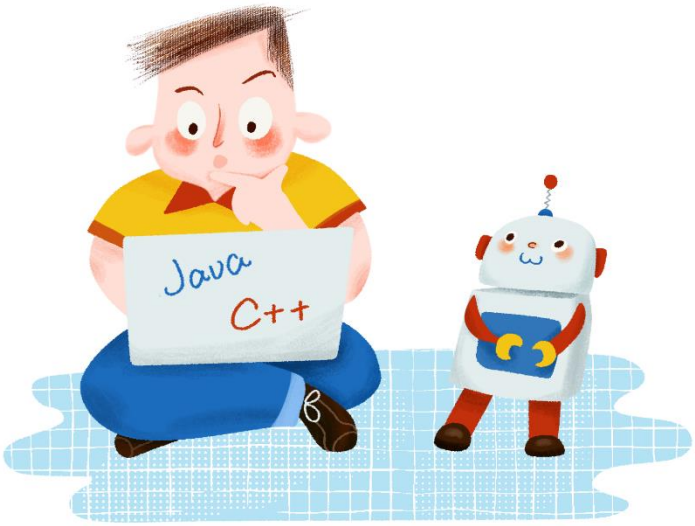
手撸Spring

作者：小傅哥

公众号：bugstack虫洞栈



让懂了就是真的懂！

IOC / AOP / 代理 / 切面 / 自动扫描 / 循环依赖

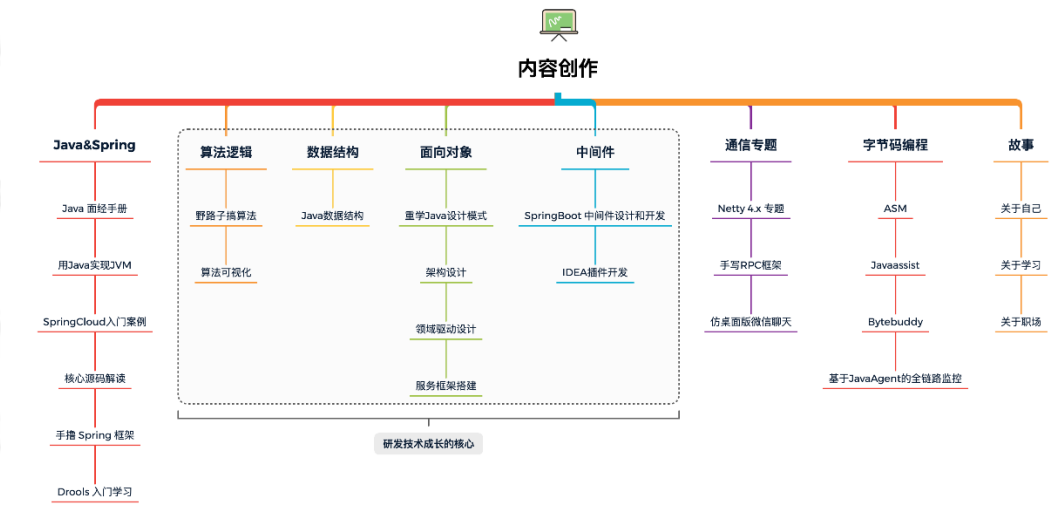
工厂模式 · 策略模式 · 观察者模式

作者

你好，我是小傅哥。《重学 Java 设计模式》图书作者，一线互联网 java 工程师、架构师，开发过交易&营销、写过运营&活动、设计过中间件也倒腾过中继器、IO 板卡。不只是写 Java 语言，也搞过 C#、PHP，是一个技术活跃的折腾者。

从 19 年开始萌生生编写技术资料料想法，以沉淀、分享、成长为核心，让自己和他人都能有所收获的想法，截止止到当前已编写的内容包括；《用 Java 实现 JVM》、《Netty4.x 专题》、《中间件开发》、《领域驱动设计》、《全链路路监控》、《字节码编程》、《中间件设计和实现》等 14 个专题共计 250 篇左右原创内容，帮助很多同学成长。同年 github 的两个项目， CodeGuide_、itstack-demo-design ，持续霸榜 Trending，成为全球热门项目。

我的作品



《重学 Java 设计模式》，这是一本互联网真实案例的实践书籍，从实际业务中抽离出，交易、营销、秒杀、中间件、源码等众多场景进行学习代码设计。

《字节码编程》，全书共计 107 页，11 万 7 千字，20 个章节涵盖三个字节码框架(ASM、Javassist、Byte-budy)和 JavaAgent 使用并附带整套案例源码！

《面经手册》，这是一本以面试为引子的核心技术讲解，全书分为 5 个章 29 节分别介绍了：面试、数据结构、算法、锁、多线程以及 JVM 的核心内容。并且

每一个章节都有对应的案例源码，读者在阅读的过程中可以参考源码进行验证实践的学习，只有这样才会让你有更多的收获。

我的技术站

1. 公众号： bugstack 虫洞栈 - 日常原创技术推文文
2. 博客： http://bugstack.cn/ - 原创技术文章汇总，适合电脑(PC)端阅读
3. Github： https://github.com/fuzhengwei/CodeGuide/wiki - 所有文文章涉及的源码汇总以及各类资料料

问题交流

- **加群交流**
  本群的宗旨是给大家提供一个良好的技术学习交流平台，所以杜绝一切广告！由于微信群人满 100 之后无法加入，请扫描下方二维码先添加作者 “小傅哥” 微信(fustack)，备注：Spring 学习加群。

- **公众号(bugstack 虫洞栈)**
  沉淀、分享、成长，专注于原创专题案例，以最易学习编程的方式分享知识，让自己和他人都能有所收获。目前已完成的专题有；Netty4.x 实战专题案例、用 Java 实现 JVM、基于 JavaAgent 的全链路监控、手写 RPC 框架、DDD 专题案例、源码分析等。

友情打赏

感谢对作者辛苦码文的支持，自愿赞赏，我会记住你的

介绍

Hello, world of spring ！ 你好，spring 的世界！
欢迎来到这里，很高兴你能拿到这本书。如果你能坚持看完书中每章节的内容，那么不仅可以在你的面试求职上有所帮助，也更能让你对 Spring 核心技术有更加深入的学习。

小傅哥，之所以开始撸 Spring 源码，主要就是因为在编写《面经手册》时，涉及到的 Spring 源码都会写很多的文字描述、绘制冗长的流程图稿、做不少的内容铺垫，但对于新人来说想直接学习这部分内容仍是非常困难的，那么现在为了让我以及更多的伙伴能有一个学习的抓手，我们来一起研究研究什么是快乐星球！

本仓库以 Spring 源码学习为目的，通过手写简化版 Spring 框架，了解 Spring 核心原理。

在手写的过程中会简化 Spring 源码，摘取整体框架中的核心逻辑，简化代码实现过程，保留核心功能，例如：IOC、AOP、Bean 生命周期、上下文、作用域、资源处理等内容实现，逐步完成整个生命周期。

---

1. 此专栏为实战编码类资料，在学习的过程中需要结合文中每个章节里，要解决的**目标**，进行的思路**设计**，带入到编码实操过程。在学习编码的同时也最好理解关于这部分内容为什么这样的实现，它用到了哪样的设计模式，采用了什么手段做了什么样的职责分离。只有通过这样的学习才能更好的理解和掌握 Spring 源码的实现过程，也能帮助你在以后的深入学习和实践应用的过程中打下一个扎实的基础。
2. 另外此专栏内容的学习上结合了设计模式，下对应了 SpringBoot 中间件设计和开发，所以读者在学习的过程中如果遇到不理解的设计模式可以翻阅相应的资料，在学习完 Spring 后还可以结合中间件的内容进行练习。
3. **源码：**此专栏涉及到的源码已经全部整合到当前工程下，可以与章节中对应的案例源码一一匹配上。大家拿到整套工程可以直接运行，也可以把每个章节对应的源码工程单独打开运行。

4. 如果你在学习的过程中遇到什么问题，包括：不能运行、优化意见、文字错误等任何问题都可以提交 issue，也可以联系作者：小傅哥 的微信，fustack
5. 在专栏的内容编写中，每一个章节都提供了清晰的设计图稿和对应的类图，所以学习过程中一定不要只是在乎代码是怎么编写的，更重要的是理解这些设计的内容是如何来的。

1. 适合人群

1. 具备一定编程基础，工作 1-3 年的研发人员
2. 想阅读 Spring 源码，但不知道从哪开始
3. 对 Spring 容器中 Bean 对象的注册管理等生命周期有些模糊
4. 需要依赖于 Spring 开发一些中间件，但不知道用哪些接口
5. 想看看设计模式在 Spring 框架下的应用
6. 希望彻底的了解 Spring 框架，并能在面试过程中占据上风

