指定为"")，否则注入失败；如果使用了 @Qualifier，则该注解将被忽略。而对于用户自定义类型的注入，@Qualifier 和 name 等价，并且不被忽略。

<bean> 的 primary 和 autowire-candidate 属性对 @Resource、@Autowired 仍然有效。

## Spring的自动扫描

Spring自动扫描机制是指，我们使用基于注解的spring配置方式后，spring的配置文件中只需要注册注解处理器，不用显式地配置Bean，当spring在运行时会自动扫描根据给定的条件下类路径中的所有bean，根据注解将它们注入和装配起来并进行初始化的过程。

自动扫描很简单，只需要在spring的配置文件中添加如：

<context:component-scan base-package=“要自动扫描的包名“>

  Spring在运行时就可以对指定的包中所有添加了Spring注解的bean自动扫描，注入，装配和初始化。

注意：基于注解的spring配置，自动装配和自动扫描是紧密联系在一起协同工作的，都需要引入context的命名空间。

## 使用@Configuration和@Bean进行Bean的声明

虽然 2.0 版本发布以来，Spring 陆续提供了十多个注解，但是提供的这些注解只是为了在某些情况下简化 XML 的配置，并非要取代 XML 配置方式。这一点可以从 Spring IoC 容器的初始化类可以看出：ApplicationContext 接口的最常用的实现类是 ClassPathXmlApplicationContext 和 FileSystemXmlApplicationContext，以及面向 Portlet 的 XmlPortletApplicationContext 和面向 web 的 XmlWebApplicationContext，它们都是面向 XML 的。Spring 3.0 新增了另外两个实现类：AnnotationConfigApplicationContext 和 AnnotationConfigWebApplicationContext。从名字便可以看出，它们是为注解而生，直接依赖于注解作为容器配置信息来源的 IoC 容器初始化类。由于 AnnotationConfigWebApplicationContext 是 AnnotationConfigApplicationContext 的 web 版本，其用法与后者相比几乎没有什么差别，因此本文将以 AnnotationConfigApplicationContext 为例进行讲解。

AnnotationConfigApplicationContext 搭配上 @Configuration 和 @Bean 注解，自此，XML 配置方式不再是 Spring IoC 容器的唯一配置方式。两者在一定范围内存在着竞争的关系，但是它们在大多数情况下还是相互协作的关系，两者的结合使得 Spring IoC 容器的配置更简单，更强大。

之前，我们将配置信息集中写在 XML 中，如今使用注解，配置信息的载体由 XML 文件转移到了 Java 类中。我们通常将用于存放配置信息的类的类名以 “Config” 结尾，比如 AppDaoConfig.java、AppServiceConfig.java 等等。我们需要在用于指定配置信息的类上加上 @Configuration 注解，以明确指出该类是 Bean 配置的信息源。并且 Spring 对标注 Configuration 的类有如下要求：

* 配置类不能是 final 的；
* 配置类不能是本地化的，亦即不能将配置类定义在其他类的方法内部；
* 配置类必须有一个无参构造函数。

AnnotationConfigApplicationContext 将配置类中标注了 @Bean 的方法的返回值识别为 Spring Bean，并注册到容器中，受 IoC 容器管理。@Bean 的作用等价于 XML 配置中的 <bean/> 标签。示例如下：

@Configuration

 public class BookStoreDaoConfig{

    @Bean

    public UserDao userDao(){ return new UserDaoImpl();}

    @Bean

    public BookDao bookDao(){return new BookDaoImpl();}

 }

Spring 在解析到以上文件时，将识别出标注 @Bean 的所有方法，执行之，并将方法的返回值 ( 这里是 UserDaoImpl 和 BookDaoImpl 对象 ) 注册到 IoC 容器中。默认情况下，Bean 的名字即为方法名。因此，与以上配置等价的 XML 配置如下：

<bean id=”userDao” class=”bookstore.dao.UserDaoImpl”/>

<bean id=”bookDao” class=”bookstore.dao.BookDaoImpl”/>

@Bean 具有以下四个属性：

* name -- 指定一个或者多个 Bean 的名字。这等价于 XML 配置中 <bean> 的 name 属性。
* initMethod -- 容器在初始化完 Bean 之后，会调用该属性指定的方法。这等价于 XML 配置中 <bean> 的 init-method 属性。
* destroyMethod -- 该属性与 initMethod 功能相似，在容器销毁 Bean 之前，会调用该属性指定的方法。这等价于 XML 配置中 <bean> 的 destroy-method 属性。
* autowire -- 指定 Bean 属性的自动装配策略，取值是 Autowire 类型的三个静态属性。Autowire.BY\_NAME，Autowire.BY\_TYPE，Autowire.NO。与 XML 配置中的 autowire 属性的取值相比，这里少了 constructor，这是因为 constructor 在这里已经没有意义了。

@Bean 没有直接提供指定作用域的属性，可以通过 @Scope 来实现该功能，关于 @Scope 的用法已在上文列举。

下面讲解基于注解的容器初始化。AnnotationConfigApplicationContext 提供了三个构造函数用于初始化容器。

* AnnotationConfigApplicationContext()：该构造函数初始化一个空容器，容器不包含任何 Bean 信息，需要在稍后通过调用其 register() 方法注册配置类，并调用 refresh() 方法刷新容器。
* AnnotationConfigApplicationContext(Class<?>... annotatedClasses)：这是最常用的构造函数，通过将涉及到的配置类传递给该构造函数，以实现将相应配置类中的 Bean 自动注册到容器中。
* AnnotationConfigApplicationContext(String... basePackages)：该构造函数会自动扫描以给定的包及其子包下的所有类，并自动识别所有的 Spring Bean，将其注册到容器中。它不但识别标注 @Configuration 的配置类并正确解析，而且同样能识别使用 @Repository、@Service、@Controller、@Component 标注的类。

除了使用上面第三种类型的构造函数让容器自动扫描 Bean 的配置信息以外，AnnotationConfigApplicationContext 还提供了 scan() 方法，其功能与上面也类似，该方法主要用在容器初始化之后动态增加 Bean 至容器中。调用了该方法以后，通常需要立即手动调用 refresh() 刷新容器，以让变更立即生效。

需要注意的是，AnnotationConfigApplicationContext 在解析配置类时，会将配置类自身注册为一个 Bean，因为 @Configuration 注解本身定义时被 @Component 标注了。因此可以说，一个 @Configuration 同时也是一个 @Component。大多数情况下，开发者用不到该 Bean，并且在理想情况下，该 Bean 应该是对开发者透明的。@Configuration 的定义如下所示：

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Configuration {

String value() default "";

}

在一般的项目中，为了结构清晰，通常会根据软件的模块或者结构定义多个 XML 配置文件，然后再定义一个入口的配置文件，该文件使用 <import/> 将其他的配置文件组织起来。最后只需将该文件传给ClassPathXmlApplicationContext 的构造函数即可。针对基于注解的配置，Spring 也提供了类似的功能，只需定义一个入口配置类，并在该类上使用 @Import 注解引入其他的配置类即可，最后只需要将该入口类传递给 AnnotationConfigApplicationContext。具体示例如下：

@Configuration

@Import({BookStoreServiceConfig.class,BookStoreDaoConfig.class})

public class BookStoreConfig{ … }

## Spring中的几个点

### id和name

每个 Bean 在 Spring 容器中都有一个唯一的名字（beanName）和 0 个或多个别名（aliases）。

我们从 Spring 容器中获取 Bean 的时候，可以根据 beanName，也可以通过别名。

beanFactory.getBean("beanName or alias");

在配置 <bean /> 的过程中，我们可以配置 id 和 name，看几个例子就知道是怎么回事了。

<bean id="messageService" name="m1, m2, m3" class="com.javadoop.example.MessageServiceImpl">

以上配置的结果就是：beanName 为 messageService，别名有 3 个，分别为 m1、m2、m3。

<bean name="m1, m2, m3" class="com.javadoop.example.MessageServiceImpl" />

以上配置的结果就是：beanName 为 m1，别名有 2 个，分别为 m2、m3。

<bean class="com.example.MessageServiceImpl">

以上配置的结果就是：beanName为com.example.MessageServiceImpl#0，

别名 1 个，为： com.example.MessageServiceImpl

<bean id="messageService" class="com.javadoop.example.MessageServiceImpl">

以上配置的结果就是：beanName 为 messageService，没有别名。

### 配置是否允许 Bean 覆盖、是否允许循环依赖

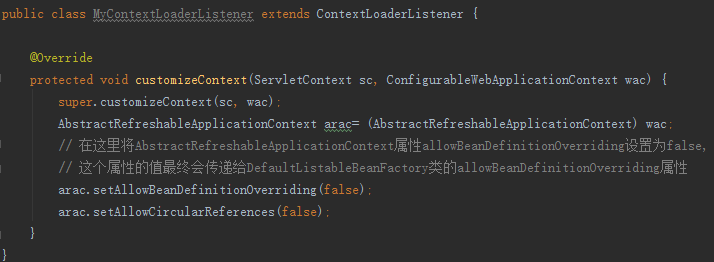
我们说过，默认情况下，allowBeanDefinitionOverriding 属性为 null。如果在同一配置文件中 Bean id 或 name 重复了，会抛错，但是如果不是同一配置文件中，会发生覆盖。

可是有些时候我们希望在系统启动的过程中就严格杜绝发生 Bean 覆盖，因为万一出现这种情况，会增加我们排查问题的成本。

循环依赖说的是 A 依赖 B，而 B 又依赖 A。或者是 A 依赖 B，B 依赖 C，而 C 却依赖 A。默认 allowCircularReferences 也是 null。

它们两个属性是一起出现的，必然可以在同一个地方一起进行配置。

添加这两个属性的作者 Juergen Hoeller 在这个 jira 的讨论中说明了怎么配置这两个属性。



<listener> <listener-class>com.esther.code.problems.allowbeanoverriding.MyContextLoaderListener</listener-class>  
</listener>

### profile

我们可以把不同环境的配置分别配置到单独的文件中，举个例子：

<beans profile="development"

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:jdbc="http://www.springframework.org/schema/jdbc"

xsi:schemaLocation="...">

<jdbc:embedded-database id="dataSource">

<jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>

<jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>

</jdbc:embedded-database>

</beans>

应该不必做过多解释了吧，看每个文件第一行的 profile=""。

当然，我们也可以在一个配置文件中使用：

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:jdbc="http://www.springframework.org/schema/jdbc"

xmlns:jee="http://www.springframework.org/schema/jee"

xsi:schemaLocation="...">

<beans profile="development">

<jdbc:embedded-database id="dataSource">

<jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>

<jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>

</jdbc:embedded-database>

</beans>

<beans profile="production">

<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>

</beans>

</beans>

理解起来也很简单吧。

接下来的问题是，怎么使用特定的 profile 呢？Spring 在启动的过程中，会去寻找 “spring.profiles.active” 的属性值，根据这个属性值来的。那怎么配置这个值呢？

Spring 会在这几个地方寻找 spring.profiles.active 的属性值：操作系统环境变量、JVM 系统变量、web.xml 中定义的参数、JNDI。

最简单的方式莫过于在程序启动的时候指定：

-Dspring.profiles.active="profile1,profile2"

profile 可以激活多个。

当然，我们也可以通过代码的形式从 Environment 中设置 profile：

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.getEnvironment().setActiveProfiles("development");

ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);

ctx.refresh(); // 重启

如果是 Spring Boot 的话更简单，我们一般会创建 application.properties、application-dev.properties、application-prod.properties 等文件，其中 application.properties 配置各个环境通用的配置，application-{profile}.properties 中配置特定环境的配置，然后在启动的时候指定 profile：

java -Dspring.profiles.active=prod -jar JavaDoop.jar

如果是单元测试中使用的话，在测试类中使用 @ActiveProfiles 指定，这里就不展开了。

### 工厂模式生成Bean

请注意 factory-bean 和 FactoryBean 的区别。这节说的是前者，是说静态工厂或实例工厂，而后者是 Spring 中的特殊接口，代表一类特殊的 Bean。

设计模式里，工厂方法模式分静态工厂和实例工厂，我们分别看看 Spring 中怎么配置这两个，来个代码示例就什么都清楚了。

静态工厂：

<**bean** id="clientService"

class="examples.ClientService"

factory-method="createInstance"/>

public class ClientService {

private static ClientService clientService = new ClientService();

private ClientService() {}

// 静态方法

public static ClientService createInstance() {

return clientService;

}

}

实例工厂：

<**bean** id="serviceLocator" class="examples.DefaultServiceLocator">

<!-- inject any dependencies required by this locator bean --></**bean**>

<**bean** id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

<**bean** id="accountService"

factory-bean="serviceLocator"

factory-method="createAccountServiceInstance"/>

public class DefaultServiceLocator {

private static ClientService clientService = new ClientServiceImpl();

private static AccountService accountService= new AccountServiceImpl();

public ClientService createClientServiceInstance() {

return clientService;

}

public AccountService createAccountServiceInstance() {

return accountService;

}

}

### FactoryBean

FactoryBean 适用于 Bean 的创建过程比较复杂的场景，比如数据库连接池的创建。是 Spring 中的特殊接口，代表一类特殊的 Bean。

public interface FactoryBean<T> {

T getObject() throws Exception;

Class<T> getObjectType();

boolean isSingleton();

}

public class Person {

private Car car ;

private void setCar(Car car){ this.car = car; }

}

我们假设现在需要创建一个 Person 的 Bean，首先我们需要一个 Car 的实例，我们这里假设 Car 的实例创建很麻烦，那么我们可以把创建 Car 的复杂过程包装起来：

public class MyCarFactoryBean implements FactoryBean<Car>{

private String make;

private int year ;

public void setMake(String m){ this.make =m ; }

public void setYear(int y){ this.year = y; }

public Car getObject(){

// 这里我们假设 Car 的实例化过程非常复杂，反正就不是几行代码可以写完的那种

CarBuilder cb = CarBuilder.car();

if(year!=0) cb.setYear(this.year);

if(StringUtils.hasText(this.make)) cb.setMake( this.make );

return cb.factory();

}

public Class<Car> getObjectType() { return Car.class ; }

public boolean isSingleton() { return false; }

}

我们看看装配的时候是怎么配置的：

<**bean** class = "com.javadoop.MyCarFactoryBean" id = "car">

<**property** name = "make" value ="Honda"/>

<**property** name = "year" value ="1984"/>

</**bean**>

<**bean** class = "com.javadoop.Person" id = "josh">

<**property** name = "car" ref = "car"/>

</**bean**>

看到不一样了吗？id 为 “car” 的 bean 其实指定的是一个 FactoryBean，不过配置的时候，我们直接让配置 Person 的 Bean 直接依赖于这个 FactoryBean 就可以了。中间的过程 Spring 已经封装好了。

说到这里，我们再来点干货。我们知道，现在还用 xml 配置 Bean 依赖的越来越少了，更多时候，我们可能会采用 java config 的方式来配置，这里有什么不一样呢？

@Configuration public class CarConfiguration {

@Bean

public MyCarFactoryBean carFactoryBean(){

MyCarFactoryBean cfb = new MyCarFactoryBean();

cfb.setMake("Honda");

cfb.setYear(1984);

return cfb;

}

@Bean

public Person aPerson(){

Person person = new Person();

// 注意这里的不同

person.setCar(carFactoryBean().getObject());

return person;

}

}

这个时候，其实我们的思路也很简单，把 MyCarFactoryBean 看成是一个简单的 Bean 就可以了，不必理会什么 FactoryBean，它是不是 FactoryBean 和我们没关系。

### 初始化Bean的回调

有以下四种方案：

<**bean** id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>

public class AnotherExampleBean implements InitializingBean {

public void afterPropertiesSet() {

// do some initialization work

}

}

@Bean(initMethod = "init")

public Foo foo() {

return new Foo();

}

@PostConstruct

public void init() {

}

### 销毁Bean的回调

<**bean** id="exampleInitBean" class="...ExampleBean" destroy-method="cleanup"/>

public class AnotherExampleBean implements DisposableBean {

public void destroy() {

// do some destruction work (like releasing pooled connections)

}

}

@Bean(destroyMethod = "cleanup")

public Bar bar() {

return new Bar();

}

@PreDestroy

public void cleanup() {

}

### ConversionService

最有用的场景就是，它用来将前端传过来的参数和后端的 controller 方法上的参数进行绑定的时候用。

像前端传过来的字符串、整数要转换为后端的 String、Integer 很容易，但是如果 controller 方法需要的是一个枚举值，或者是 Date 这些非基础类型（含基础类型包装类）值的时候，我们就可以考虑采用 ConversionService 来进行转换。

<**bean** id="conversionService"

class="org.springframework.context.support.ConversionServiceFactoryBean">

<**property** name="converters">

<**list**>

<**bean** class="com.javadoop.learning.utils.StringToEnumConverterFactory"/>

</**list**>

</**property**>

</**bean**>

ConversionService 接口很简单，所以要自定义一个 convert 的话也很简单。

下面再说一个实现这种转换很简单的方式，那就是实现 Converter 接口。

来看一个很简单的例子，这样比什么都管用。

public class StringToDateConverter implements Converter<String, Date> {

@Override

public Date convert(String source) {

try {

return DateUtils.parseDate(source, "yyyy-MM-dd", "yyyy-MM-dd HH:mm:ss", "yyyy-MM-dd HH:mm", "HH:mm:ss", "HH:mm");

} catch (ParseException e) {

return null;

}

}

}

只要注册这个 Bean 就可以了。这样，前端往后端传的时间描述字符串就很容易绑定成 Date 类型了，不需要其他任何操作。

### Bean继承

在初始化 Bean 的地方，我们说过了这个：

RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);

这里涉及到的就是 <bean parent="" /> 中的 parent 属性，我们来看看 Spring 中是用这个来干什么的。

首先，我们要明白，这里的继承和 java 语法中的继承没有任何关系，不过思路是相通的。child bean 会继承 parent bean 的所有配置，也可以覆盖一些配置，当然也可以新增额外的配置。

Spring 中提供了继承自 AbstractBeanDefinition 的 ChildBeanDefinition 来表示 child bean。

看如下一个例子:

<bean id="inheritedTestBean" abstract="true" class="org.beans.TestBean">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

<bean id="inheritsWithDifferentClass" class="org.beans.DerivedTestBean"

parent="inheritedTestBean" init-method="initialize">

<property name="name" value="override"/>

</bean>

parent bean 设置了 abstract="true" 所以它不会被实例化，child bean 继承了 parent bean 的两个属性，但是对 name 属性进行了覆写。child bean 会继承 scope、构造器参数值、属性值、init-method、destroy-method 等等。

当然，我不是说 parent bean 中的 abstract = true 在这里是必须的，只是说如果加上了以后 Spring 在实例化 singleton beans 的时候会忽略这个 bean。

比如下面这个极端 parent bean，它没有指定 class，所以毫无疑问，这个 bean 的作用就是用来充当模板用的 parent bean，此处就必须加上 abstract = true。

<bean id="inheritedTestBeanWithoutClass" abstract="true">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

### 方法注入

一般来说，我们的应用中大多数的 Bean 都是 singleton 的。singleton 依赖 singleton，或者 prototype 依赖 prototype 都很好解决，直接设置属性依赖就可以了。

但是，如果是 singleton 依赖 prototype 呢？这个时候不能用属性依赖，因为如果用属性依赖的话，我们每次其实拿到的还是第一次初始化时候的 bean。

一种解决方案就是不要用属性依赖，每次获取依赖的 bean 的时候从 BeanFactory 中取。这个也是大家最常用的方式了吧。怎么取，我就不介绍了，大部分 Spring 项目大家都会定义那么个工具类的。

另一种解决方案就是这里要介绍的通过使用lookup method。

#### **lookup-method**

我们来看一下 Spring Reference 中提供的一个例子：

package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

public Object process(Object commandState) {

// grab a new instance of the appropriate Command interface

Command command = createCommand();

// set the state on the (hopefully brand new) Command instance

command.setState(commandState);

return command.execute();

}

// okay... but where is the implementation of this method?

protected abstract Command createCommand();

}

xml 配置 <lookup-method />：

<!-- a stateful bean deployed as a prototype (non-singleton) -->

<**bean** id="myCommand" class="fiona.apple.AsyncCommand" scope="prototype">

<!-- inject dependencies here as required -->

</**bean**>

<!-- commandProcessor uses statefulCommandHelper -->

<**bean** id="commandManager" class="fiona.apple.CommandManager">

<**lookup-method** name="createCommand" bean="myCommand"/>

</**bean**>

Spring 采用 CGLIB 生成字节码的方式来生成一个子类。我们定义的类不能定义为 final class，抽象方法上也不能加 final。

lookup-method 上的配置也可以采用注解来完成，这样就可以不用配置 <lookup-method /> 了，其他不变：

public abstract class CommandManager {

public Object process(Object commandState) {

MyCommand command = createCommand();

command.setState(commandState);

return command.execute();

}

@Lookup("myCommand")

protected abstract Command createCommand();

}

注意：既然用了注解，要配置注解扫描：<context:component-scan />

甚至，我们可以像下面这样：

public abstract class CommandManager {

public Object process(Object commandState) {

MyCommand command = createCommand();

command.setState(commandState);

return command.execute();

}

@Lookup

protected abstract MyCommand createCommand();

}

注意：上面的返回值用了 MyCommand，当然，如果 Command 只有一个实现类，那返回值也可以写 Command。

#### **replaced-method**

记住它的功能，就是替换掉 bean 中的一些方法。

public class MyValueCalculator {

public String computeValue(String input) {

// some real code...

}

// some other methods...

}

方法覆写，注意要实现 MethodReplacer 接口：

public class ReplacementComputeValue implements org.springframework.beans.factory.support.MethodReplacer {

public Object reimplement(Object o, Method m, Object[] args) throws Throwable {

// get the input value, work with it, and return a computed result

String input = (String) args[0];

...

return ...;

}

}

配置也很简单：

<**bean** id="myValueCalculator" class="x.y.z.MyValueCalculator">

<!-- 定义 computeValue 这个方法要被替换掉 -->

<**replaced-method** name="computeValue" replacer="replacementComputeValue">

<**arg-type**>String</**arg-type**>

</**replaced-method**></**bean**>

<**bean** id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>

注意：arg-type 明显不是必须的，除非存在方法重载，这样必须通过参数类型列表来判断这里要覆盖哪个方法。

### BeanPostProcessor

应该说 BeanPostProcessor 概念在 Spring 中也是比较重要的。我们看下接口定义：

public interface BeanPostProcessor {

Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;

Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException;

}

看这个接口中的两个方法名字我们大体上可以猜测 bean 在初始化之前会执行 postProcessBeforeInitialization 这个方法，初始化完成之后会执行 postProcessAfterInitialization 这个方法。但是，这么理解是非常片面的。

首先，我们要明白，除了我们自己定义的 BeanPostProcessor 实现外，Spring 容器在启动时自动给我们也加了几个。如在获取 BeanFactory 的 obtainFactory() 方法结束后的 prepareBeanFactory(factory)，大家仔细看会发现，Spring 往容器中添加了这两个 BeanPostProcessor：ApplicationContextAwareProcessor、ApplicationListenerDetector。

我们回到这个接口本身，读者请看第一个方法，这个方法接受的第一个参数是 bean 实例，第二个参数是 bean 的名字，重点在返回值将会作为新的 bean 实例，所以，没事的话这里不能随便返回个 null。

那意味着什么呢？我们很容易想到的就是，我们这里可以对一些我们想要修饰的 bean 实例做一些事情。但是对于 Spring 框架来说，它会决定是不是要在这个方法中返回 bean 实例的代理，这样就有更大的想象空间了。

最后，我们说说如果我们自己定义一个 bean 实现 BeanPostProcessor 的话，它的执行时机是什么时候？

如果仔细看了代码分析的话，其实很容易知道了，在 bean 实例化完成、属性注入完成之后，会执行回调方法，具体请参见类 AbstractAutowireCapableBeanFactory#initBean 方法。

首先会回调几个实现了 Aware 接口的 bean，然后就开始回调 BeanPostProcessor 的 postProcessBeforeInitialization 方法，之后是回调 init-method，然后再回调 BeanPostProcessor 的 postProcessAfterInitialization 方法。

# Spring源码学习

## Bean加载流程

## 入口

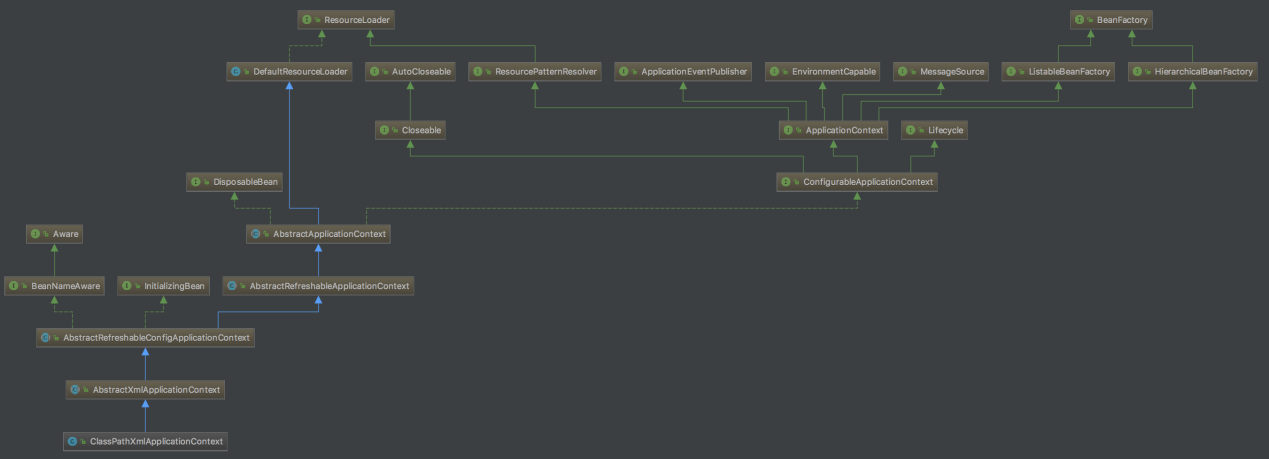
下面有很简单的一段代码可以作为Spring代码加载的入口：

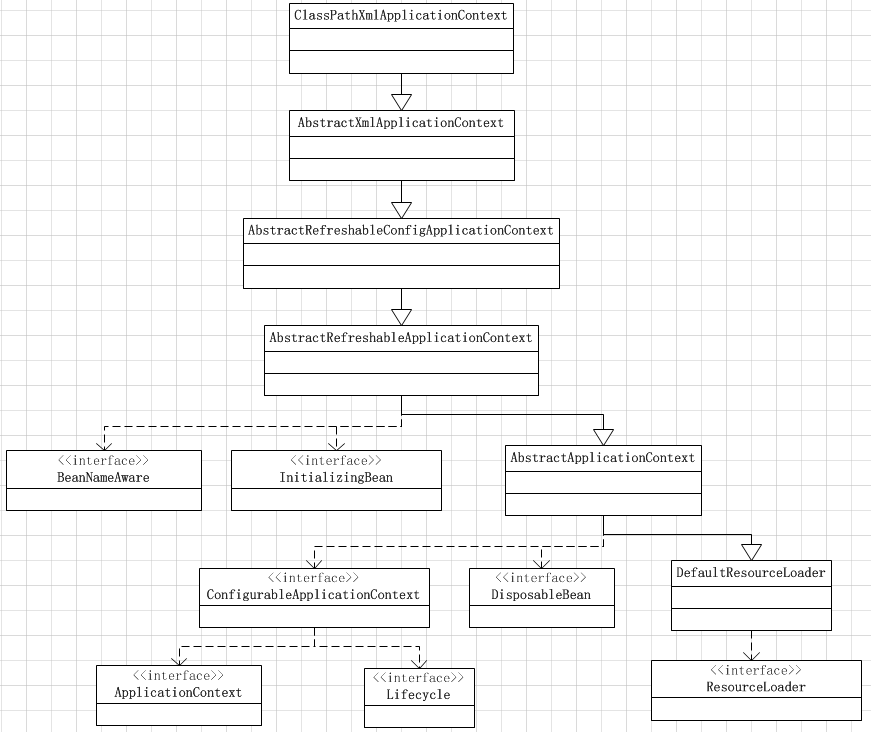
ApplicationContext applicationContext=new ClassPathXmlApplicationContext("classpath:/spring/applicationContext.xml");

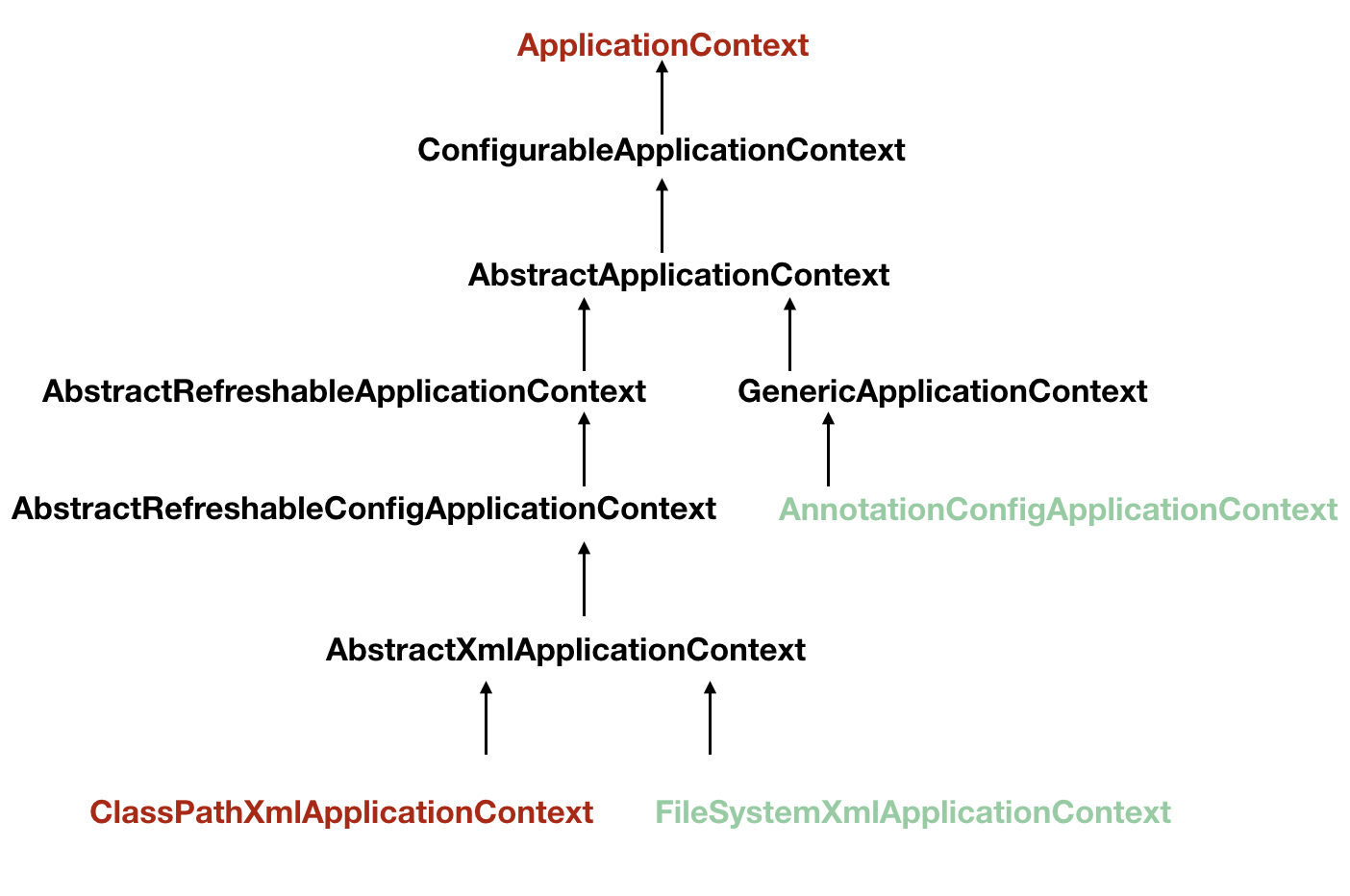
User user=(User)ApplicationContext.getBean(User.class);

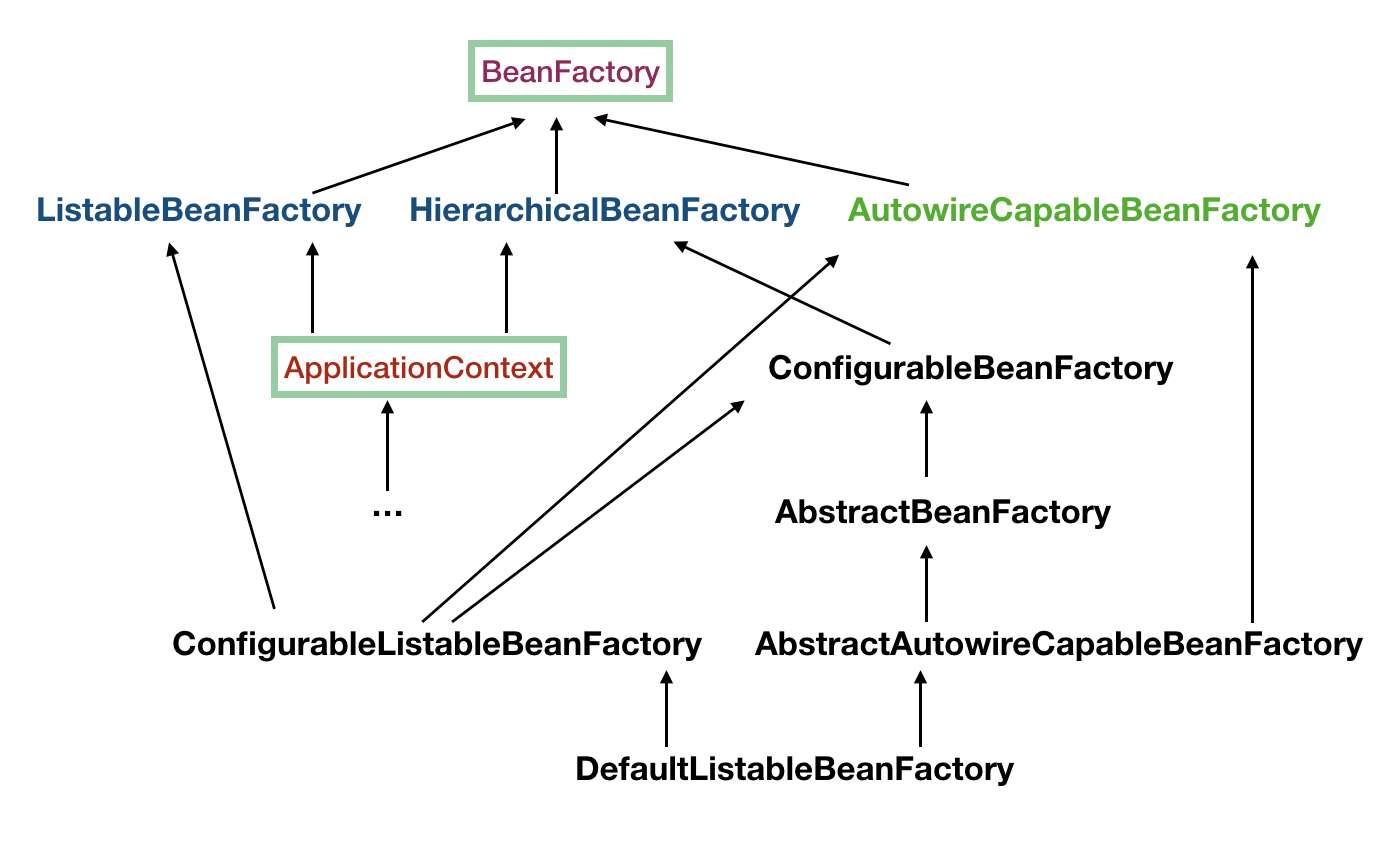
ApplicationContext 启动过程中，会负责创建实例 Bean，往各个 Bean 中注入依赖等。