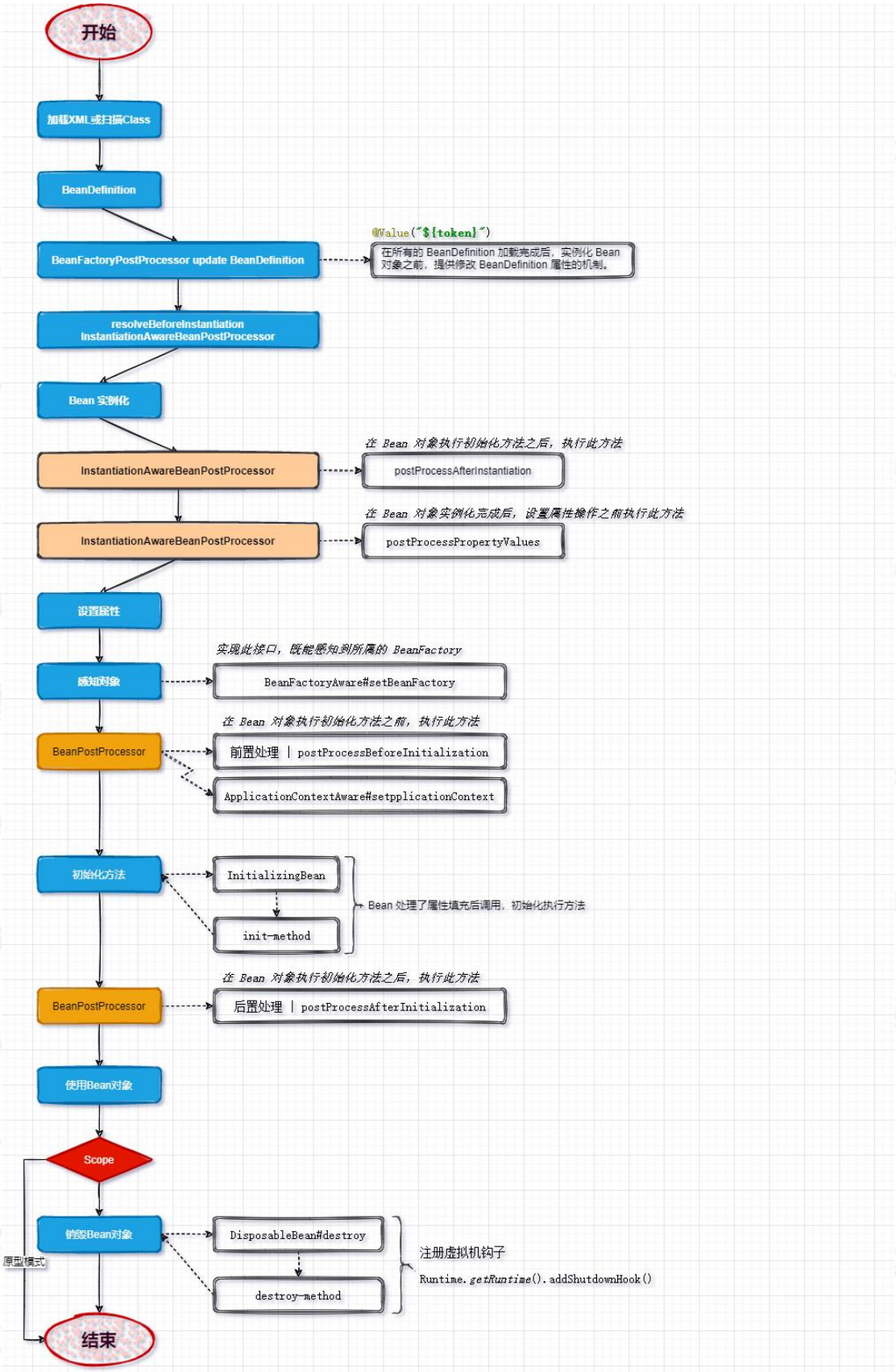
2. 我能学到什么

1. 看得懂，Bean 容器是如何定义和实现的
2. 学得会，工厂模式、策略模式、观察者模式等都是怎么在 Spring 中体现的
3. 搞得清，从应用上下文到 Bean 对象的创建，是串联出一整套生命周期
4. 弄得明，IOC、AOP、代理、切面、循环依赖都是如何设计和实现的

3. 阅读建议

此专栏是一本以开发简化版 Spring 学习其原理和内核的知识内容，不仅是代码编写实现也更注重内容上的需求分析和方案设计，所以在学习的过程要结合这些内容一起来实践，并调试对应的代码。粉丝伙伴在阅读的过程中，**千万不要害怕在学习的过程中遇到问题，这些都是正常的！** 希望你可以一直坚持把这些内容事必躬亲、亲历亲为的学完，加油！

4. 生命周期



后续的文章会陆续实现各个生命周期阶段

目录

容器篇： IOC

代理篇： AOP

高级篇：Design

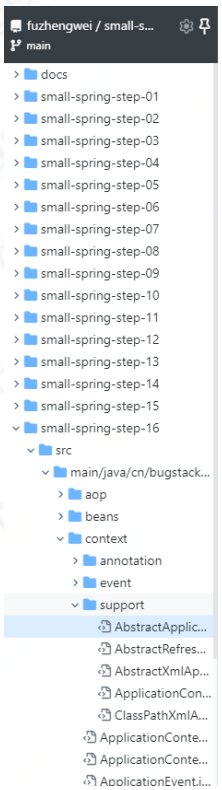
源码获取

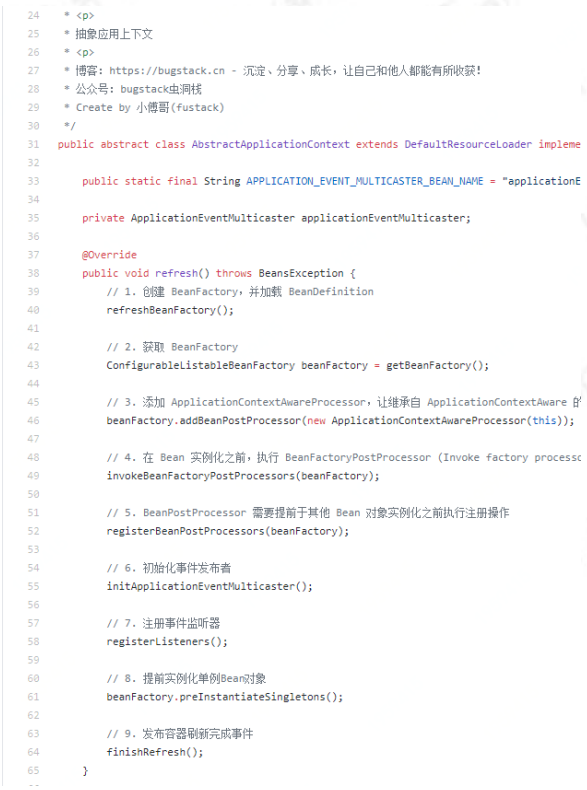
🌱《Spring 手撸专栏》，以 Spring 源码学习为目的，通过手写简化版 Spring 框架，了解 Spring 核心原理。在手写的过程中会简化 Spring 源码，摘取整体框架中的核心逻辑，简化代码实现过程，保留核心功能，例如：IOC、AOP、Bean 生命周期、上下文、作用域、资源处理等内容实现。

获取源码

1. 添加小傅哥的微信：**fustack**，备注：Spring 专栏，学习交流、加群讨论
2. 扫码关注微信公众号：**bugstack 虫洞栈**，回复：Spring 专栏

源码截图





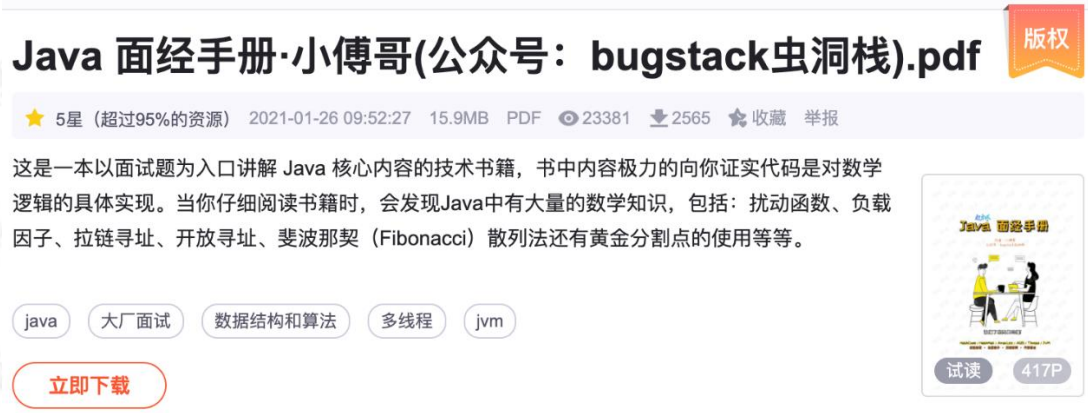
容器篇： IOC

第 01 章：开篇介绍

一、开卷有益

不正经！写写面经，去撸 Spring 源码啦？

是的，在写了 4 篇关于 Spring 核心源码的面经内容后，我决定要去手撸一个 Spring 了。为啥这么干呢？因为所有我想写的内容，都希望它是以理科思维理解为目的的学会，而不是靠着硬背记住。而目前面经中涉及到的每一篇 Spring 源码内容分析，在即使去掉部分非主流逻辑后，依然会显得非常庞大。*对有经验的老司机尚可阅读几遍接受，但就新人来讲只能放入收藏夹吃灰啦！*



可能已经阅读过 2.5k 下载量的 《Java 面经手册》的小伙伴会知晓，这是一本以面试题为入口讲解 Java 核心内容的技术书籍，书中内容极力的向你证实代码是对数学逻辑的具体实现。当你仔细阅读书籍时，会发现 Java 中有大量的数学知识，包括：扰动函数、负载因子、拉链寻址、开放寻址、斐波那契（Fibonacci）散列法还有黄金分割点的使用等等。

所以在编写面经手册关于 Spring 系列时，我也希望它是一项有益于程序员真正成长的技术资料和价值汇总，而不仅仅是对一些列繁杂内容的罗列。那么从借鉴 tiny-spring、mini-spring 以及对我对 Spring 的学习和常折腾开发中间件的经验上，来编写一款适合自己沉淀也满足于大家学习的 Spring 资料。*傅哥的面经都是“假”的，一上来就学数学、撸源码、挖核心！* 好！既然你这么说，接下来我们定义目标、计划，开始撸源码！

二、学习目标

本仓库以 Spring 源码学习为目的，通过带着读者一点点手写简化版 Spring 框架，了解 Spring 核心原理，为后续再深入学习 Spring 打下基础。
在手写的过程中会剔除 Spring 源码中繁杂的内容，摘取整体框架中的核心逻辑，简化代码实现过程，保留核心功能，例如：IOC、AOP、Bean 生命周期、上下文、作用域、资源处理等内容实现。

所有的内容实现都会由简开始，一步步带着大家实现，最终所有的内容完成后，在提供一个相对完整的 small-spring，在这个过程中只要你能跟着走下来，那么最后你一定可以较容易的阅读 Spring 源码了。

三、执行计划

原定这周已经准备了 Spring AOP 筛选通知器的相关文章，源码已经撸好了。但发现这样发下去我估计阅读量是要劈叉，多数都进收藏夹。

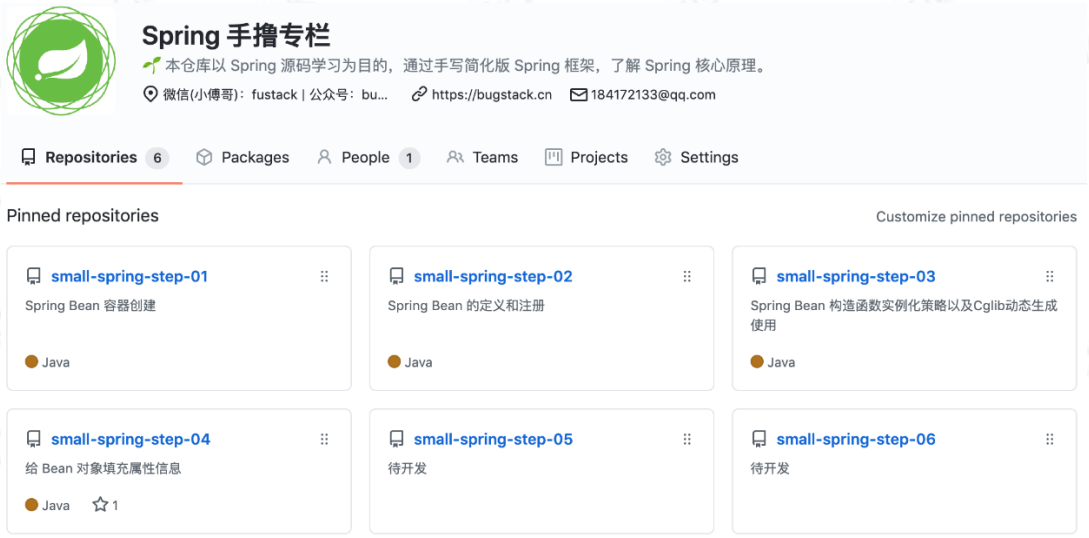
写一篇文章最大的希望是与读者互动起来，不怕你提提意见，就怕你不给三连！所有读者给出的留言、评论、点赞、分享，都是下一篇文章的 120 迈动力的开始，所以这篇文章的源码撸完后，决定该把 Spring 整理整理了，不仅让我自己有一个学习的过程沉淀感，也让读者能真的学会这部分内容。*背，那是八股文，懂，才能涨姿势！*

讲道理，其实我也是一个乐于手撸源码的主，因为从源码的学习中我可以拿到一部分在业务系统开发过程中，不太可能接触到的技术内容。而这部分从源码中学到的技术内容又可以复用到业务系统开发中，例如我写过的很多中间件以及设计模式，都来自于对框架源码的内容的挖掘和运用。

站在我的角度撸源码要比写文章快，哪怕是非常简单的知识点，也要做既不繁杂冗余的介绍，也要能把知识的广度和深度讲清楚。所以在这个过程中我也会阅读不少资料以及官网上的文档，最终把相对那些符合当前章节有价值的内容，展示给读者学习，同时这也是个人对技术内容的一个积累。

四、获取源码

本章节是整个 Spring 手撸专栏的开篇，所以这里先把源码地址以及学习使用方式交代给读者，方便后续大家在后续可以顺利的学习到这部分内容。



- 源码目录：https://github.com/fuzhengwei/small-spring - 汇总文章、源码、visio、xmind、ppt 等包括创作过程中的整理内容，方便读者学习
- 源码实现：https://github.com/small-spring - 拆解实现步骤，搭建组织工程，展示每一个章节的具体源码实现过程，如果你愿意也可以参与到工程建设中

五、总结

- 当你阅读 Spring 源码时你会看到各种的嵌套、递归、代理，以及可能连想调试时都不清楚断点要打在哪里，运行起来的程序跳来跳去。最终导致自己也就看不下去这份源码了！这是因为 Spring 发展的太久了，它为了满足不同的场景，已经做了太多的补充和优化，所以我们要做的是剥丝抽茧，体现核心，把最直接相干的内容体现出来进行学习，才更容易理解。
- 在源码学习的过程中，小傅哥会和你一起从最简单、最简单的 Bean 容器开始，可能有些时候某些章节内容并不会太多，不过我会帮你建立一些知识关联，尽可能让你在这个学习过程中，收获更多。
- 那么本章节关于 Spring 手撸专栏的开篇介绍就到这了，接下来你可以阅读到文章、获取到源码，直至我们把所有的内容全部完成，到时候就可以开发出一个相对完整的 Spring 框架了。希望在这个过程中你能和我一直坚持学习打卡！

第 02 章：创建简单的 Bean 容器

一、初步建设

上学时，老师总说：不会你就问，但多数时候都不知道要问什么！

你总会在小傅哥的文章前言里，发现一些关于成长、学习、感悟以及对当篇内容的一个介绍，其实之所以写这样的铺垫性内容，主要是为了让大家对接下来的内容学习有一个较轻松的开场和过度。

就像我们上学时如果某一科的内容不会时，老师经常会说，你有不会的就要问。但对于学生本身来讲，可能已经不会的太多了，或者压根不知道自己不会什么，只有等看到老师出完的试卷才发现自己什么都不会。但要是让问，又不知道从哪问，问出萝卜带出泥，到处都是知识漏洞。

所以我希望用一些前置内容的铺垫，让大家可以在一个稍有共识的场景下进行学习，或多或少能为你铺垫出一个稍许平缓的接受期。有可能某些时候也会打打鸡血、刺激刺激学习、总归把知识学到手就是好的！