ClassPathXmlApplicationContext用于加载CLASSPATH下的Spring配置文件，可以看到，第二行就已经可以获取到Bean的实例了，那么必然第一行就已经完成了对所有Bean实例的加载，因此可以通过ClassPathXmlApplicationContext作为入口。为了后面便于代码阅读，先给出一下ClassPathXmlApplicationContext这个类的继承关系：







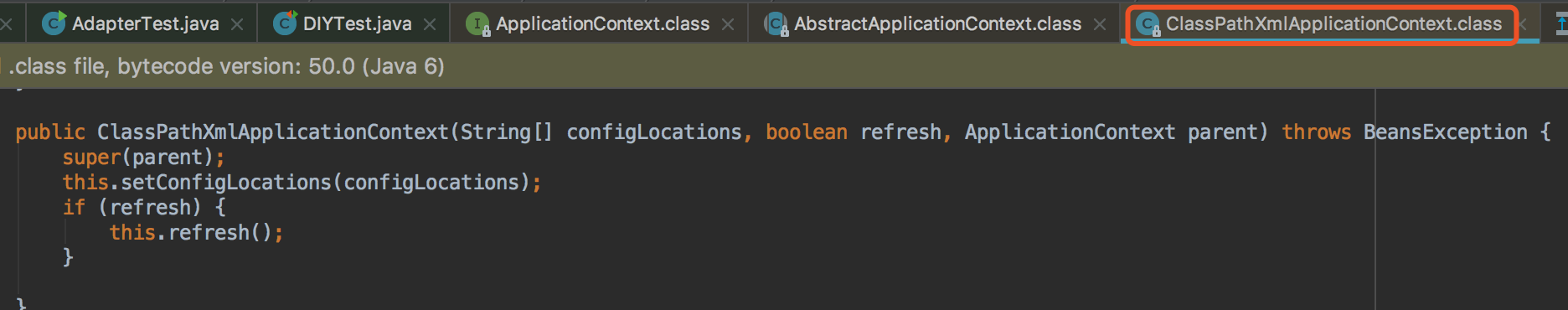


1. ApplicationContext 继承了 ListableBeanFactory，这个 Listable 的意思就是，通过这个接口，我们可以获取多个 Bean，最顶层 BeanFactory 接口的方法都是获取单个 Bean 的。
2. ApplicationContext 继承了 HierarchicalBeanFactory，Hierarchical 单词本身已经能说明问题了，也就是说我们可以在应用中起多个 BeanFactory，然后可以将各个 BeanFactory 设置为父子关系。
3. AutowireCapableBeanFactory 这个名字中的 Autowire 大家都非常熟悉，它就是用来自动装配 Bean 用的，但是仔细看上图，ApplicationContext 并没有继承它，不过不用担心，不使用继承，不代表不可以使用组合，如果你看到 ApplicationContext 接口定义中的最后一个方法 getAutowireCapableBeanFactory() 就知道了。
4. ConfigurableListableBeanFactory 也是一个特殊的接口，看图，特殊之处在于它继承了第二层所有的三个接口，而 ApplicationContext 没有。这点之后会用到。
5. 请先不用花时间在其他的接口和类上，先理解我说的这几点就可以了。

## ClassPathXmlApplicationContext

为了更理解ApplicationContext，拿一个实例ClassPathXmlApplicationContext举例，看一下里面存储的内容，加深对ApplicationContext的认识，以表格形式展现：

|  |  |  |  |
| --- | --- | --- | --- |
| **对象名** | **类  型** | **作  用** | **归属类** |
| configResources | Resource[] | 配置文件资源对象数组 | ClassPathXmlApplicationContext |
| configLocations | String[] | 配置文件字符串数组，存储配置文件路径 | AbstractRefreshableConfigApplicationContext |
| beanFactory | DefaultListableBeanFactory | 上下文使用的Bean工厂 | AbstractRefreshableApplicationContext |
| beanFactoryMonitor | Object | Bean工厂使用的同步监视器 | AbstractRefreshableApplicationContext |
| id | String | 上下文使用的唯一Id，标识此ApplicationContext | AbstractApplicationContext |
| parent | ApplicationContext | 父级ApplicationContext | AbstractApplicationContext |
| beanFactoryPostProcessors | List<BeanFactoryPostProcessor> | 存储BeanFactoryPostProcessor接口，Spring提供的一个扩展点 | AbstractApplicationContext |
| startupShutdownMonitor | Object | refresh方法和destory方法公用的一个监视器，避免两个方法同时执行 | AbstractApplicationContext |
| shutdownHook | Thread | Spring提供的一个钩子，JVM停止执行时会运行Thread里面的方法 | AbstractApplicationContext |
| resourcePatternResolver | ResourcePatternResolver | 上下文使用的资源格式解析器 | AbstractApplicationContext |
| lifecycleProcessor | LifecycleProcessor | 用于管理Bean生命周期的生命周期处理器接口 | AbstractApplicationContext |
| messageSource | MessageSource | 用于实现国际化的一个接口 | AbstractApplicationContext |
| applicationEventMulticaster | ApplicationEventMulticaster | Spring提供的事件管理机制中的事件多播器接口 | AbstractApplicationContext |
| applicationListeners | Set<ApplicationListener> | Spring提供的事件管理机制中的应用监听器 | AbstractApplicationContext |



从构造函数代码看，总共就做了三件事：

1. super(parent)

　　没什么太大的作用，设置一下父级ApplicationContext，这里是null。

1. setConfigLocations(configLocations)

　　代码就不贴了，一看就知道，里面做了两件事情：

1. 将指定的Spring配置文件的路径存储到本地

2. 解析Spring配置文件路径中的${PlaceHolder}占位符，替换为系统变量中PlaceHolder对应的Value值，System本身就自带一些系统变量比如class.path、os.name、user.dir等，也可以通过System.setProperty()方法设置自己需要的系统变量

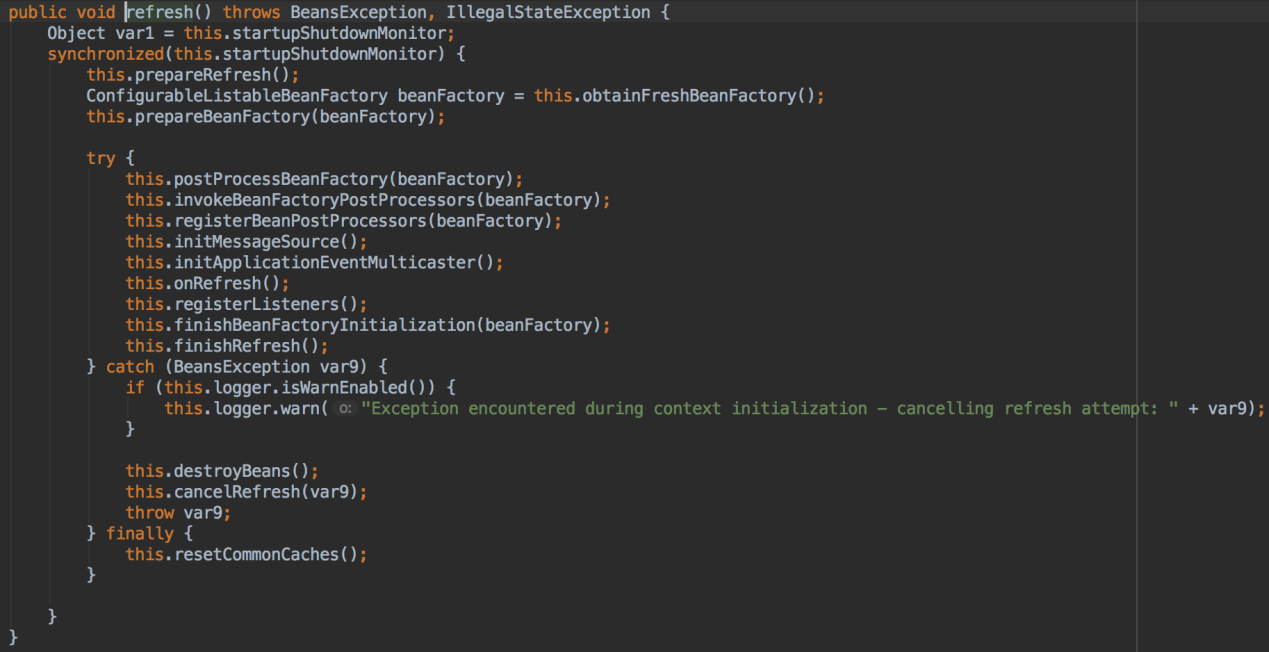
1. refresh()

　 这个就是整个Spring Bean加载的核心了，它是ClassPathXmlApplicationContext的父类AbstractApplicationContext的一个方法，顾名思义，用于刷新整个Spring上下文信息，定义了整个Spring上下文加载的流程。这里简单说下为什么是 refresh()，而不是 init() 这种名字的方法。因为 ApplicationContext 建立起来以后，其实我们是可以通过调用 refresh() 这个方法重建的，这样会将原来的 ApplicationContext 销毁，然后再重新执行一次初始化操作。

## refresh方法

上面已经说了，refresh()方法是整个Spring Bean加载的核心，因此看一下整个refresh()方法的定义：

方法的实现在类AbstractApplicationContext中。

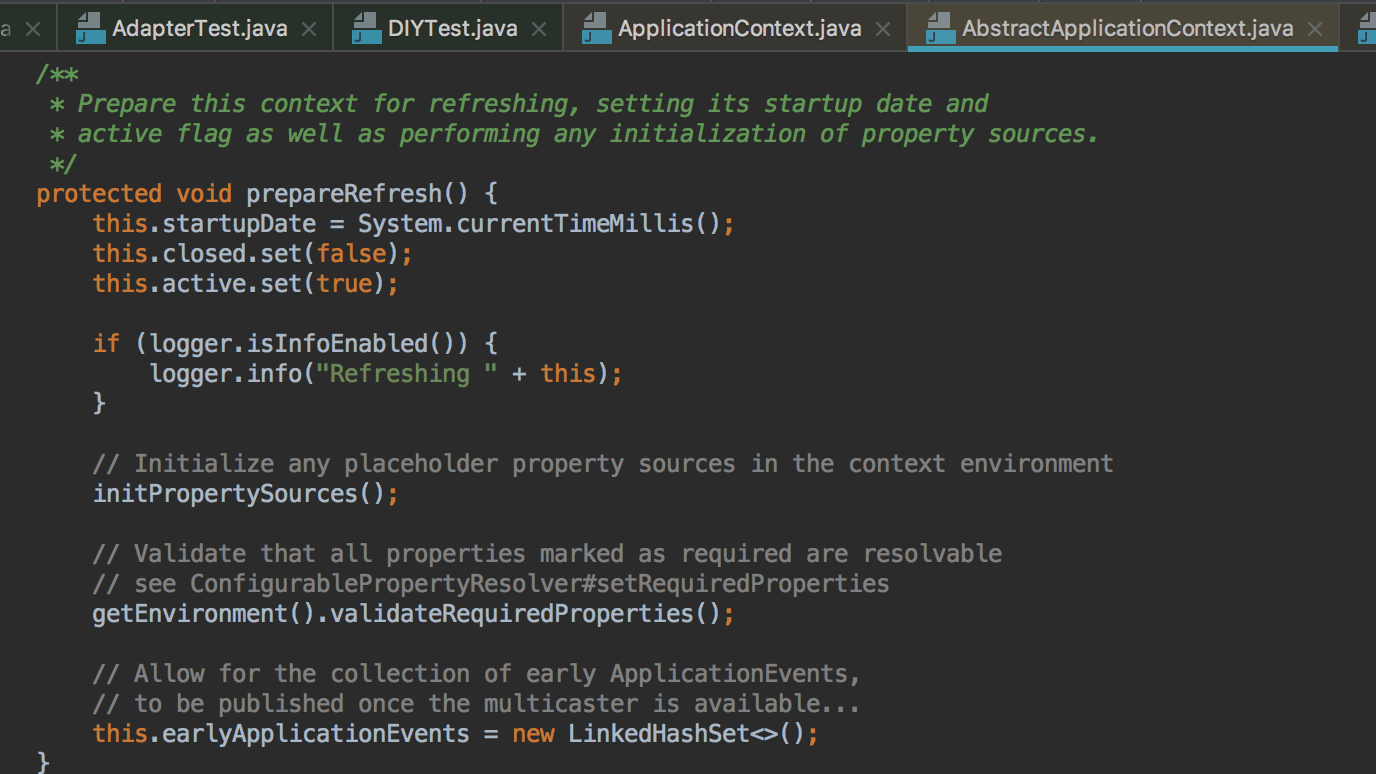


每个子方法的功能之后一点一点再分析，首先refresh()方法有几点是值得我们学习的：

1. 方法是加锁的，这么做的原因是避免多线程同时刷新Spring上下文。
2. 尽管加锁可以看到是针对整个方法体的，但是没有在方法前加synchronized关键字，而使用了对象锁startUpShutdownMonitor，这样做有两个好处：
3. refresh()方法和close()方法都使用了startUpShutdownMonitor对象锁加锁，这就保证了在调用refresh()方法的时候无法调用close()方法，反之亦然，避免了冲突
4. 另外一个好处不在这个方法中体现，但是提一下，使用对象锁可以减小了同步的范围，只对不能并发的代码块进行加锁，提高了整体代码运行的效率
5. 方法里面使用了每个子方法定义了整个refresh()方法的流程，使得整个方法流程清晰易懂。这点是非常值得学习的，一个方法里面几十行甚至上百行代码写在一起，在我看来会有三个显著的问题：
6. 扩展性降低。反过来讲，假使把流程定义为方法，子类可以继承父类，可以根据需要重写方法。
7. 代码可读性差。很简单的道理，看代码的人是愿意看一段500行的代码，还是愿意看10段50行的代码？
8. 代码可维护性差。这点和上面的类似但又有不同，可维护性差的意思是，一段几百行的代码，功能点不明确，不易后人修改，可能会导致“牵一发而动全身”。

### 创建Bean容器前的准备工作：prepareRefresh方法

下面挨个看refresh方法中的子方法，首先是prepareRefresh方法，看一下源码：



这个方法功能比较简单，顾名思义，准备刷新Spring上下文，其功能注释上写了：

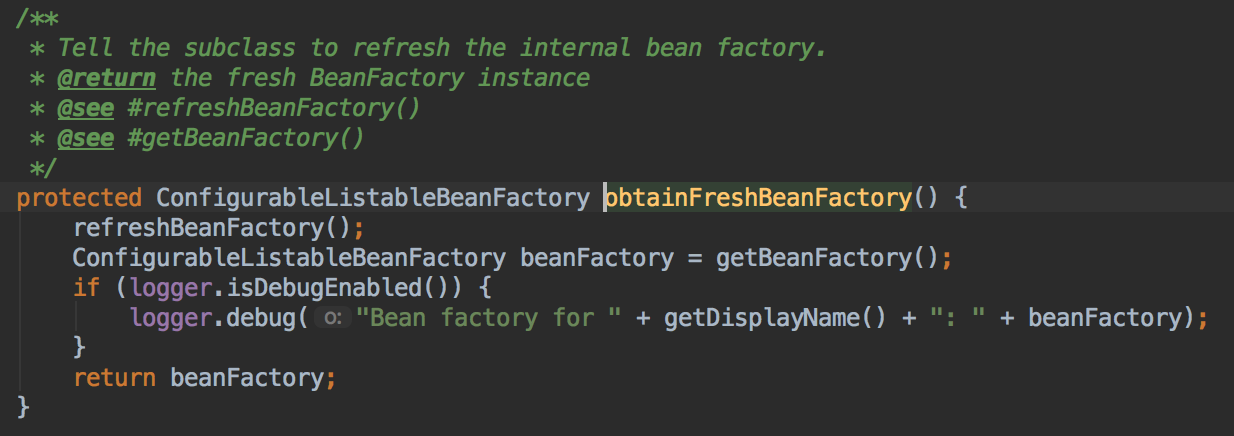
1. 设置一下刷新Spring上下文的开始时间
2. 将active标识位设置为true，closed标识位设置为false，AtomicBoolean
3. 处理配置文件中的占位符