二、目标

Spring Bean 容器是什么？

Spring 包含并管理应用对象的配置和生命周期，在这个意义上它是一种用于承载对象的容器，你可以配置你的每个 Bean 对象是如何被创建的，这些 Bean 可以创建一个单独的实例或者每次需要时都生成一个新的实例，以及它们是如何相互关联构建和使用的。

如果一个 Bean 对象交给 Spring 容器管理，那么这个 Bean 对象就应该以类似零件的方式被拆解后存放到 Bean 的定义中，这样相当于一种把对象解耦的操作，可以由 Spring 更加容易的管理，就像处理循环依赖等操作。

当一个 Bean 对象被定义存放以后，再由 Spring 统一进行装配，这个过程包括 Bean 的初始化、属性填充等，最终我们就可以完整的使用一个 Bean 实例

化后的对象了。

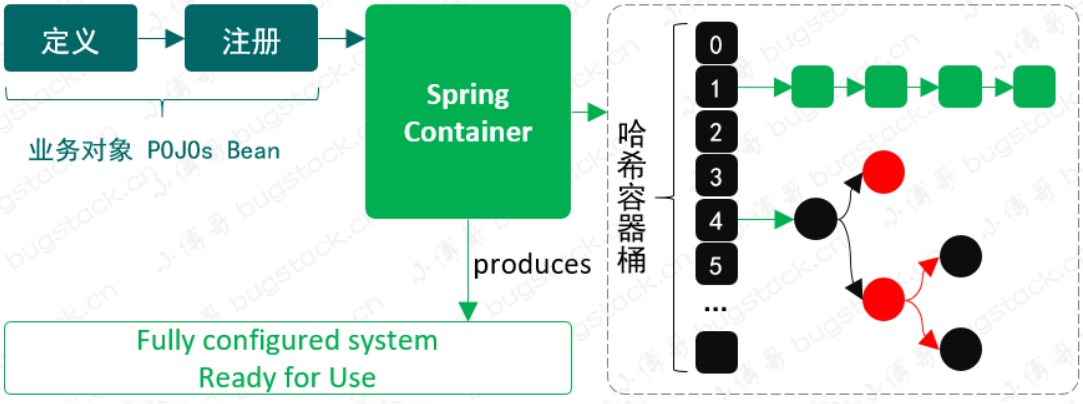
而我们本章节的案例目标就是定义一个简单的 Spring 容器，用于定义、存放和获取 Bean 对象。

三、设计

凡是可以存放数据的具体数据结构实现，都可以称之为容器。例如：ArrayList、LinkedList、HashSet 等，但在 Spring Bean 容器的场景下，我们需要一种可以用于存放和名称索引式的数据结构，所以选择 HashMap 是最合适不过的。

这里简单介绍一下 HashMap，HashMap 是一种基于扰动函数、负载因子、红黑树转换等技术内容，形成的拉链寻址的数据结构，它能让数据更加散列的分布在哈希桶以及碰撞时形成的链表和红黑树上。它的数据结构会尽可能最大限度的让整个数据读取的复杂度在 O(1) ~ O(Logn) ~O(n)之间，当然在极端情况下也会有 O(n) 链表查找数据较多的情况。不过我们经过 10 万数据的扰动函数再寻址验证测试，数据会均匀的散列在各个哈希桶索引上，所以 HashMap 非常适合用在 Spring Bean 的容器实现上。

另外一个简单的 Spring Bean 容器实现，还需 Bean 的定义、注册、获取三个基本步骤，简化设计如下；



- 定义：BeanDefinition，可能这是你在查阅 Spring 源码时经常看到的一个类，例如它会包括 singleton、prototype、BeanClassName 等。但目前我们初步实现会更加简单的处理，只定义一个 Object 类型用于存放对象。
- 注册：这个过程就相当于我们把数据存放到 HashMap 中，只不过现在 HashMap 存放的是定义了的 Bean 的对象信息。

- 获取：最后就是获取对象，Bean 的名字就是 key，Spring 容器初始化好 Bean 以后，就可以直接获取了。

接下来我们就按照这个设计，做一个简单的 Spring Bean 容器代码实现。*编码的过程往往并不会有多复杂，但知晓设计过程却更加重要！*

四、实现

1. 工程结构

```
small-spring-step-01
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── BeanDefinition.java
    │           └── BeanFactory.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   └── UserService.java
                └── ApiTest.java
```

工程源码：公众号：bugstack 虫洞栈，回复：Spring 专栏，获取整套源码

Spring Bean 容器类关系，如图 2-2

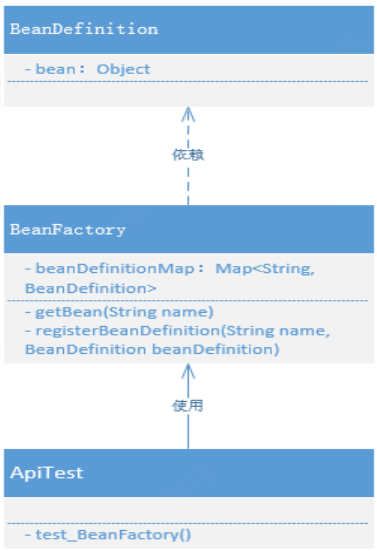


图 2-2

Spring Bean 容器的整个实现内容非常简单，也仅仅是包括了一个简单的 BeanFactory 和 BeanDefinition，这里的类名称是与 Spring 源码中一致，只不过现在的类实现会相对来说更简化一些，在后续的实现过程中再不断的添加内容。

1. BeanDefinition，用于定义 Bean 实例化信息，现在的实现是以一个 Object 存放对象
2. BeanFactory，代表了 Bean 对象的工厂，可以存放 Bean 定义到 Map 中以及获取。

2. Bean 定义

cn.bugstack.springframework.BeanDefinition

```
public class BeanDefinition {

    private Object bean;

    public BeanDefinition(Object bean) {
        this.bean = bean;
    }

    public Object getBean() {
        return bean;
    }

}
```

- 目前的 Bean 定义中，只有一个 Object 用于存放 Bean 对象。如果感兴趣可以参考 Spring 源码中这个类的信息，名称都是一样的。
- 不过在后面陆续的实现中会逐步完善 BeanDefinition 相关属性的填充，例如：SCOPE_SINGLETON、SCOPE_PROTOTYPE、ROLE_APPLICATION、ROLE_SUPPORT、ROLE_INFRASTRUCTURE 以及 Bean Class 信息。

3. Bean 工厂

cn.bugstack.springframework.BeanFactory

```
public class BeanFactory {

    private Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<>
();
```

```
    public Object getBean(String name) {
        return beanDefinitionMap.get(name).getBean();
    }

    public void registerBeanDefinition(String name, BeanDefinition beanDefinition)
{
        beanDefinitionMap.put(name, beanDefinition);
    }

}
```

- 在 Bean 工厂的实现中，包括了 Bean 的注册，这里注册的是 Bean 的定义信息。同时在这个类中还包括了获取 Bean 的操作。
- 目前的 BeanFactory 仍然是非常简化的实现，但这种简化的实现内容也是整个 Spring 容器中关于 Bean 使用的最终体现结果，只不过实现过程只展示出基本的核心原理。在后续的补充实现中，这个会不断变得庞大。

五、测试

1. 事先准备

cn.bugstack.springframework.test.bean.UserService

```
public class UserService {

    public void queryUserInfo(){
        System.out.println("查询用户信息");
    }

}
```

- 这里简单定义了一个 UserService 对象，方便我们后续对 Spring 容器测试。

2. 测试用例

cn.bugstack.springframework.test.ApiTest

```
@Test
public void test_BeanFactory(){
    // 1.初始化 BeanFactory
```

```
        BeanFactory beanFactory = new BeanFactory();

        // 2.注册 bean
        BeanDefinition beanDefinition = new BeanDefinition(new UserService());
        beanFactory.registerBeanDefinition("userService", beanDefinition);

        // 3.获取 bean
        UserService userService = (UserService) beanFactory.getBean("userService");
        userService.queryUserInfo();
    }
```

- 在单测中主要包括初始化 Bean 工厂、注册 Bean、获取 Bean，三个步骤，使用效果上贴近与 Spring，但显得会更简化。
- 在 Bean 的注册中，这里是直接把 UserService 实例化后作为入参传递给 BeanDefinition 的，在后续的陆续实现中，我们会把这部分内容放入 Bean 工厂中实现。