另外可以注意一下，日志打印了真正加载Spring上下文的Java类。

### 创建Bean容器，加载并注册Bean ：obtainFreshBeanFactory方法

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)。

obtainFreshBeanFactory方法的作用是获取刷新Spring上下文的Bean工厂，其代码实现为：



其核心是refreshBeanFactory方法，这是一个抽象方法，有AbstractRefreshableApplicationContext和GenericApplicationContext这两个子类实现了这个方法，看一下上面ClassPathXmlApplicationContext的继承关系图即知，调用的应当是AbstractRefreshableApplicationContext中实现的refreshBeanFactory()，其源码为下一节内容。

#### AbstractRefreshableApplicationContext



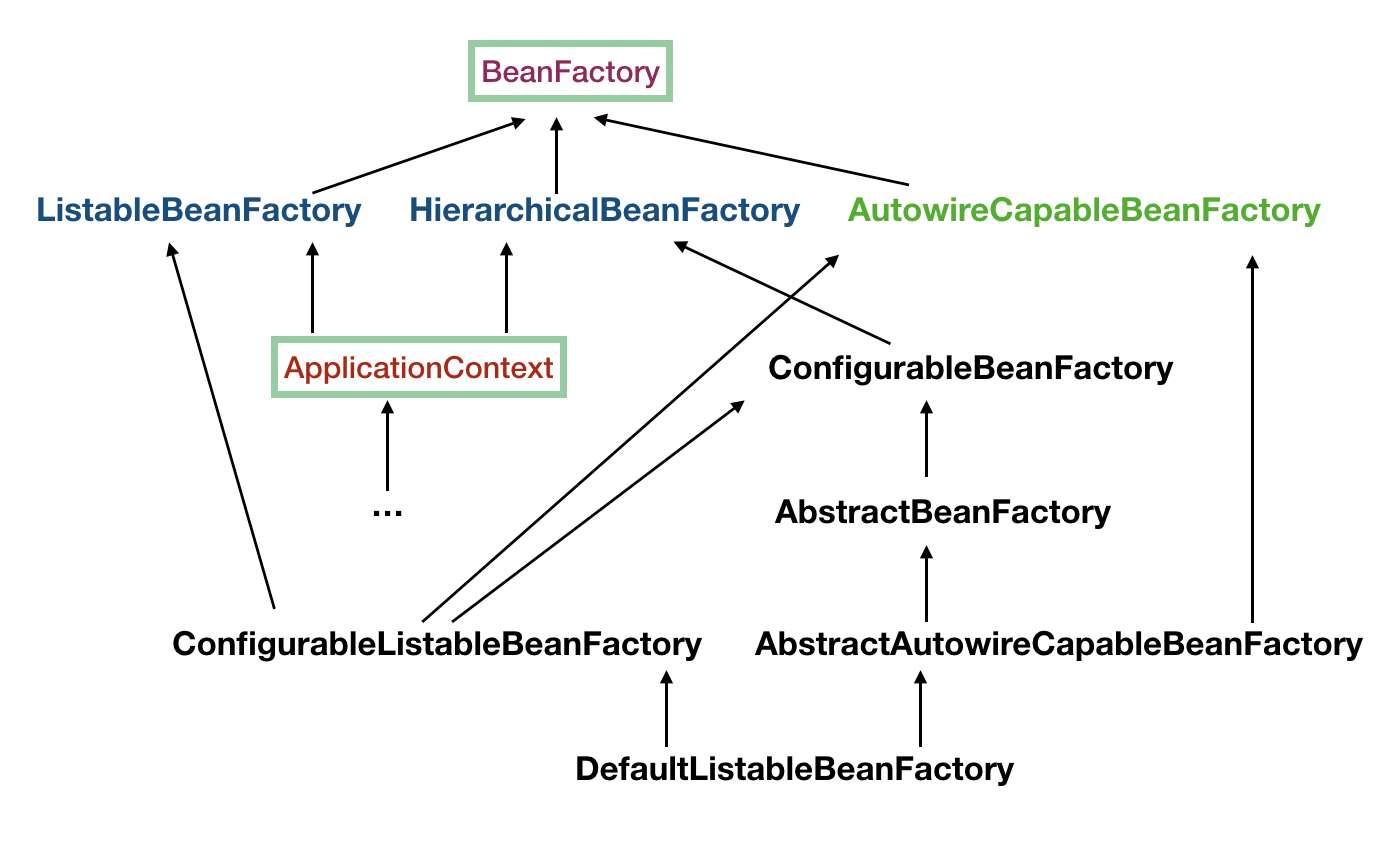
前几行代码：如果 ApplicationContext 中已经加载过 BeanFactory 了，销毁所有 Bean，关闭 BeanFactory 。注意，应用中 BeanFactory 本来就是可以多个的，这里可不是说应用全局是否有 BeanFactory，而是当前 ApplicationContext 是否有 BeanFactory。

customizeBeanFactory(beanFactory)方法设置BeanFactory的两个配置属性：是否允许 Bean 覆盖、是否允许循环引用。

loadBeanDefinitions(beanFactory)方法加载 Bean 到 BeanFactory 中。

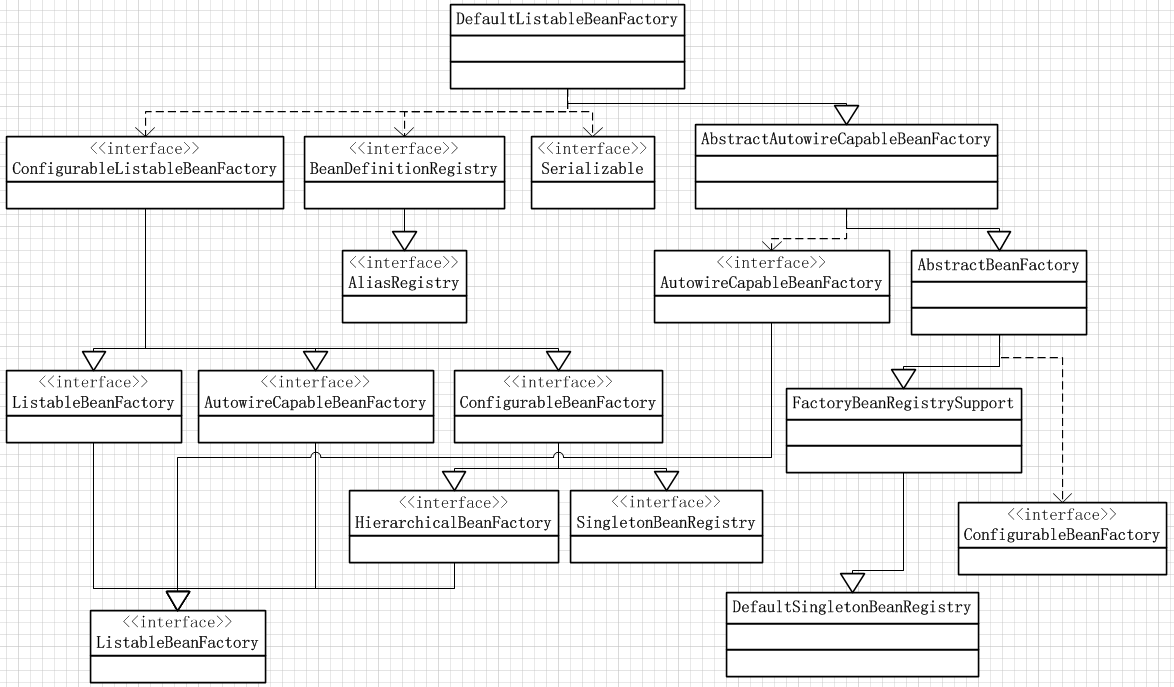
看到这里的时候，我觉得读者就应该站在高处看 ApplicationContext 了，ApplicationContext 继承自 BeanFactory，但是它不应该被理解为 BeanFactory 的实现类，而是说其内部持有一个实例化的 BeanFactory（DefaultListableBeanFactory）。以后所有的 BeanFactory 相关的操作其实是给这个实例来处理的。

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，我把之前的继承图再拿过来大家再仔细看一下：



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。

这段代码的核心点出了**DefaultListableBeanFactory**这个类，这个类是构造Bean的核心类，这个类的功能会在以后详细解读，首先给出DefaultListableBeanFactory的继承关系图：



为了更清晰地说明DefaultListableBeanFactory的作用，列举一下DefaultListableBeanFactory中存储的一些重要对象及对象中的内容，DefaultListableBeanFactory基本就是操作这些对象，以表格形式说明：

|  |  |  |  |
| --- | --- | --- | --- |
| 对象名 | 类  型 | 作    用 | 归属类 |
| aliasMap | Map<String, String> | 存储Bean名称->Bean别名映射关系 | SimpleAliasRegistry |
| singletonObjects | Map<String, Object> | 存储单例Bean名称->单例Bean实现映射关系 | DefaultSingletonBeanRegistry |
| singletonFactories | Map<String, ObjectFactory> | 存储Bean名称->ObjectFactory实现映射关系 | DefaultSingletonBeanRegistry |
| earlySingletonObjects | Map<String, Object> | 存储Bean名称->预加载Bean实现映射关系 | DefaultSingletonBeanRegistry |
| registeredSingletons | Set<String> | 存储注册过的Bean名 | DefaultSingletonBeanRegistry |
| singletonsCurrentlyInCreation | Set<String> | 存储当前正在创建的Bean名 | DefaultSingletonBeanRegistry |
| disposableBeans | Map<String, Object> | 存储Bean名称->Disposable接口实现Bean实现映射关系 | DefaultSingletonBeanRegistry |
| factoryBeanObjectCache | Map<String, Object> | 存储Bean名称->FactoryBean接口Bean实现映射关系 | FactoryBeanRegistrySupport |
| propertyEditorRegistrars | Set<PropertyEditorRegistrar> | 存储PropertyEditorRegistrar接口实现集合 | AbstractBeanFactory |
| embeddedValueResolvers | List<StringValueResolver> | 存储StringValueResolver（字符串解析器）接口实现列表 | AbstractBeanFactory |
| beanPostProcessors | List<BeanPostProcessor> | 存储 BeanPostProcessor接口实现列表 | AbstractBeanFactory |
| mergedBeanDefinitions | Map<String, RootBeanDefinition> | 存储Bean名称->合并过的根Bean定义映射关系 | AbstractBeanFactory |
| alreadyCreated | Set<String> | 存储至少被创建过一次的Bean名集合 | AbstractBeanFactory |
| ignoredDependencyInterfaces | Set<Class> | 存储不自动装配的接口Class对象集合 | AbstractAutowireCapableBeanFactory |
| resolvableDependencies | Map<Class, Object> | 存储修正过的依赖映射关系 | DefaultListableBeanFactory |
| beanDefinitionMap | Map<String, BeanDefinition> | 存储Bean名称-->Bean定义映射关系 | DefaultListableBeanFactory |
| beanDefinitionNames | List<String> | 存储Bean定义名称列表 | DefaultListableBeanFactory |

XML文件解析：继续执行方法loadBeanDefinitions(beanFactory)，这个方法对xml文件进行解析，转成JavaBean，以Key-Value的形式存入DefaultListableBeanFactory的beanDefinitionMap 对象中。该方法的实现在AbstractRefreshableApplicationContext的子类AbstractXmlApplicationContext中。

最终DefaultListableBeanFactory会先遍历beanDefinitionNames，从beanDefinitionMap中拿到对应的BeanDefinition，最终转为具体的Bean对象。

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本身是一个接口，AbstractBeanDefinition这个抽象类存储了Bean的属性，看一下AbstractBeanDefinition这个抽象类的定义：

这个类的属性与方法很多，这里就列举了一些最主要的方法和属性，可以看到包含了bean标签中的所有属性，之后就是根据AbstractBeanDefinition中的属性值构造出对应的Bean对象。

Spring没有直接拿到XML中的bean定义就直接转为具体的Bean对象，就是给Spring开发者留下了扩展点，比如之前BeanPostProcessor。

#### AbstractXmlApplicationContext

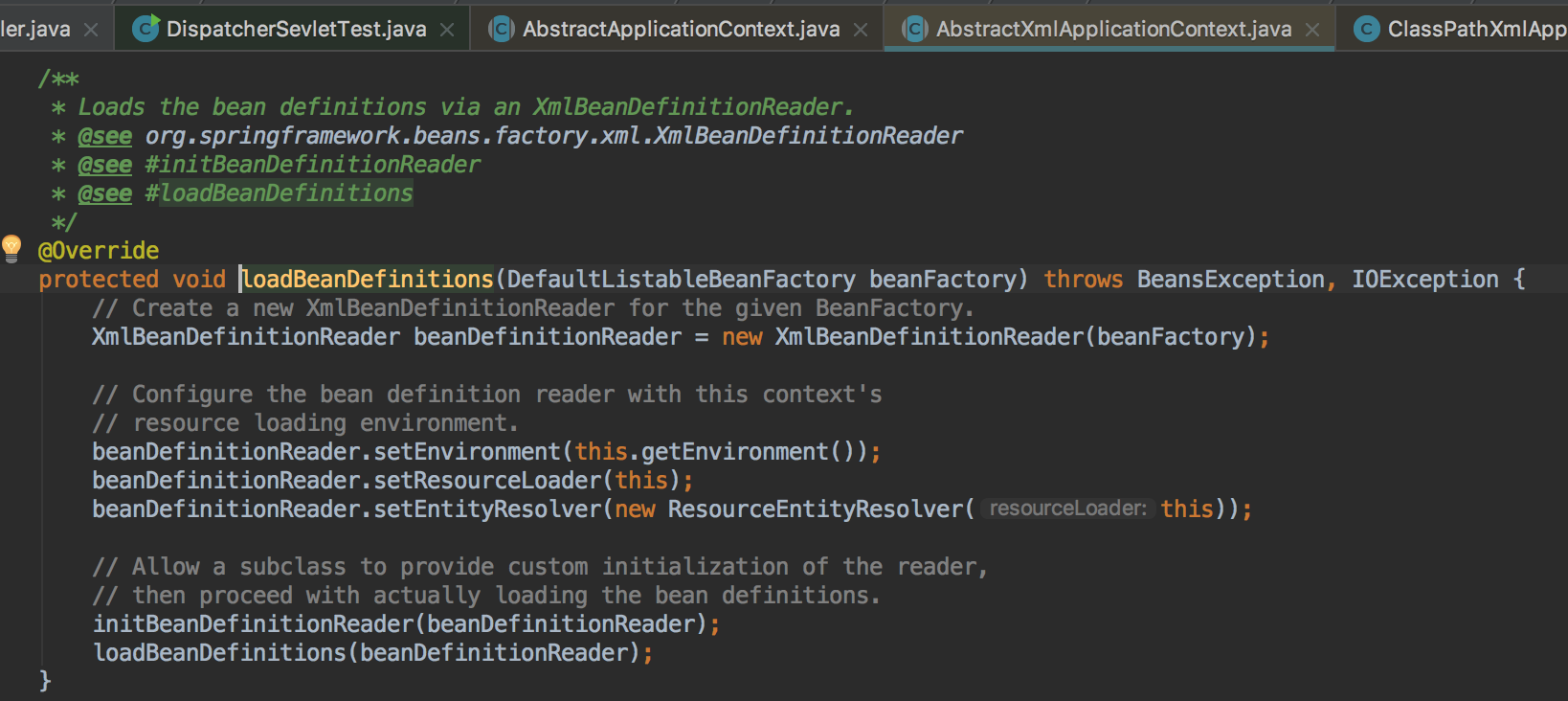
XML是如何转为Bean的，首先在AbstractXmlApplicationContext中将DefaultListableBeanFactory转换为XmlBeanDefinitionReader【一个XML文件中读取Bean定义的工具】：

XmlBeanDefinitionReader主要通过以下三步来加载Spring配置文件中的bean：

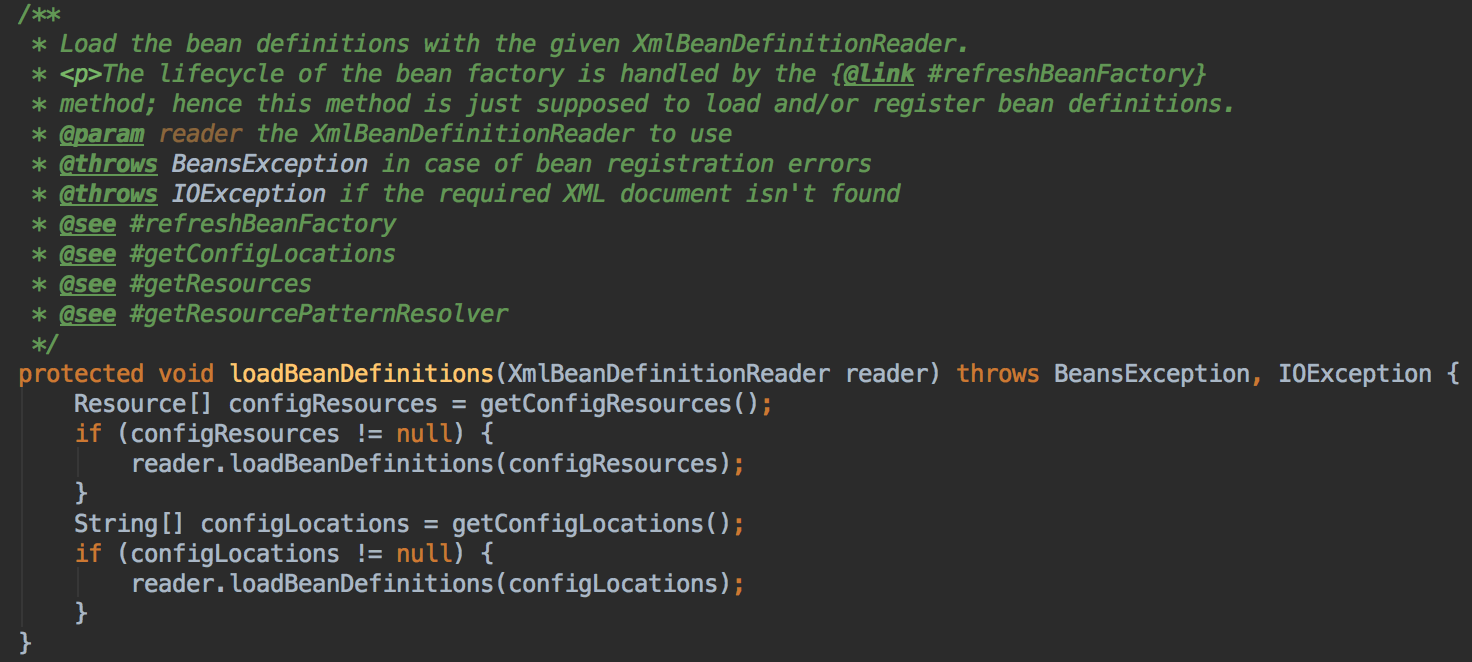
1. 通过继承自AbstractBeanDefinitionReader中的方法，使用ResourLoader将资源文件（applicationContext.xml）路径转换为对应的Resource文件；