3. 测试结果

```
查询用户信息

Process finished with exit code 0
```

- 通过测试结果可以看到，目前的 Spring Bean 容器案例，已经稍有雏形。

六、总结

- 整篇关于 Spring Bean 容器的一个雏形就已经实现完成了，相对来说这部分代码并不会难住任何人，只要你稍加尝试就可以接受这部分内容的实现。
- 但对于一个知识的学习来说，写代码只是最后的步骤，往往整个思路、设计、方案，才更重要，只要你知道了因为什么、所以什么，才能让你有一个真正的理解。
- 下一章节会在此工程基础上扩容实现，要比现在的类多一些。不过每一篇的实现上，我都会以一个需求视角进行目标分析和方案设计，让大家在学习编码之外更能注重更多技术价值的学习。

第 03 章：实现 Bean 的定义、注册、获取

一、逻辑设计

你是否能预见复杂内容的设计问题？

讲道理，无论产品功能是否复杂，都有很大一部分程序员会写出一堆 if...else 来完成开发并顺利上线。这主要是原因没法预见当前的需求，发展是否长远、流量是否庞大、迭代是否迅速，所以在被催促上线的情况，不写 if...else 是不可能的！

那你说，既然 if...else 实现的这么快，还考虑数据结构、算法逻辑、设计模式、系统架构吗？当然这基本要看你的项目在可预见下能活多久，如果一个项目至少存活一年，并且在这一年中又会不断的的迭代。就像；你做了一个营销优惠券系统，在各种条件下发放各种类型的券，如果在最开始没有考虑好系统设计和架构模式，那么当活动频发、流量暴增、需求迭代下、最后你可能会挂在系统事故上！

我们在把系统设计的视角聚焦到具体代码实现上，你会有什么手段来实现你想要的设计模式呢？其实编码方式主要依托于：接口定义、类实现接口、抽象类实现接口、继承类、继承抽象类，而这些操作方式可以很好的隔离开每个类的基础功能、通用功能和业务功能，当类的职责清晰后，你的整个设计也会变得容易扩展和迭代。

接下来在本章节继续完善 Spring Bean 容器框架的功能开发，在这个开发过程中会用到较多的接口、类、抽象类，它们之间会有类的实现、类的继承。可以仔细参考这部分内容的开发实现，虽然并不会很复杂，但这种设计思路是完全可以复用到我们自己的业务系统开发中的。

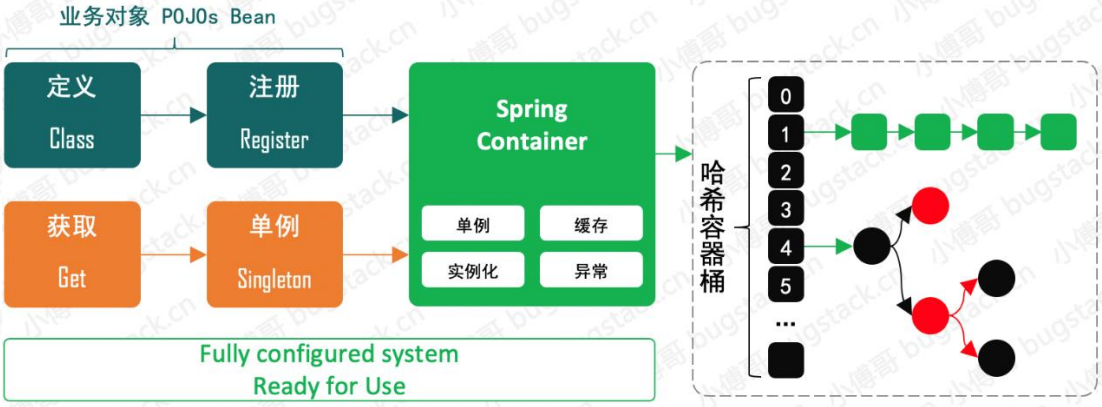
二、目标

在上一章节我们初步依照 Spring Bean 容器的概念，实现了一个粗糙版本的代码实现。那么本章节我们需要结合已实现的 Spring Bean 容器进行功能完善，实现 Bean 容器关于 Bean 对象的注册和获取。

这一次我们把 Bean 的创建交给容器，而不是我们在调用时候传递一个实例化好的 Bean 对象，另外还需要考虑单例对象，在对象的二次获取时是可以从内存中获取对象的。此外不仅要实现功能还需要完善基础容器框架的类结构体，否则将来就很难扩容进去其他的功能了。

三、设计

鉴于本章节的案例目标，我们需要将 Spring Bean 容器完善起来，首先非常重要的一点是在 Bean 注册的时候只注册一个类信息，而不会直接把实例化信息注册到 Spring 容器中。那么就需要修改 BeanDefinition 中的属性 Object 为 Class，接下来在需要做的就是在获取 Bean 对象时需要处理 Bean 对象的实例化操作以及判断当前单例对象在容器中是否已经缓存起来了。整体设计如图 3-1



- 首先我们需要定义 BeanFactory 这样一个 Bean 工厂，提供 Bean 的获取方法 `getBean(String name)`，之后这个 Bean 工厂接口由抽象类 AbstractBeanFactory 实现。这样使用**模板模式**的设计方式，可以统一收口通用核心方法的调用逻辑和标准定义，也就很好的控制了后续的实现者不用关心调用逻辑，按照统一方式执行。那么类的继承者只需要关心具体方法的逻辑实现即可。
- 那么在继承抽象类 AbstractBeanFactory 后的 AbstractAutowireCapableBeanFactory 就可以实现相应的抽象方法了，因为 AbstractAutowireCapableBeanFactory 本身也是一个抽象类，所以它只会实现属于自己的抽象方法，其他抽象方法由继承 AbstractAutowireCapableBeanFactory 的类实现。这里就体现了类实现过程中的各司其职，你只需要关心属于你的内容，不是你的内容，不要参与。*这一部分内容我们会在代码里有具体的体现*
- 另外这里还有块非常重要的知识点，就是关于单例 SingletonBeanRegistry 的接口定义实现，而 DefaultSingletonBeanRegistry 对接口实现后，会被抽象类 AbstractBeanFactory 继承。现在 AbstractBeanFactory 就是一个非常完整且强大的抽象类了，也能非常好的体现出它对模板模式的抽象定义。*接下来我们就带着这些设计层面的思考，去看代码的具体实现结果*