2. 通过DocumentLoader对Resource文件进行转换，将Resource文件转换为Ducument文件；

3. 通过DefaultBeanDefinitionDocumentReader类对Document进行解析，最后再对解析后的Element进行解析。

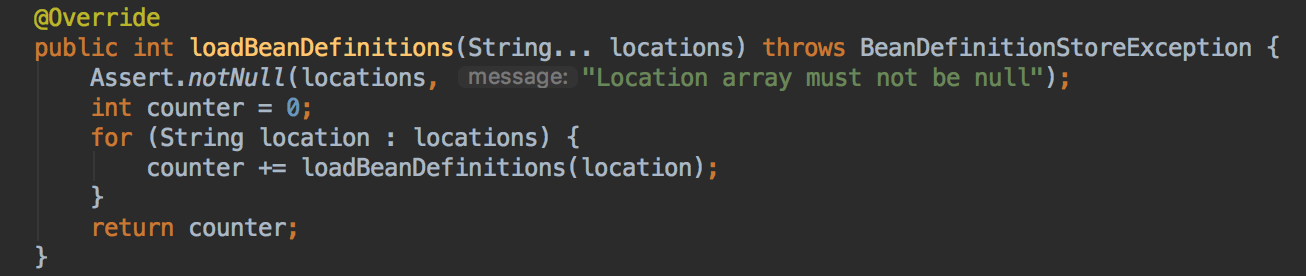


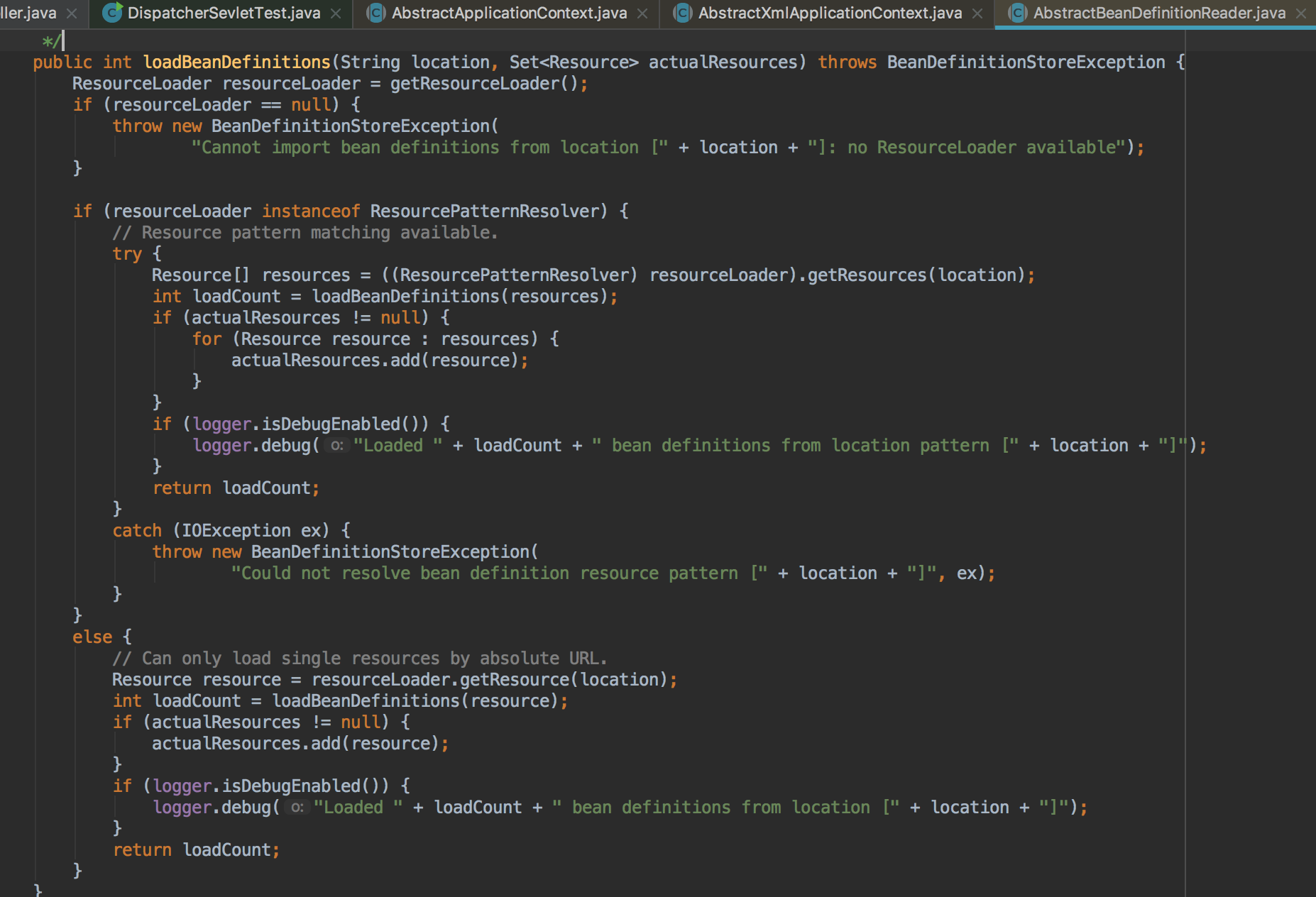
执行到方法loadBeanDefinitions()：



继续执行reader.loadBeanDefinitions()进入到类AbstractBeanDefinitionReader。

#### AbstractBeanDefinitionReader

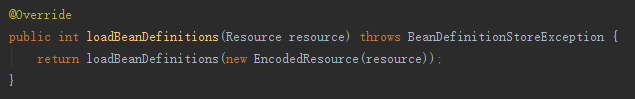


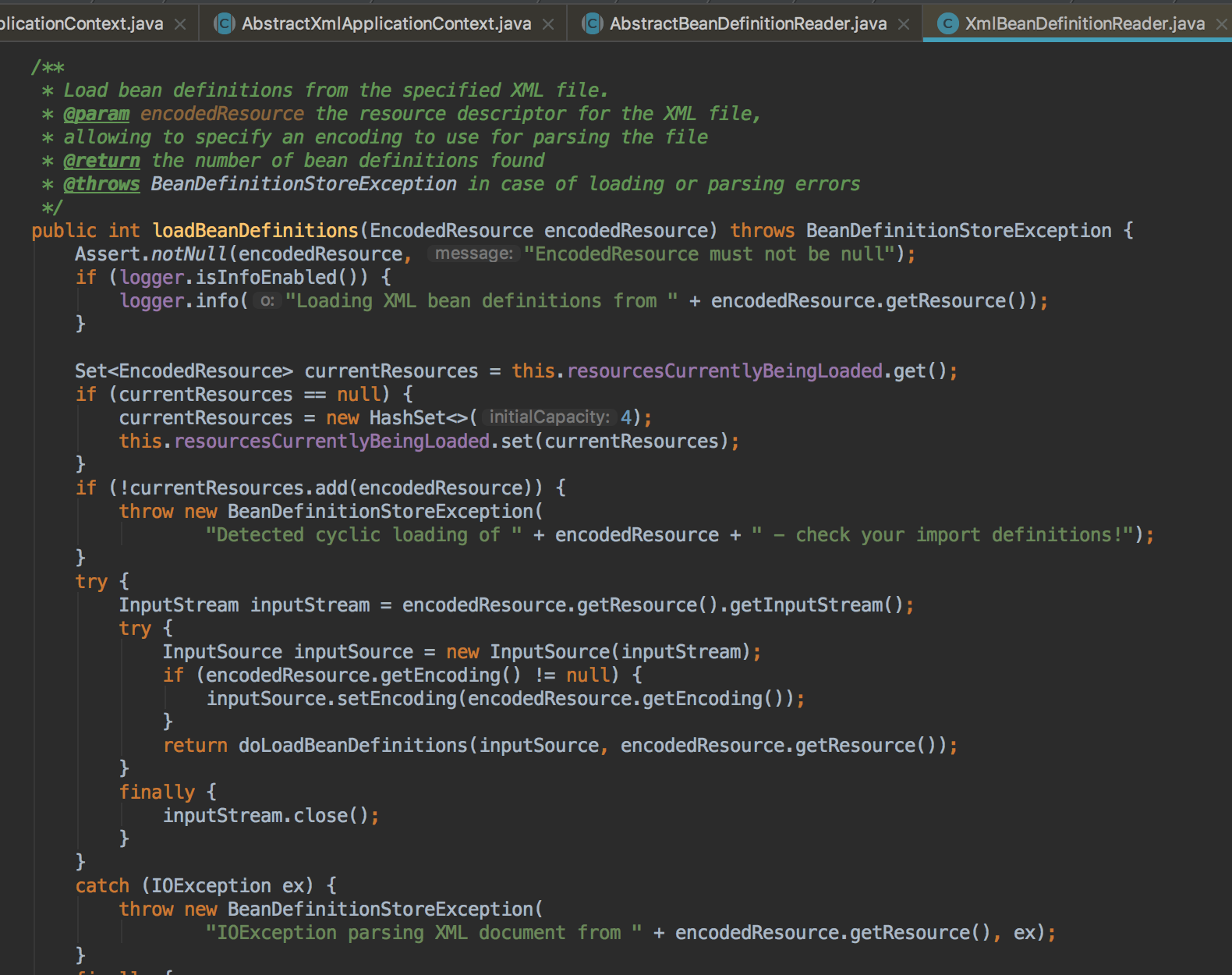


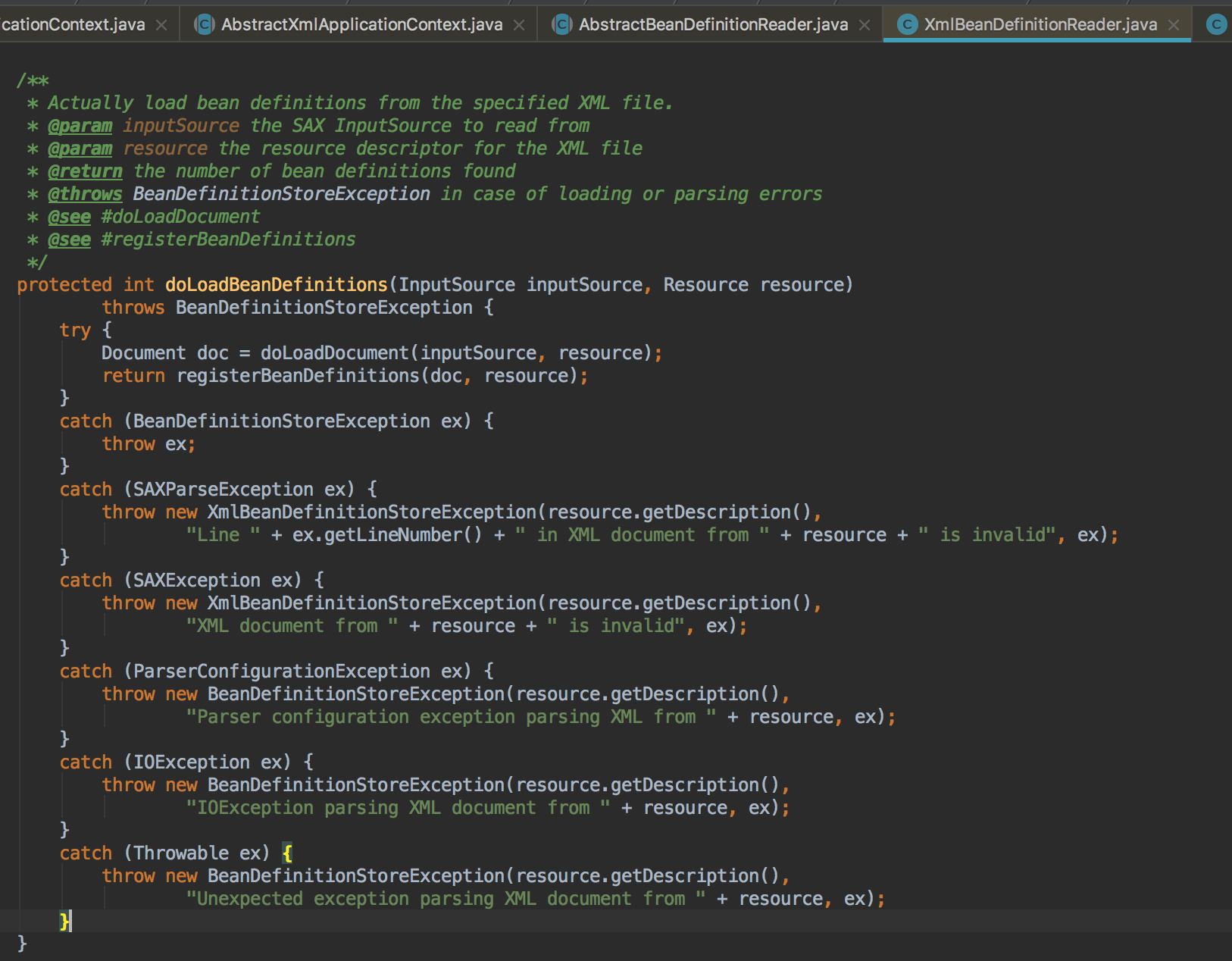
该方法返回读取到的Bean的个数。

继续执行loadBeanDefinitions()进入到类XmlBeanDefinitionReader。

#### XmlBeanDefinitionReader



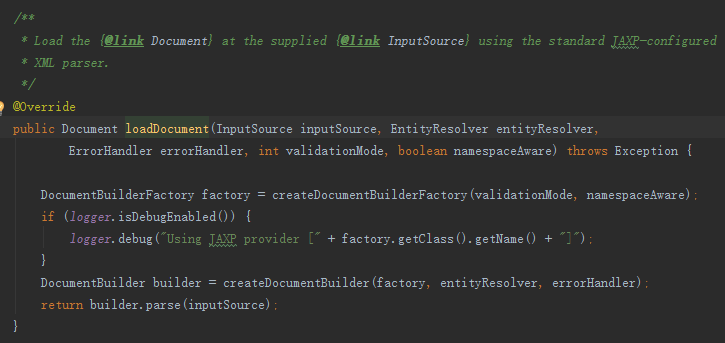




跟进方法doLoadDocument()：

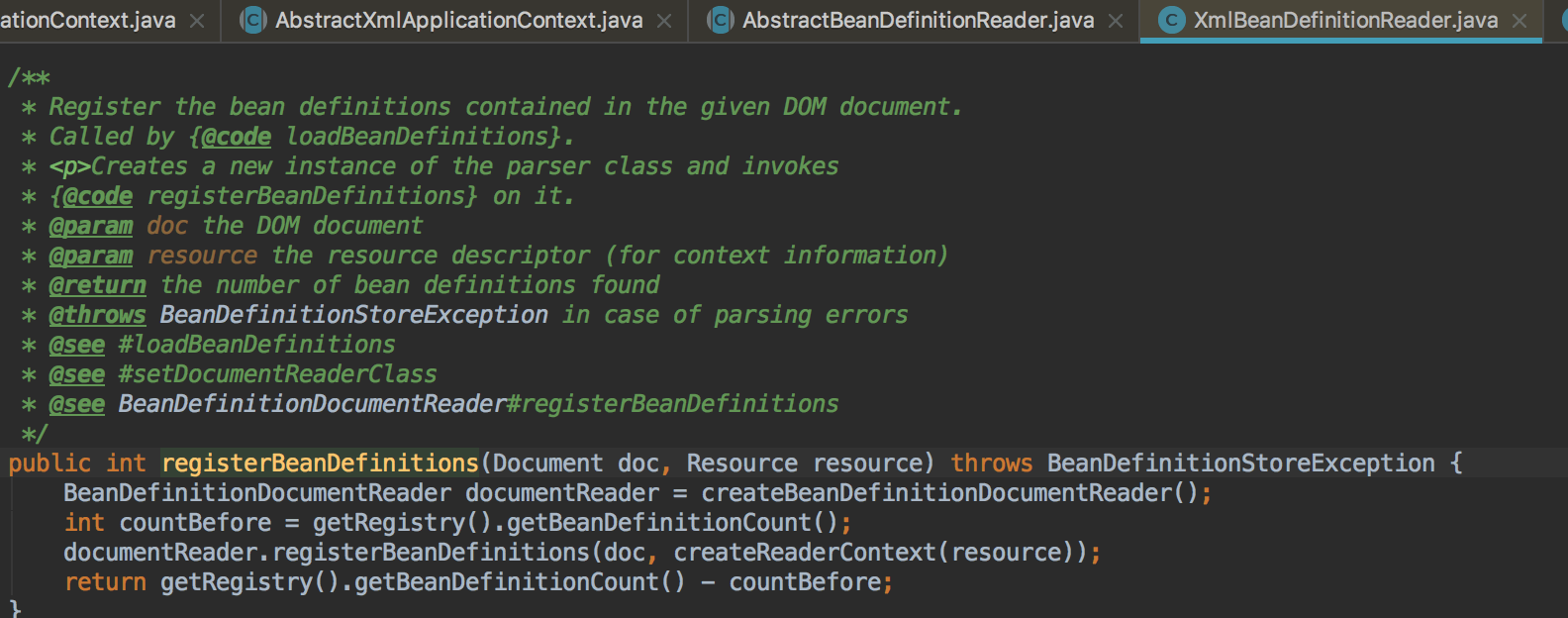


DocumentLoader是个接口，真正调用的是其实现类DefaultDocumentLoader的loadDocument方法，跟进源码：



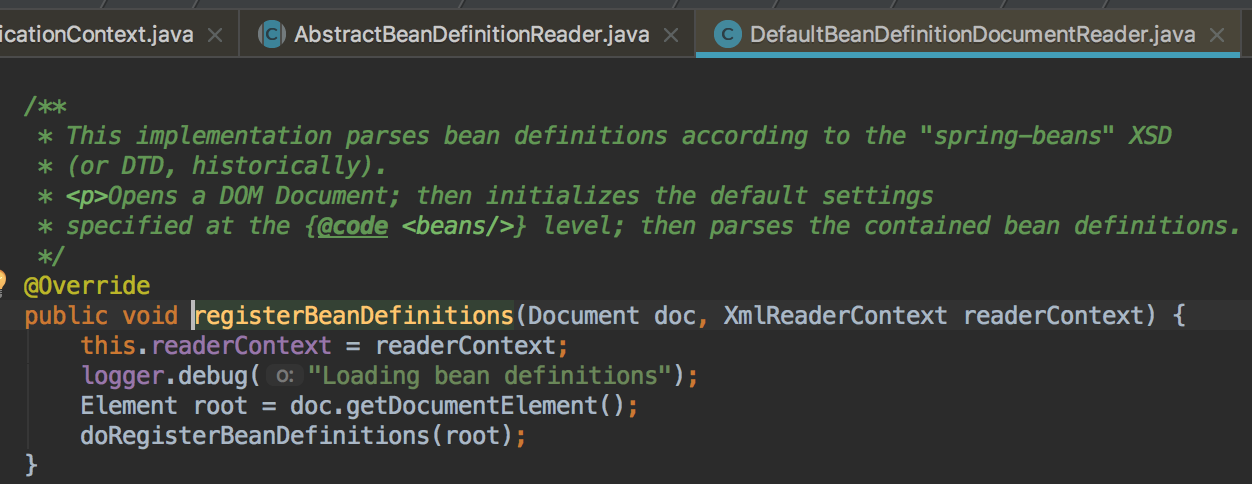
从源码可以看出这里先创建DocumentBuilderFactory，再用它创建DocumentBuilder，进而解析inputSource来返回Document对象。得到Document对象后就可以准备注册我们的Bean信息了。

执行方法registerBeanDefinitions()：

 这个方法返回本次从当前配置文件中加载了多少数量的Bean。

documentReader.registerBeanDefinitions()方法由BeanDefinitionDocumentReader的继承类DefaultBeanDefinitionDocumentReader实现。

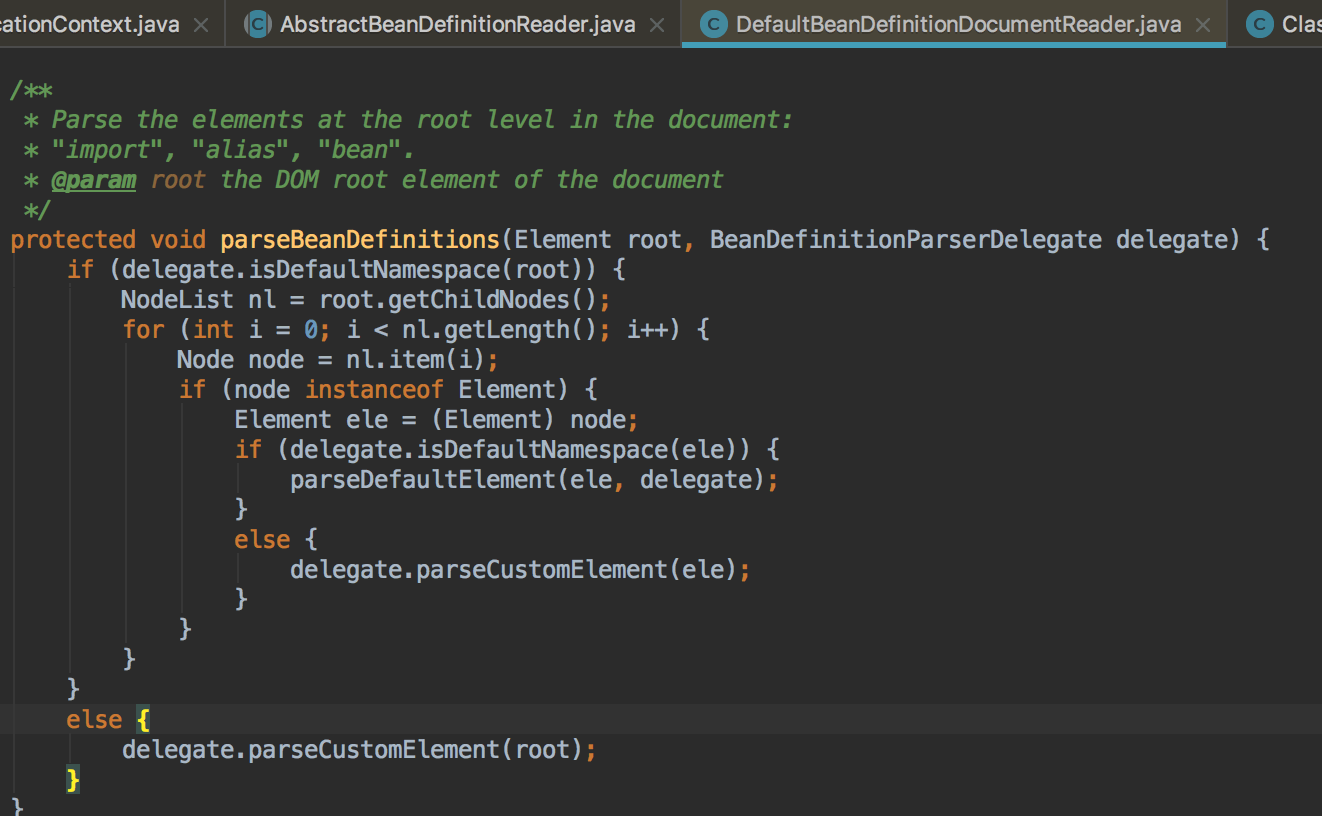
#### DefaultBeanDefinitionDocumentReader



到这里通过doc.getDocumentElement()获得Element对象后，交给doRegisterBeanDefinitions()方法后就是真正执行XML文档的解析了，跟进doRegisterBeanDefinitions()方法源码：

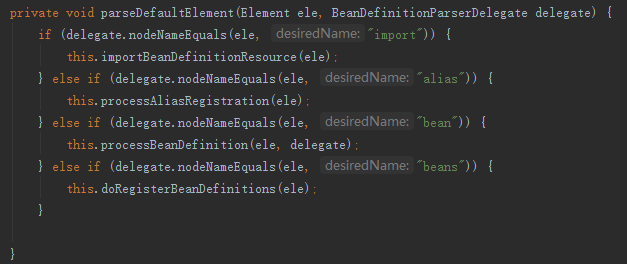


到这里处理流程就很清晰了，先是对profile进行处理，之后就通过parseBeanDefinitions()方法进行文档的解析操作，跟进parseBeanDefinitions()方法源码：

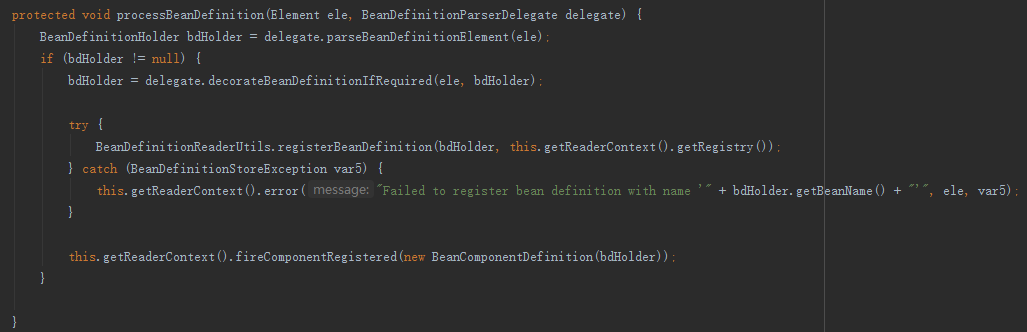


1. 如果是默认命名空间下的节点，如<import> <alias> <bean> <beans>等，则执行this.parseDefaultElement()方法。

默认命名空间是http://www.springframework.org/schema/beans。



举例：如果是<bean>节点，



核心的解析bean节点的代码为delegate.parseBeanDefinitionElement(ele)，作用是将<bean /> 节点中的信息提取出来，然后封装到一个 BeanDefinitionHolder 中。这里调用了类BeanDefinitionParserDelegate，下面为代码：

1. /\*\*
2. \* Parses the supplied {@code <bean>} element. May return {@code null}
3. \* if there were errors during parse. Errors are reported to the
4. \* {@link org.springframework.beans.factory.parsing.ProblemReporter}.
5. \*/
6. public BeanDefinitionHolder parseBeanDefinitionElement(Element ele, BeanDefinition containingBean) {
7. String id = ele.getAttribute(ID\_ATTRIBUTE);
8. String nameAttr = ele.getAttribute(NAME\_ATTRIBUTE);
9. List<String> aliases = new ArrayList<String>();
10. if (StringUtils.hasLength(nameAttr)) {
11. String[] nameArr = StringUtils.tokenizeToStringArray(nameAttr, MULTI\_VALUE\_ATTRIBUTE\_DELIMITERS);
12. aliases.addAll(Arrays.asList(nameArr));
13. }
14. String beanName = id;
15. if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
16. beanName = aliases.remove(0);
17. if (logger.isDebugEnabled()) {
18. logger.debug("No XML 'id' specified - using '" + beanName +
19. "' as bean name and " + aliases + " as aliases");
20. }
21. }
22. if (containingBean == null) {
23. checkNameUniqueness(beanName, aliases, ele);
24. }
25. AbstractBeanDefinition beanDefinition = parseBeanDefinitionElement(ele, beanName, containingBean);
26. if (beanDefinition != null) {
27. // 如果我们不定义 id 和 name，那么我们引言里的那个例子：
28. // 1. beanName 为：com.javadoop.example.MessageServiceImpl#0
29. // 2. beanClassName 为：com.javadoop.example.MessageServiceImpl
30. if (!StringUtils.hasText(beanName)) {
31. try {
32. if (containingBean != null) {
33. beanName = BeanDefinitionReaderUtils.generateBeanName(
34. beanDefinition, this.readerContext.getRegistry(), true);
35. }
36. else {
37. beanName = this.readerContext.generateBeanName(beanDefinition);
38. // Register an alias for the plain bean class name, if still possible,
39. // if the generator returned the class name plus a suffix.
40. // This is expected for Spring 1.2/2.0 backwards compatibility.
41. String beanClassName = beanDefinition.getBeanClassName();
42. if (beanClassName != null &&
43. beanName.startsWith(beanClassName) && beanName.length() > beanClassName.length() &&
44. !this.readerContext.getRegistry().isBeanNameInUse(beanClassName)) {
45. // 把 beanClassName 设置为 Bean 的别名

aliases.add(beanClassName);

1. }
2. }
3. if (logger.isDebugEnabled()) {
4. logger.debug("Neither XML 'id' nor 'name' specified - " +
5. "using generated bean name [" + beanName + "]");
6. }
7. }
8. catch (Exception ex) {
9. error(ex.getMessage(), ele);
10. return null;
11. }
12. }
13. String[] aliasesArray = StringUtils.toStringArray(aliases);
14. return new BeanDefinitionHolder(beanDefinition, beanName, aliasesArray);
15. }
16. return null;
17. }

总结一下代码逻辑：

（1）第7行和第8行，获取id属性和name属性

（2）第10行~第14行，如果填写了name属性的话，将name属性以”,;”，分割出来的字符串全部认为这个bean的别名，这里我们可以学到Spring的StringUtils的tokenizeToStringArray方法，可以将字符串按照指定分割符分割为字符串数组

（3）第16行~第23行，默认beanName为id属性，如果bean有配置别名（即上面的name属性的话），以name属性的第一个值作为beanName，发现很多人不知道beanName是什么，这几行代码就表示了容器是如何定义beanName的

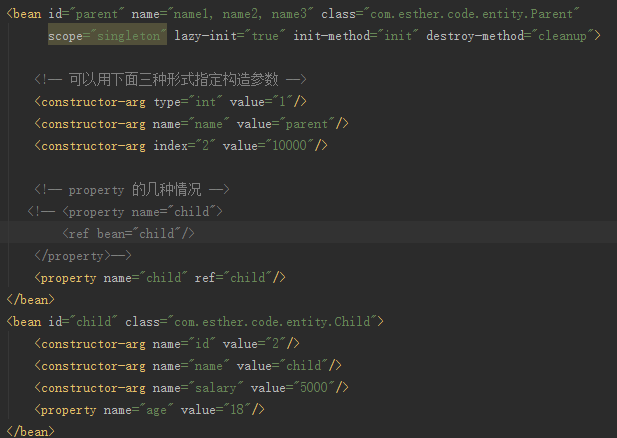
（4）第25行~第27行，这段用于保证beanName的唯一性的，BeanDefinitionParserDelegate中有一个属性usedNames，这是一个Set，强制性地保证了beanName的唯一性

（5）第29行用于解析bean的其他属性，根据 <bean ...>...</bean> 中的配置创建 BeanDefinition，然后把配置中的信息都设置到实例中,到这里，整个 <bean /> 标签就算解析结束了，一个 BeanDefinition 就形成了。后面的代码不太重要，看一下parseBeanDefinitionElement的实现：

（6）第31行~第58行，如果没有定义id和name的处理。

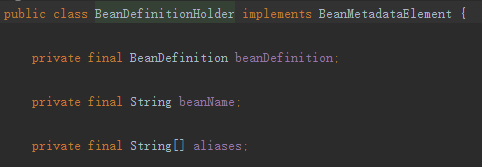
继续往下看怎么解析之前，我们先看下 <bean /> 标签中可以定义哪些属性：

| **Property** | **说明** |
| --- | --- |
| class | 类的全限定名 |
| name | 可指定 id、name(用逗号、分号、空格分隔) |
| scope | 作用域 |
| constructor arguments | 指定构造参数 |
| properties | 设置属性的值 |
| autowiring mode | no(默认值)、byName、byType、 constructor |
| lazy-initialization mode | 是否懒加载(如果被非懒加载的bean依赖了那么其实也就不能懒加载了) |
| initialization method | bean 属性设置完成后，会调用这个方法 |
| destruction method | bean 销毁后的回调方法 |

示例：

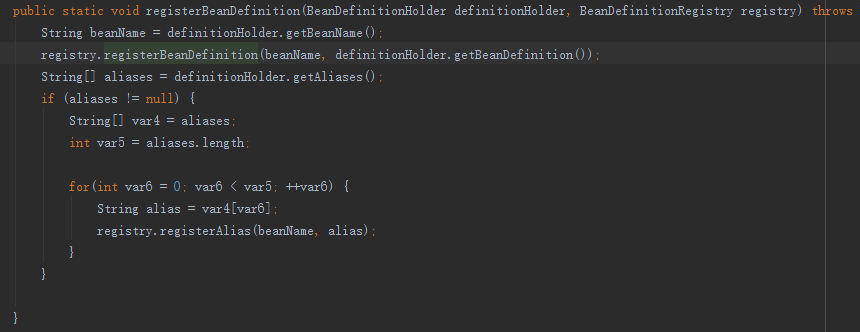
这里会取class属性、parent属性，parseBeanDefinitionAttributes()方法会取得scope、lazy-init、abstract、depends-on属性等等，设置到BeanDefinition中，这样大致上，一个Bean的定义就被存入了BeanDefinition中。

这里已经根据一个 <bean /> 标签产生了一个 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。接着看一下BeanDefinitionHolder：



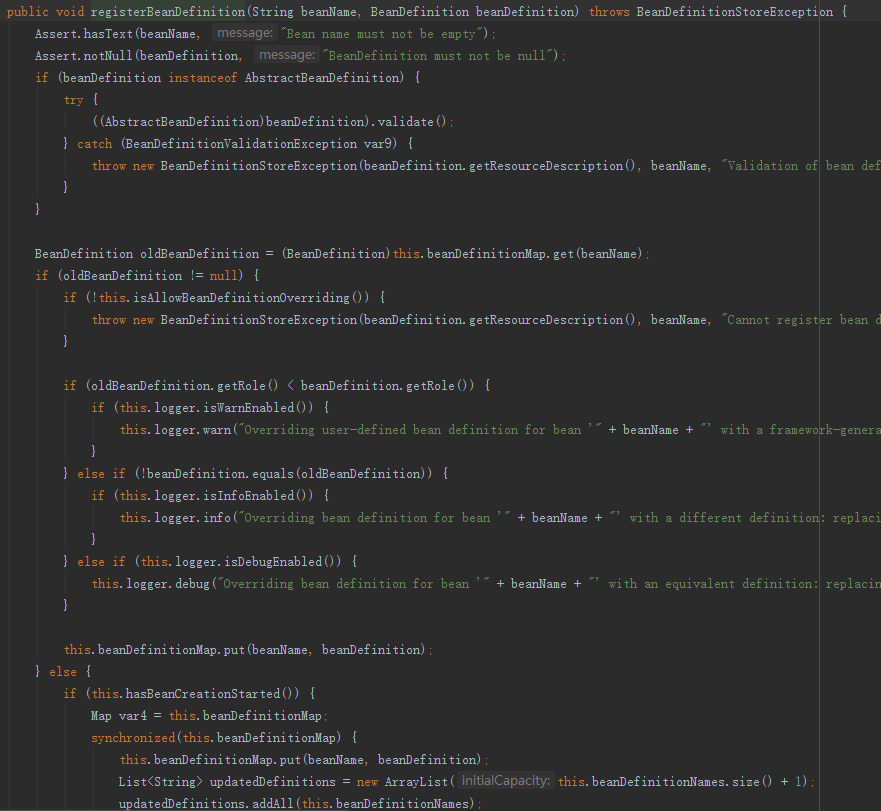
最后一步追溯到之前DefaultBeanDefinitionDocumentReader的processBeanDefinition方法，下一步是注册这个BeanDefinition：

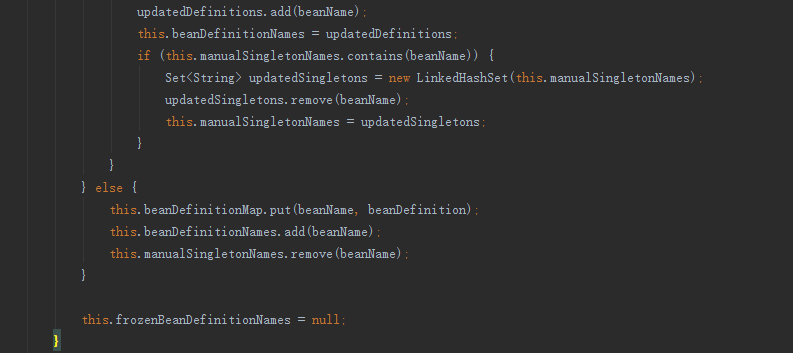
类BeanDefinitionReaderUtils：



第一步获取beanName，第二步注册这个Bean，最后根据别名注册，不然根据别名就找不到Bean了。alias->beanName是用一个map保存的，获取的时候 ，会先将alias转换为beanName，然后再查找。

根据以上分析，此时的BeanDefinitionRegistry是DefaultListableBeanFactory，registerBeanDefinition()注册Bean方法为：





首先从beanDefinitionMap中取出Bean，处理重复名称的Bean定义的情况：

* 如果不允许覆盖的话，即allowBeanDefinitionOverriding=false，抛异常；
* 用框架定义的 Bean 覆盖用户自定义的 Bean；
* 用新的 Bean 覆盖旧的 Bean；
* 用同等的 Bean 覆盖旧的 Bean*，*这里指的是equals方法返回 true的Bean。

或者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。

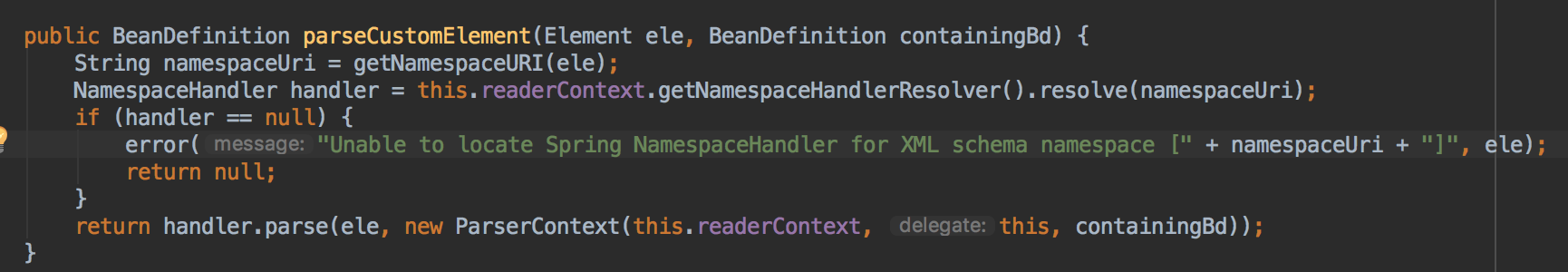
或者走到最后的else里。将 BeanDefinition 放到 beanDefinitionMap 中，这个 map 保存了所有的 BeanDefinition。按照 bean 配置的顺序保存每一个注册的 Bean 的名字到beanDefinitionNames。manualSingletonNames是个 LinkedHashSet，代表的是手动注册的 singleton bean。注意这里是 remove 方法，到这里的 Bean 当然不是手动注册的。手动指的是通过调用以下方法注册的 bean ： registerSingleton(String beanName, Object singletonObject)。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。

最终上述方法将BeanDefinition存入DefaultListableBeanFactory的beanDefinitionMap中，追踪一下代码最终到DefaultListableBeanFactory的registerBeanDefinition方法中。

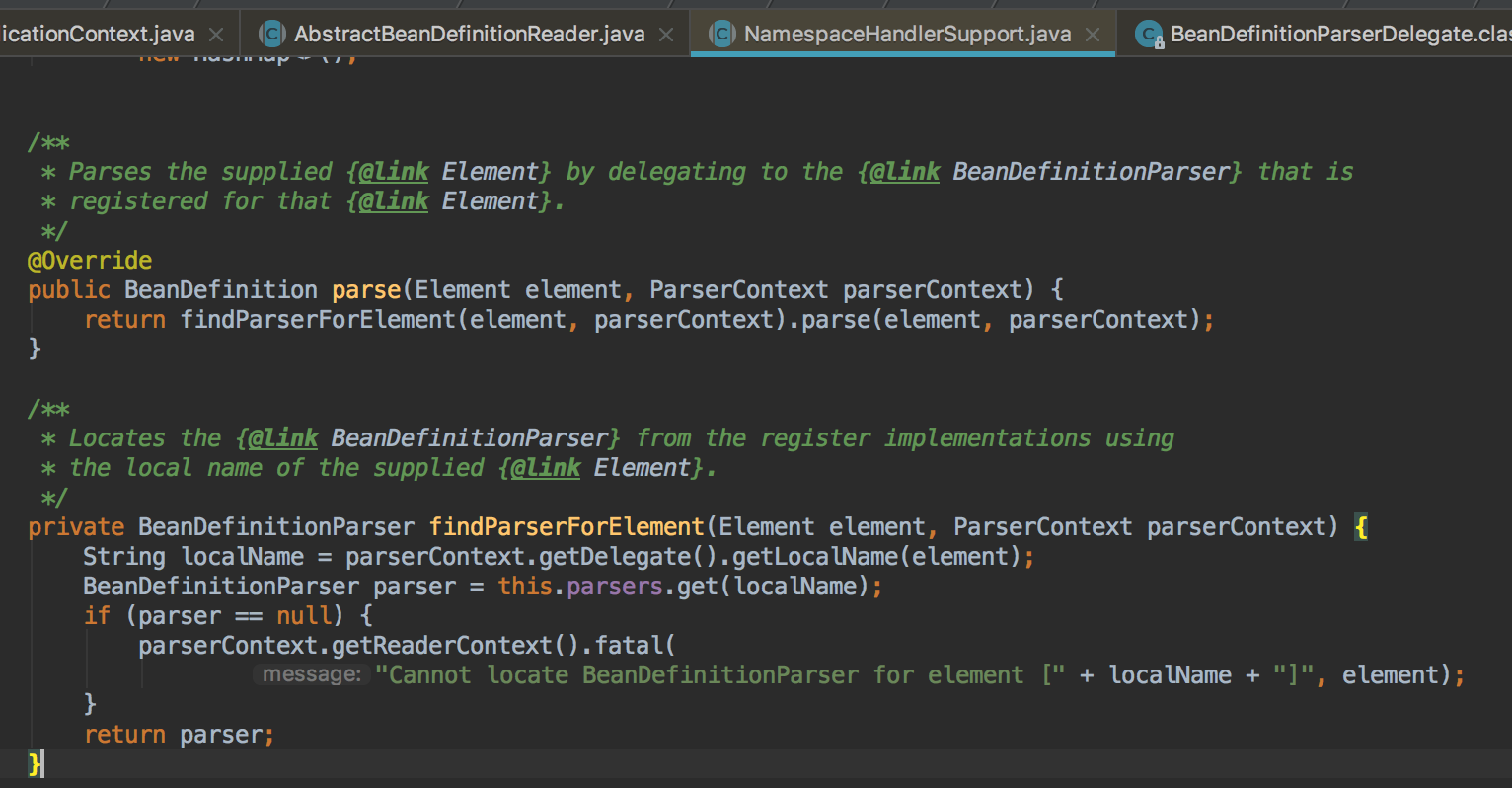
大致上就是beanDefinitionNames中增加一个beanName，beanDefinitionMap将老的BeanDefinition替换（假如不允许BeanDefinition重写的话会抛出异常）。这样一个漫长的流程过后，XML文件中的各个bean节点被转换为BeanDefinition，存入了DefaultListableBeanFactory中，后续DefaultListableBeanFactory可以根据BeanDefinition，构造对应的Bean对象出来。

总结一下，到这里已经初始化了 Bean 容器，<bean /> 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。

1. 如果是其他命名空间下的节点，如<context> <aop> <cache> <mvc> <util> <dubbo> <tx>等等，则执行delegate.parseCustomElement()方法，进入到类BeanDefinitionParserDelegate。



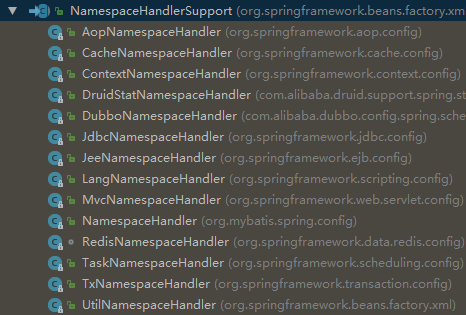
执行handler.parse()方法，进入到类NamespaceHandlerSupport执行方法parse()。



先分析NamespaceHandlerSupport的重要对象：

|  |  |  |  |
| --- | --- | --- | --- |
| **对象名** | **类  型** | **作  用** | **归属类** |
| parsers | Map<String, BeanDefinitionParser> | 存储Bean名称-->Bean定义解析器映射关系 | NamespaceHandlerSupport |
| decorators | Map<String, BeanDefinitionDecorator> | 存储Bean名称-->Bean定义装饰器映射关系 | NamespaceHandlerSupport |
| attributeDecorators | Map<String, BeanDefinitionDecorator> | 存储Bean名称-->Bean定义装饰器映射关系 | NamespaceHandlerSupport |

以下是NamespaceHandlerSupport的子类，其中parse()方法在父类中：



1、如果是<aop>节点，对应AOPNamespaceHandler：在xml文件中，包括

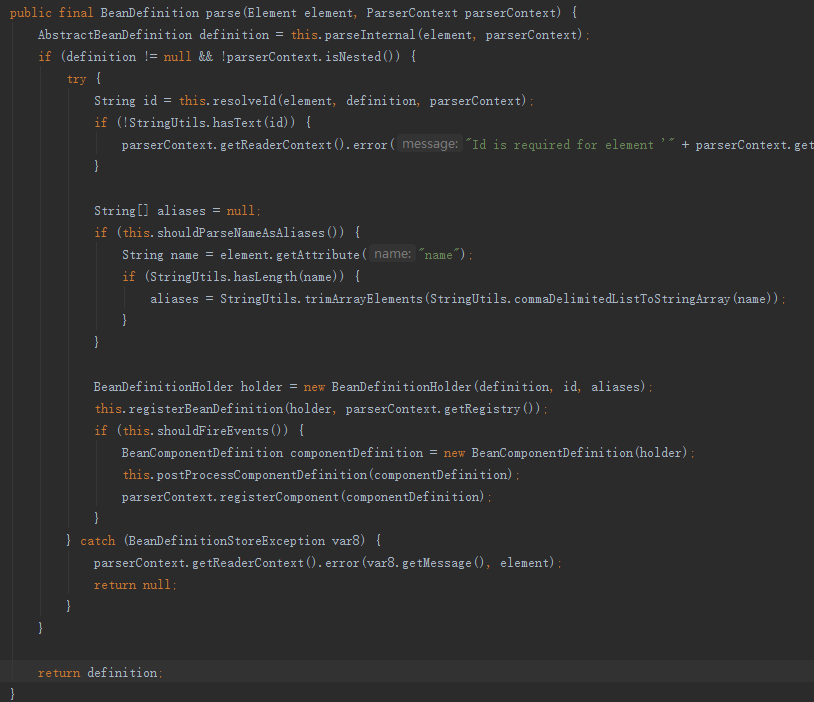
<aop:aspectj-autoproxy /> <aop:config/> <aop:scoped-proxy/>。其中<aop:config/>对应的是ConfigBeanDefinitionParser，其parse()方法为：

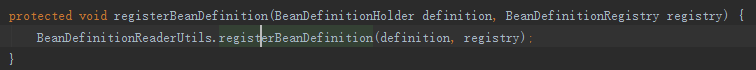


1. 如果是<cache>节点，对应CacheNamespaceHandler：在xml文件中，包括<cache:advice/> <cache:annotation-driven/>。其中<cache:annotation-driven/>对应的是AnnotationDrivenCacheBeanDefinitionParser，其parse()方法为：

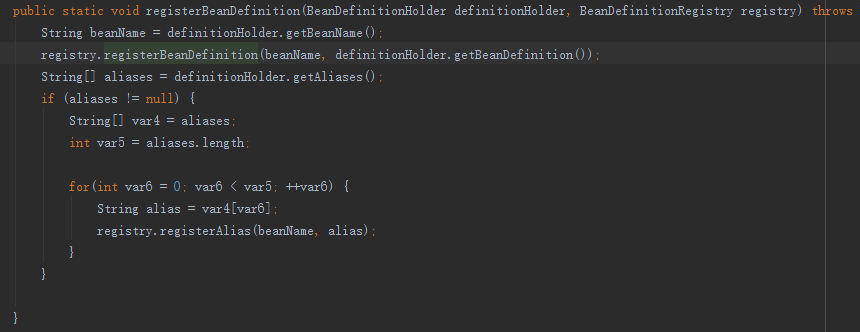


4、如果是<util>节点，对应UtilNamespaceHandler：在xml文件中，包括<util:map/> <util:set/> <util:list/> <util:constant/> <util:properties/> <util:property-path/>。其中<util:map/>对应的是MapBeanDefinitionParser，真正执行的是AbstractBeanDefinitionParser，其parse()方法为：