四、实现

1. 工程结构

```
small-spring-step-02
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework.beans
    │           ├── factory
    │           │   ├── config
    │           │   │   ├── BeanDefinition.java
    │           │   │   └── SingletonBeanRegistry.java
    │           │   ├── support
    │           │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   ├── AbstractBeanFactory.java
    │           │   │   ├── BeanDefinitionRegistry.java
    │           │   │   ├── DefaultListableBeanFactory.java
    │           │   │   └── DefaultSingletonBeanRegistry.java
    │           │   └── BeanFactory.java
    │           └── BeansException.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   └── UserService.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取源码

Spring Bean 容器类关系，如图 3-2

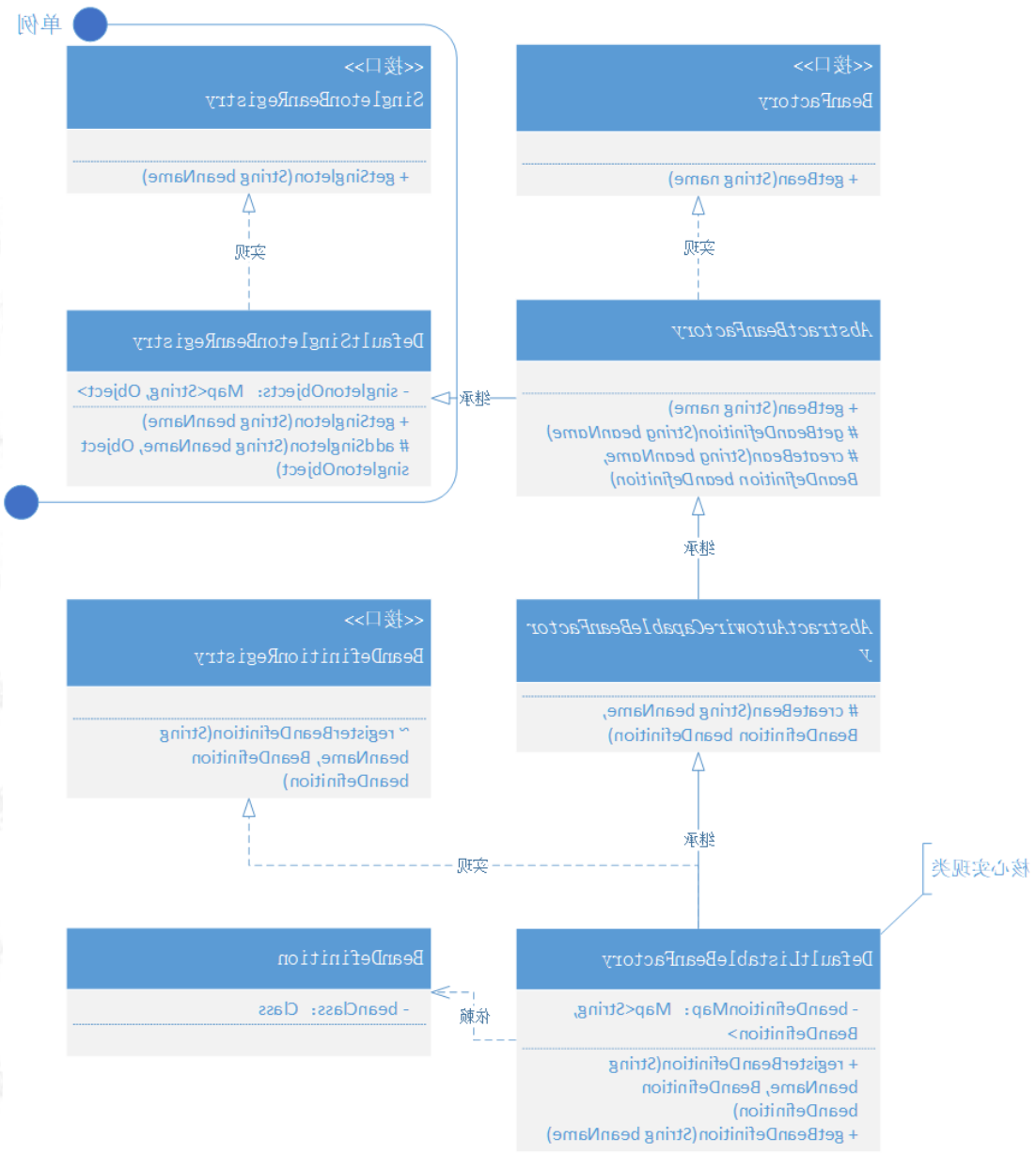


图 3-2

虽然这一章节关于 Spring Bean 容器的功能实现与 Spring 源码中还有不少的差距，但以目前实现结果的类关系图来看，其实已经具备了一定的设计复杂性，这些复杂的类关系设计在各个接口定义和实现以及在抽象类继承中都有所体现，例如：

- BeanFactory 的定义由 AbstractBeanFactory 抽象类实现接口的 getBean 方法
- 而 AbstractBeanFactory 又继承了实现了 SingletonBeanRegistry 的 DefaultSingletonBeanRegistry 类。这样 AbstractBeanFactory 抽象类就具备了单例 Bean 的注册功能。
- AbstractBeanFactory 中又定义了两个抽象方法：getBeanDefinition(String beanName)、createBean(String beanName, BeanDefinition beanDefinition) ，而这

两个抽象方法分别由 DefaultListableBeanFactory、AbstractAutowireCapableBeanFactory 实现。

- 最终 DefaultListableBeanFactory 还会继承抽象类 AbstractAutowireCapableBeanFactory 也就可以调用抽象类中的 createBean 方法了。

综上这一部分的类关系和实现过程还是会有一些复杂的，因为所有的实现都以职责划分、共性分离以及调用关系定义为标准搭建的类关系。*这部分内容的学习，可能会丰富你在复杂业务系统开发中的设计思路。*

2. BeanDefinition 定义

cn.bugstack.springframework.beans.factory.config.BeanDefinition

```
public class BeanDefinition {

    private Class beanClass;

    public BeanDefinition(Class beanClass) {
        this.beanClass = beanClass;
    }
  // ...get/set
}
```

- 在 Bean 定义类中已经把上一章节中的 Object bean 替换为 Class，这样就可以把 Bean 的实例化操作放到容器中处理了。*如果你有仔细阅读过上一章并做了相应的测试，那么你会发现 Bean 的实例化操作是放在初始化调用阶段传递给 BeanDefinition 构造函数的。*

3. 单例注册接口定义和实现

cn.bugstack.springframework.beans.factory.config.SingletonBeanRegistry

```
public interface SingletonBeanRegistry {

    Object getSingleton(String beanName);

}
```

- 这个类比较简单主要是定义了一个获取单例对象的接口。

cn.bugstack.springframework.beans.factory.config.DefaultSingletonBeanRegistry

```
public class DefaultSingletonBeanRegistry implements SingletonBeanRegistry {

    private Map<String, Object> singletonObjects = new HashMap<>();

    @Override
    public Object getSingleton(String beanName) {
        return singletonObjects.get(beanName);
    }

    protected void addSingleton(String beanName, Object singletonObject) {
        singletonObjects.put(beanName, singletonObject);
    }

}
```

- 在 DefaultSingletonBeanRegistry 中主要实现 getSingleton 方法，同时实现了一个受保护的 addSingleton 方法，这个方法可以被继承此类的其他类调用。包括：AbstractBeanFactory 以及继承的 DefaultListableBeanFactory 调用。

4. 抽象类定义模板方法(AbstractBeanFactory)

cn.bugstack.springframework.beans.factory.support.AbstractBeanFactory

```
public abstract class AbstractBeanFactory extends DefaultSingletonBeanRegistry implements BeanFactory {

    @Override
    public Object getBean(String name) throws BeansException {
        Object bean = getSingleton(name);
        if (bean != null) {
            return bean;
        }

        BeanDefinition beanDefinition = getBeanDefinition(name);
        return createBean(name, beanDefinition);
    }

    protected abstract BeanDefinition getBeanDefinition(String beanName) throws BeansException;
```

```
    protected abstract Object createBean(String beanName, BeanDefinition beanDefini
tion) throws BeansException;

}
```

- AbstractBeanFactory 首先继承了 DefaultSingletonBeanRegistry，也就具备了使用单例注册类方法。
- 接下来很重要的一点是关于接口 BeanFactory 的实现，在方法 getBean 的实现过程中可以看到，主要是对单例 Bean 对象的获取以及在获取不到时需要拿到 Bean 的定义做相应 Bean 实例化操作。那么 getBean 并没有自身的去实现这些方法，而是只定义了调用过程以及提供了抽象方法，由实现此抽象类的其他类做相应实现。
- 后续继承抽象类 AbstractBeanFactory 的类有两个，包括：AbstractAutowireCapableBeanFactory、DefaultListableBeanFactory，这两个类分别做了相应的实现处理，接着往下看。

5. 实例化 Bean 类(AbstractAutowireCapableBeanFactory)

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactor
y {

    @Override
    protected Object createBean(String beanName, BeanDefinition beanDefinition) thr
ows BeansException {
        Object bean = null;
        try {
            bean = beanDefinition.getBeanClass().newInstance();
        } catch (InstantiationException | IllegalAccessException e) {
            throw new BeansException("Instantiation of bean failed", e);
        }

        addSingleton(beanName, bean);
        return bean;
    }

}
```

- 在 AbstractAutowireCapableBeanFactory 类中实现了 Bean 的实例化操作 newInstance，其实这块会埋下一个坑，有构造函数入参的对象怎么处理？*可以提前思考*
- 在处理完 Bean 对象的实例化后，直接调用 addSingleton 方法存放到单例对象的缓存中去。

6. 核心类实现(DefaultListableBeanFactory)

cn.bugstack.springframework.beans.factory.support.DefaultListableBeanFactory

```
public class DefaultListableBeanFactory extends AbstractAutowireCapableBeanFactory
implements BeanDefinitionRegistry {

    private Map<String, BeanDefinition> beanDefinitionMap = new HashMap<>();

    @Override
    public void registerBeanDefinition(String beanName, BeanDefinition beanDefiniti
on) {
        beanDefinitionMap.put(beanName, beanDefinition);
    }

    @Override
    public BeanDefinition getBeanDefinition(String beanName) throws BeansException
{
        BeanDefinition beanDefinition = beanDefinitionMap.get(beanName);
        if (beanDefinition == null) throw new BeansException("No bean named '" + be
anName + "' is defined");
        return beanDefinition;
    }

}
```

- DefaultListableBeanFactory 在 Spring 源码中也是一个非常核心的类，在我们目前的实现中也是逐步贴近于源码，与源码类名保持一致。
- DefaultListableBeanFactory 继承了 AbstractAutowireCapableBeanFactory 类，也就具备了接口 BeanFactory 和 AbstractBeanFactory 等一连串的功能实现。*所以有时候你会看到一些类的强转，调用某些方法，也是因为你强转的类实现接口或继承了某些类。*
- 除此之外这个类还实现了接口 BeanDefinitionRegistry 中的 registerBeanDefinition(String beanName, BeanDefinition beanDefinition) 方法，当然你还会看到一个 getBeanDefinition 的实现，这个方法我们文中提到过它是抽象类 AbstractBeanFactory 中定义的抽象方法。*现在注册 Bean 定义与获取 Bean 定义*

就可以同时使用了，是不感觉这个套路还蛮深的。接口定义了注册，抽象类定义了获取，都集中在 DefaultListableBeanFactory 中的 beanDefinitionMap 里

五、测试

1. 事先准备

cn.bugstack.springframework.test.bean.UserService

```
public class UserService {

    public void queryUserInfo(){
        System.out.println("查询用户信息");
    }

}
```

- 这里简单定义了一个 UserService 对象，方便我们后续对 Spring 容器测试。

2. 测试用例

cn.bugstack.springframework.test.ApiTest

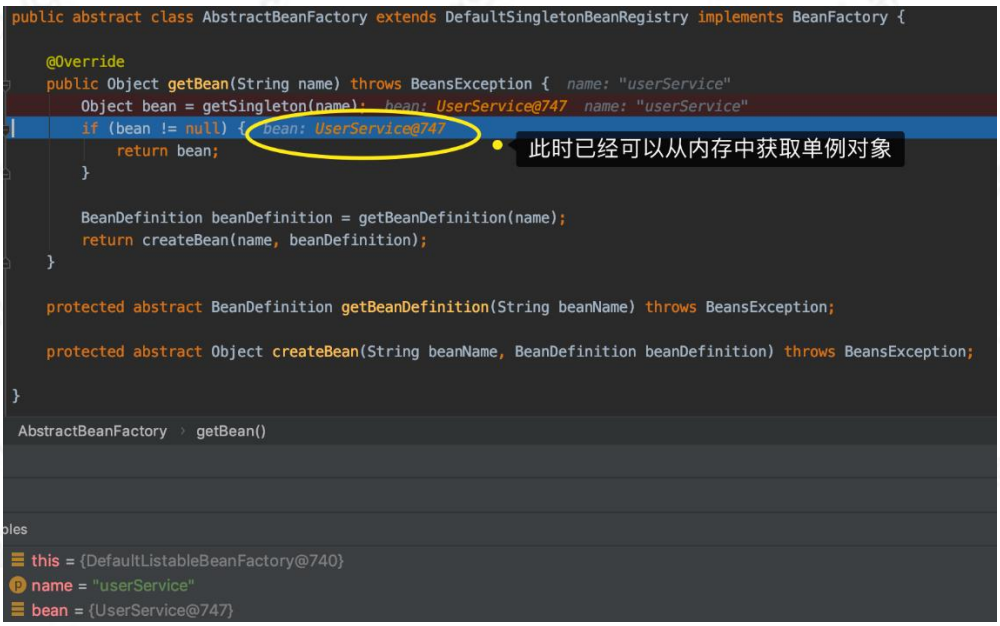
```
@Test
public void test_BeanFactory(){
    // 1.初始化 BeanFactory
    DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();
    // 2.注册 bean
    BeanDefinition beanDefinition = new BeanDefinition(UserService.class);
    beanFactory.registerBeanDefinition("userService", beanDefinition);
    // 3.第一次获取 bean
    UserService userService = (UserService) beanFactory.getBean("userService");
    userService.queryUserInfo();
    // 4.第二次获取 bean from Singleton
    UserService userService_singleton = (UserService) beanFactory.getBean("userServ
ice");
    userService_singleton.queryUserInfo();
}
```

- 在此次的单元测试中除了包括；Bean 工厂、注册 Bean、获取 Bean，三个步骤，还额外增加了一次对象的获取和调用。这里主要测试验证单例对象的是否正确的存放到了缓存中。
- 此外与上一章节测试过程中不同的是，我们把 UserService.class 传递给了 BeanDefinition 而不是像上一章节那样直接 new UserService() 操作。

3. 测试结果

```
查询用户信息
查询用户信息

Process finished with exit code 0
```

- 这里会有两次测试信息，一次是获取 Bean 时直接创建的对象，另外一次是从缓存中获取的实例化对象。
- 此外从调试的截图中也可以看到第二次获取单例对象，已经可以从内存中获取了，如图 3-3



- 到这本章节的功能实现和测试验证就完成了，关于测试过程中可以再去断点调试下各个阶段类的调用，熟悉调用关系。

六、总结

- 相对于前一章节对 Spring Bean 容器的简单概念实现，本章节中加强了功能的完善。在实现的过程中也可以看到类的关系变得越来越多了，如果没有做过一些稍微复杂的系统类系统，那么即使现在这样 9 个类搭出来的容器工厂也可以给你绕晕。

- 在 Spring Bean 容器的实现类中要重点关注类之间的职责和关系，几乎所有的程序功能设计都离不开接口、抽象类、实现、继承，而这些不同特性类的使用就可以非常好的隔离开类的功能职责和作用范围。而这样的知识点也是在学习手写 Spring Bean 容器框架过程非常重要的知识。
- 最后要强调一下关于整个系列内容的学习，可能在学习的过程中会遇到像第二章节那样非常简单的代码实现，但要做一个有成长的程序员要记住代码实现只是最后的落地结果，而那些设计上的思考才是最有价值的地方。*就像你是否遇到过，有人让你给一个内容做个描述、文档、说明，你总觉得太简单了没什么可写的，即使要动笔写了也不知道要从哪开始！其实这些知识内容都来源你对整体功能的理解，这就不只是代码开发还包括了需求目标、方案设计、技术实现、逻辑验证等等过程性的内容。所以，不要只是被看似简单的内容忽略了整体全局观，要学会放开视野，开放学习视角。*

第 04 章：对象实例化策略

一、查漏补缺

技术成长，是对场景设计细节不断的雕刻！

你觉得自己的技术什么时候得到了快速的提高，是 CRUD 写的多了以后吗？想都不要想，绝对不可能！CRUD 写的再多也只是能满足你作为一个搬砖工具人，敲击少逻辑流水代码的速度而已，而编程能力这一块，除了最开始的从不熟练到熟练以外，就很少再有其他提升了。

那你可能会想什么才是编程能力提升？其实更多的编程能力的提升是你对复杂场景的架构把控以及对每一个技术实现细节点的不断用具有规模体量的流量冲击验证时，是否能保证系统稳定运行从而决定你见识了多少、学到了多少、提升了多少！

最终当你在接一个产品需求时，开始思考程序数据结构的设计、核心功能的算法逻辑实现、整体服务的设计模式使用、系统架构的搭建方式、应用集群的部署结构，那么也就是的编程能力真正提升的时候！

二、目标

这一章节的目标主要是为了解决上一章节我们埋下的坑，那是什么坑呢？其实就是一个关于 Bean 对象在含有构造函数进行实例化的坑。

在上一章节我们扩充了 Bean 容器的功能，把实例化对象交给容器来统一处理，但在我们实例化对象的代码里并没有考虑对象类是否含构造函数，也就是说如果我们去实例化一个含有构造函数的对象那么就要抛异常了。

怎么验证？其实就是把 UserService 添加一个含入参信息的构造函数就可以，如下：

```
public class UserService {

    private String name;
```

```
    public UserService(String name) {
        this.name = name;
    }

    // ...
}
```

报错如下：

```
java.lang.InstantiationException: cn.bugstack.springframework.test.bean.UserService

 at java.lang.Class.newInstance(Class.java:427)
 at cn.bugstack.springframework.test.ApiTest.test_newInstance(ApiTest.java:51)
 ...
```

发生这一现象的主要原因就是因为 `beanDefinition.getBeanClass().newInstance();` 实例化方式并没有考虑构造函数的入参，所以就这个坑就在这等着你了！**那么我们的目标就很明显了，来把这个坑填平！**