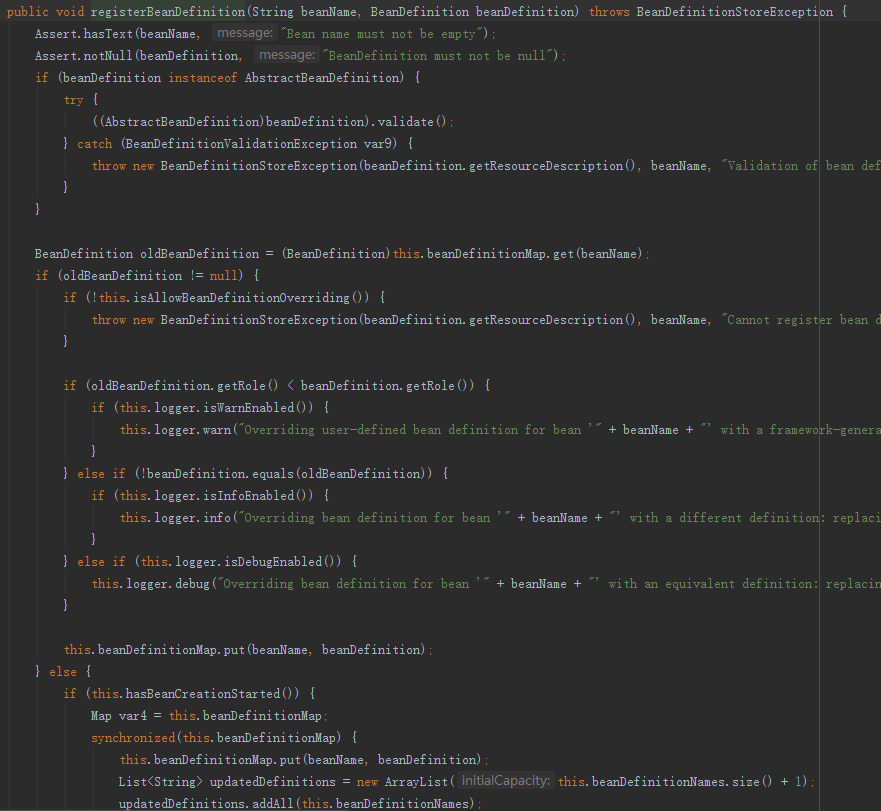


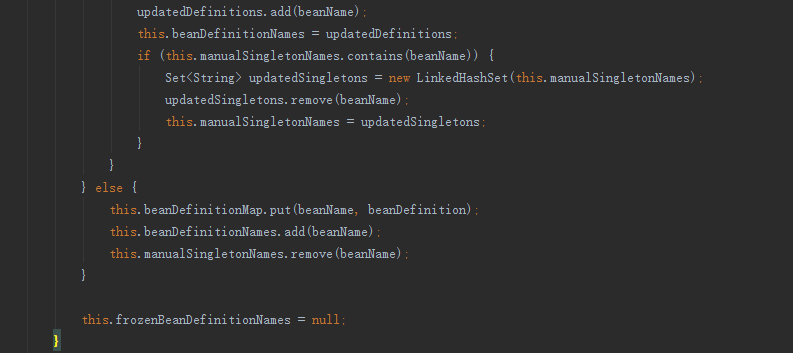


类BeanDefinitionReaderUtils：

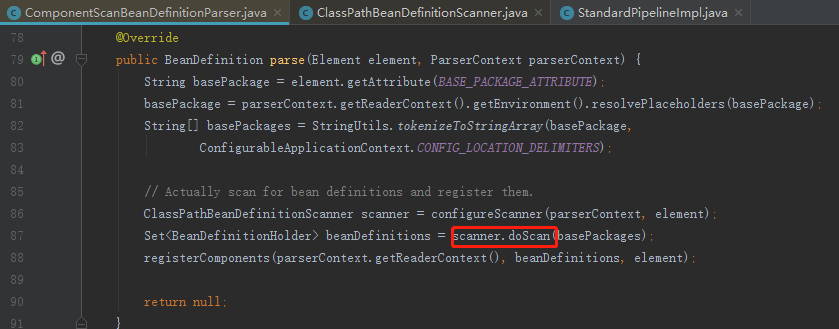


根据以上分析，此时的BeanDefinitionRegistry是DefaultListableBeanFactory，registerBeanDefinition()方法为：

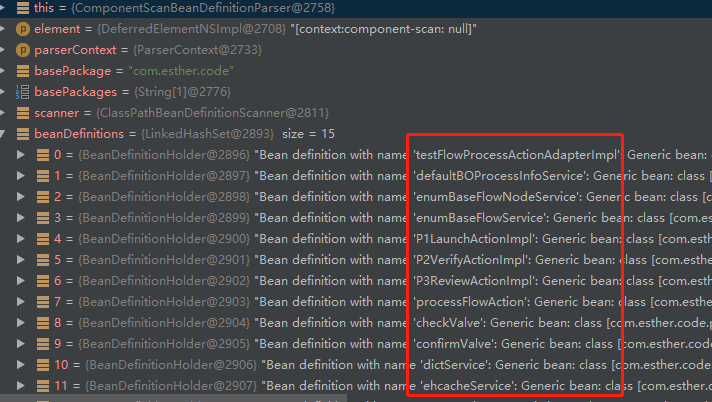




5、如果是<context>节点，对应ContextNamespaceHandler：在xml文件中，包括<context:component-scan/> <context:property-placeholder/> <context:annotation-config/>。其中<context:component-scan/>对应的是ComponentScanBeanDefinitionParser，相应的parse()方法：



scanner.doScan()方法返回在basePackage包下扫描到的组件，如下图所示：



### <bean>中不定义id及id重复场景Spring处理方式

<bean>中不定义id或者id重复，这两种场景Spring是如何处理的？

#### <bean>中不定义id

首先看一下不定义id的场景，代码在BeanDefinitionParserDelegate类的判断这里：



当bean的id未定义时，即beanName为空，进入if判断。containingBean可以看一下，这里是由方法传入的，是一个null值，因此进入else判断，即beanName由this.readerContext.generateBeanName(beanDefinition)方法生成，看一下生成方式，代码最终要追踪到BeanDefinitionReaderUtils的generateBeanName方法：

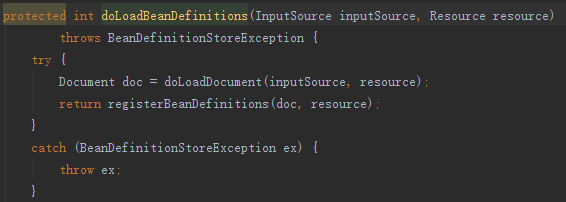


这段代码的逻辑很容易看懂，即：

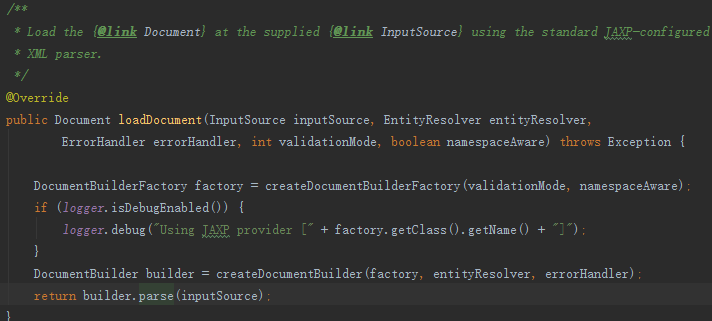
1. 假如是innerBean（比如Spring AOP产生的Bean），使用【类全路径+#+对象HashCode的16进制】的格式来命名Bean。
2. 假如不是innerBean，使用【类全路径+#+数字】的格式来命名Bean，其中数字指的是，同一个Bean出现1次，只要该Bean没有id，就从0开始依次向上累加，比如a.b.c#0、a.b.c#1、a.b.c#2。

#### <bean>中定义id重复

接着看一下id重复的场景Spring的处理方式，重复id是这样的，Spring使用XmlBeanDefinitionReader读取xml文件，在这个类的doLoadBeanDefinitions的方法中：



第4行的代码将xml解析成Document，这里的解析使用的是JDK自带的DocumentBuilder，DocumentBuilder处理xml文件输入流，发现两个<bean>中定义的id重复即会抛出XNIException异常，最终将导致Spring容器启动失败。



因此，结论就是：Spring不允许两个<bean>定义相同的id。

### 准备Bean容器：prepareBeanFactory方法

设置 BeanFactory 的类加载器，添加几个 BeanPostProcessor，手动注册几个特殊的 bean。

*/\*\*  
 \* 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？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());  
 }  
}

### postProcessBeanFactory方法

【这里需要知道 BeanFactoryPostProcessor 这个知识点，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

### invokeBeanFactoryPostProcessors方法

调用 BeanFactoryPostProcessor 各个实现类的 postProcessBeanFactory(factory) 方法。

### registerBeanPostProcessors方法

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。

### initMessageSource方法

初始化当前 ApplicationContext 的 MessageSource，国际化这里就不展开说了，不然没完没了了。

### initApplicationEventMulticaster方法

初始化当前 ApplicationContext 的事件广播器。

### onRefresh方法

从方法名就可以知道，典型的模板方法(钩子方法)， 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）。

### registerListeners方法

注册事件监听器，监听器需要实现 ApplicationListener 接口。

### 初始化所有的singleton beans：finishBeanFactoryInitialization方法

重点，重点，重点：初始化所有的 singleton beans （lazy-init 的除外）。

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。

注意，后面的描述中，我都会使用初始化或预初始化来代表这个阶段。主要是 Spring 需要在这个阶段完成所有的 singleton beans 的实例化。

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。

类AbstractApplicationContext：

// 初始化剩余的 singleton beans  
protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {  
 // 首先，初始化名字为 conversionService 的 Bean。本着送佛送到西的精神，我在附录中简单介绍了一下 ConversionService，因为这实在太实用了  
 // 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();  
}

从上面最后一行往里看，我们又回到 DefaultListableBeanFactory 这个类了。

@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，这个方法在类AbstractBeanFactory。

@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

// Eagerly check singleton cache for manually registered singletons.  
 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

// Check if bean definition exists in this factory.  
 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;  
 }

大家应该也猜到了，接下来当然是分析 createBean 方法：

protected abstract Object createBean(String beanName, RootBeanDefinition mbd, Object[] args) throws BeanCreationException;

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。

这回我们要到一个新的类了 AbstractAutowireCapableBeanFactory，看类名，AutowireCapable？类名是不是也说明了点问题了。主要是为了以下场景，采用 @Autowired 注解注入属性值。

*/\*\*  
 \* Central method of this class: creates a bean instance,  
 \* populates the bean instance, applies post-processors, etc.  
 \** ***@see*** *#doCreateBean  
 \*/*@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;  
}

**创建 Bean**

往里看 doCreateBean 这个方法：

*/\*\*  
 \* 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，我们已经分析完了 doCreateBean 方法，总的来说，我们已经说完了整个初始化流程。

接下来我们挑 doCreateBean 中的三个细节出来说说。一个是创建 Bean 实例的 createBeanInstance 方法，一个是依赖注入的 populateBean 方法，还有就是回调方法 initializeBean。

注意了，接下来的这三个方法要认真说那也是极其复杂的，很多地方我就点到为止了，感兴趣的读者可以自己往里看，最好就是碰到不懂的，自己写代码去调试它。

**创建 Bean 实例**

我们先看看 createBeanInstance 方法。需要说明的是，这个方法如果每个分支都分析下去，必然也是极其复杂冗长的，我们挑重点说。此方法的目的就是实例化我们指定的类。

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);  
}

挑个简单的**无参构造函数**构造实例来看看：

protected BeanWrapper instantiateBean(final String beanName, final RootBeanDefinition mbd) {  
 try {  
 Object beanInstance;  
 final BeanFactory parent = this;  
 if (System.*getSecurityManager*() != null) {  
 beanInstance = AccessController.doPrivileged(new PrivilegedAction<Object>() {  
 @Override  
 public Object run() {  
  
 return getInstantiationStrategy().instantiate(mbd, beanName, parent);  
 }  
 }, getAccessControlContext());  
 }  
 else {  
 // 实例化  
 beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);  
 }  
 // 包装一下，返回  
 BeanWrapper bw = new BeanWrapperImpl(beanInstance);  
 initBeanWrapper(bw);  
 return bw;  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Instantiation of bean failed", ex);  
 }  
}

我们可以看到，关键的地方在于：

beanInstance = getInstantiationStrategy().instantiate(mbd, beanName, parent);

这里会进行实际的实例化过程，我们进去看看:

**SimpleInstantiationStrategy**

@Override  
public Object instantiate(RootBeanDefinition bd, String beanName, BeanFactory owner) {  
 // 如果不存在方法覆写，那就使用 java 反射进行实例化，否则使用 CGLIB,  
 // 方法覆写 请参见附录"方法注入"中对 lookup-method 和 replaced-method 的介绍  
 if (bd.getMethodOverrides().isEmpty()) {  
 Constructor<?> constructorToUse;  
 synchronized (bd.constructorArgumentLock) {  
 constructorToUse = (Constructor<?>) bd.resolvedConstructorOrFactoryMethod;  
 if (constructorToUse == null) {  
 final Class<?> clazz = bd.getBeanClass();  
 if (clazz.isInterface()) {  
 throw new BeanInstantiationException(clazz, "Specified class is an interface");  
 }  
 try {  
 if (System.*getSecurityManager*() != null) {  
 constructorToUse = AccessController.*doPrivileged*(new PrivilegedExceptionAction<Constructor<?>>() {  
 @Override  
 public Constructor<?> run() throws Exception {  
 return clazz.getDeclaredConstructor((Class[]) null);  
 }  
 });  
 }  
 else {  
 constructorToUse = clazz.getDeclaredConstructor((Class[]) null);  
 }  
 bd.resolvedConstructorOrFactoryMethod = constructorToUse;  
 }  
 catch (Throwable ex) {  
 throw new BeanInstantiationException(clazz, "No default constructor found", ex);  
 }  
 }  
 }  
 // 利用构造方法进行实例化  
 return BeanUtils.*instantiateClass*(constructorToUse);  
 }  
 else {  
 // 存在方法覆写，利用 CGLIB 来完成实例化，需要依赖于 CGLIB 生成子类，这里就不展开了  
 return instantiateWithMethodInjection(bd, beanName, owner);  
 }  
}

到这里，我们就算实例化完成了。我们开始说怎么进行属性注入。

**bean 属性注入**

**AbstractAutowireCapableBeanFactory**

protected void populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw) {  
 // bean 实例的所有属性都在这里了  
 PropertyValues pvs = mbd.getPropertyValues();  
 if (bw == null) {  
 if (!pvs.isEmpty()) {  
 throw new BeanCreationException(  
 mbd.getResourceDescription(), beanName, "Cannot apply property values to null instance");  
 }  
 else {  
 // Skip property population phase for null instance.  
 return;  
 }  
 }  
 // 到这步的时候，bean 实例化完成（通过工厂方法或构造方法），但是还没开始属性设值，  
 // InstantiationAwareBeanPostProcessor 的实现类可以在这里对 bean 进行状态修改，  
 // 我也没找到有实际的使用，所以我们暂且忽略这块吧  
 boolean continueWithPropertyPopulation = true;  
 if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {  
 for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof InstantiationAwareBeanPostProcessor) {  
 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  
 // 如果返回 false，代表不需要进行后续的属性设值，也不需要再经过其他的 BeanPostProcessor 的处理  
 if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {  
 continueWithPropertyPopulation = false;  
 break;  
 }  
 }  
 }  
 }  
 if (!continueWithPropertyPopulation) {  
 return;  
 }  
  
 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_NAME* ||  
 mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_TYPE*) {  
 MutablePropertyValues newPvs = new MutablePropertyValues(pvs);  
  
 // 通过名字找到所有属性值，如果是 bean 依赖，先初始化依赖的 bean。记录依赖关系  
 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_NAME*) {  
 autowireByName(beanName, mbd, bw, newPvs);  
 }  
  
 // 通过类型装配。复杂一些  
 if (mbd.getResolvedAutowireMode() == RootBeanDefinition.*AUTOWIRE\_BY\_TYPE*) {  
 autowireByType(beanName, mbd, bw, newPvs);  
 }  
  
 pvs = newPvs;  
 }  
  
 boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();  
 boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.*DEPENDENCY\_CHECK\_NONE*);  
  
 if (hasInstAwareBpps || needsDepCheck) {  
 PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);  
 if (hasInstAwareBpps) {  
 for (BeanPostProcessor bp : getBeanPostProcessors()) {  
 if (bp instanceof InstantiationAwareBeanPostProcessor) {  
 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;  
 // 这里有个非常有用的 BeanPostProcessor 进到这里: AutowiredAnnotationBeanPostProcessor  
 // 对采用 @Autowired、@Value 注解的依赖进行设值，这里的内容也是非常丰富的，不过本文不会展开说了，感兴趣的读者请自行研究  
 pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);  
 if (pvs == null) {  
 return;  
 }  
 }  
 }  
 }  
 if (needsDepCheck) {  
 checkDependencies(beanName, mbd, filteredPds, pvs);  
 }  
 }  
 // 设置 bean 实例的属性值  
 applyPropertyValues(beanName, mbd, bw, pvs);  
}

**initializeBean**

属性注入完成后，这一步其实就是处理各种回调了。

protected Object initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd) {  
 if (System.*getSecurityManager*() != null) {  
 AccessController.doPrivileged(new PrivilegedAction<Object>() {  
 @Override  
 public Object run() {  
 invokeAwareMethods(beanName, bean);  
 return null;  
 }  
 }, getAccessControlContext());  
 }  
 else {  
 // 如果 bean 实现了 BeanNameAware、BeanClassLoaderAware 或 BeanFactoryAware 接口，回调  
 invokeAwareMethods(beanName, bean);  
 }  
  
 Object wrappedBean = bean;  
 if (mbd == null || !mbd.isSynthetic()) {  
 // BeanPostProcessor 的 postProcessBeforeInitialization 回调  
 wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);  
 }  
  
 try {  
 // 处理 bean 中定义的 init-method，  
 // 或者如果 bean 实现了 InitializingBean 接口，调用 afterPropertiesSet() 方法  
 invokeInitMethods(beanName, wrappedBean, mbd);  
 }  
 catch (Throwable ex) {  
 throw new BeanCreationException(  
 (mbd != null ? mbd.getResourceDescription() : null),  
 beanName, "Invocation of init method failed", ex);  
 }  
  
 if (mbd == null || !mbd.isSynthetic()) {  
 // BeanPostProcessor 的 postProcessAfterInitialization 回调  
 wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);  
 }  
 return wrappedBean;  
}

大家发现没有，BeanPostProcessor 的两个回调都发生在这边，只不过中间处理了 init-method，是不是和读者原来的认知有点不一样了？

### finishRefresh方法

最后，广播事件，ApplicationContext 初始化完成