三、设计

填平这个坑的技术设计主要考虑两部分，一个是串流程从哪合理的把构造函数的入参信息传递到实例化操作里，另外一个是怎么去实例化含有构造函数的对象。

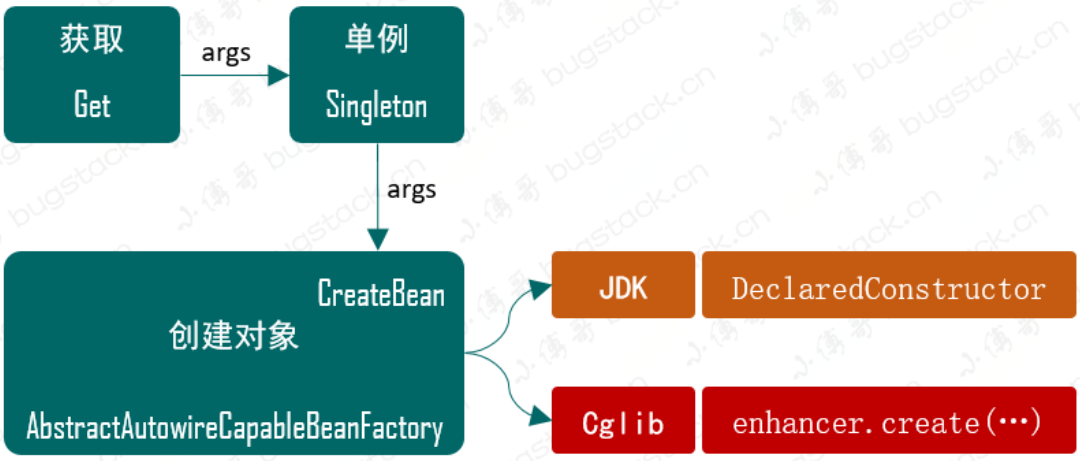


图 4-1

- 参考 Spring Bean 容器源码的实现方式，在 BeanFactory 中添加 `Object getBean(String name, Object... args)` 接口，这样就可以在获取 Bean 时把构造函数的入参信息传递进去了。

- 另外一个核心的内容是使用什么方式来创建含有构造函数的 Bean 对象呢？这里有两种方式可以选择，一个是基于 Java 本身自带的方法 `DeclaredConstructor`，另外一个是使用 Cglib 来动态创建 Bean 对象。*Cglib 是基于字节码框架 ASM 实现，所以你也可以直接通过 ASM 操作指令码来创建对象*

四、实现

1. 工程结构

```
small-spring-step-03
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework.beans
    │           ├── factory
    │           │   ├── config
    │           │   │   ├── BeanDefinition.java
    │           │   │   └── SingletonBeanRegistry.java
    │           │   ├── support
    │           │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   ├── AbstractBeanFactory.java
    │           │   │   ├── BeanDefinitionRegistry.java
    │           │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   ├── DefaultListableBeanFactory.java
    │           │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   ├── InstantiationStrategy.java
    │           │   │   └── SimpleInstantiationStrategy.java
    │           │   └── BeanFactory.java
    │           └── BeansException.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   └── UserService.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

Spring Bean 容器类关系，如图 4-2

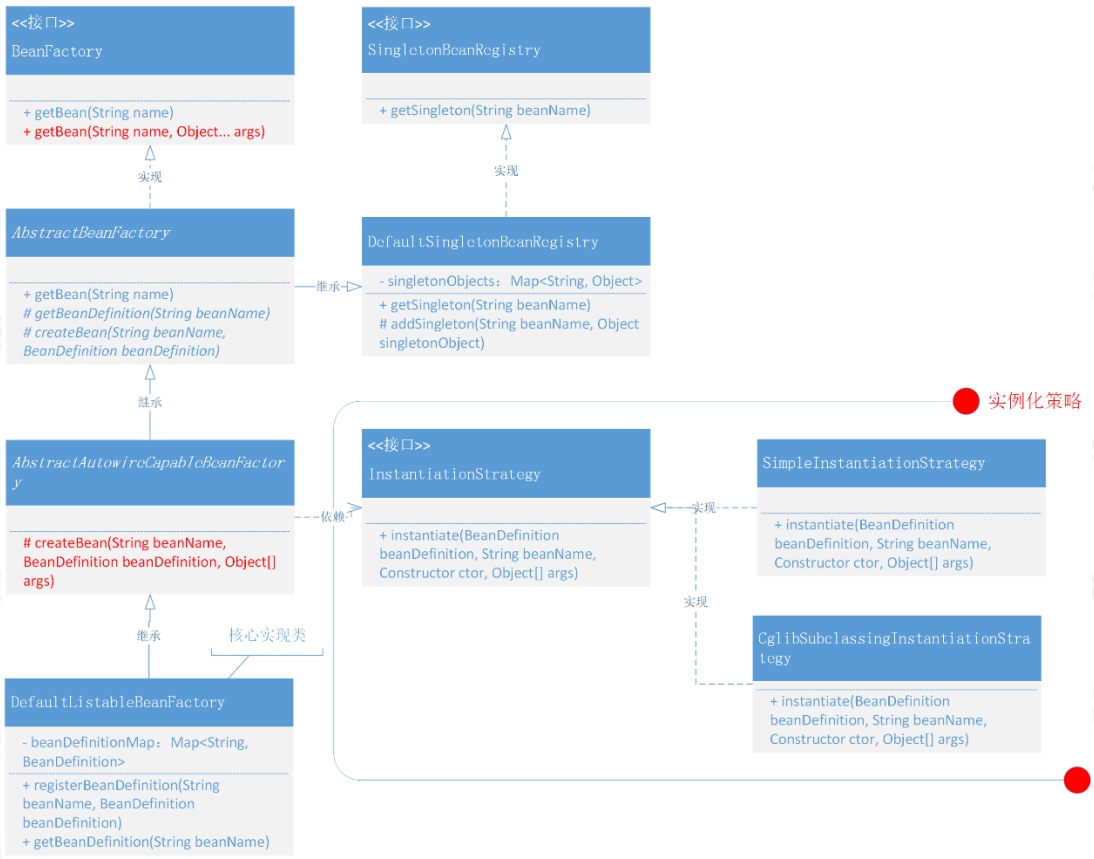


图 4-2

本章节“填坑”主要是在现有工程中添加 InstantiationStrategy 实例化策略接口，以及补充相应的 getBean 入参信息，让外部调用时可以传递构造函数的入参并顺利实例化。

2. 新增 getBean 接口

cn.bugstack.springframework.beans.factory.BeanFactory

```
public interface BeanFactory {

    Object getBean(String name) throws BeansException;

    Object getBean(String name, Object... args) throws BeansException;

}
```

- BeanFactory 中我们重载了一个含有入参信息 args 的 getBean 方法，这样就可以方便的传递入参给构造函数实例化了。

3. 定义实例化策略接口

cn.bugstack.springframework.beans.factory.support.InstantiationStrategy

```
public interface InstantiationStrategy {

    Object instantiate(BeanDefinition beanDefinition, String beanName, Constructor ctor, Object[] args) throws BeansException;

}
```

- 在实例化接口 instantiate 方法中添加必要的入参信息，包括：beanDefinition、beanName、ctor、args
- 其中 Constructor 你可能会有一点陌生，它是 java.lang.reflect 包下的 Constructor 类，里面包含了一些必要的类信息，有这个参数的目的就是为了拿到符合入参信息相对应的构造函数。
- 而 args 就是一个具体的入参信息了，最终实例化时候会用到。

4. JDK 实例化

cn.bugstack.springframework.beans.factory.support.SimpleInstantiationStrategy

```
public class SimpleInstantiationStrategy implements InstantiationStrategy {

    @Override
    public Object instantiate(BeanDefinition beanDefinition, String beanName, Constructor ctor, Object[] args) throws BeansException {
        Class clazz = beanDefinition.getBeanClass();
        try {
            if (null != ctor) {
                return clazz.getDeclaredConstructor(ctor.getParameterTypes()).newInstance(args);
            } else {
                return clazz.getDeclaredConstructor().newInstance();
            }
        } catch (NoSuchMethodException | InstantiationException | IllegalAccessException | InvocationTargetException e) {
            throw new BeansException("Failed to instantiate [" + clazz.getName() + "]", e);
        }
    }
```

```
}
```

- 首先通过 beanDefinition 获取 Class 信息，这个 Class 信息是在 Bean 定义的时候传递进去的。
- 接下来判断 ctor 是否为空，如果为空则是无构造函数实例化，否则就是需要有构造函数的实例化。
- 这里我们重点关注有构造函数的实例化，实例化方式为 `clazz.getDeclaredConstructor(ctor.getParameterTypes()).newInstance(args);`，把入参信息传递给 newInstance 进行实例化。

5. Cglib 实例化

cn.bugstack.springframework.beans.factory.support.CglibSubclassingInstantiationStrategy

```
public class CglibSubclassingInstantiationStrategy implements InstantiationStrategy
 {

    @Override
    public Object instantiate(BeanDefinition beanDefinition, String beanName, Const
ructor ctor, Object[] args) throws BeansException {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(beanDefinition.getBeanClass());
        enhancer.setCallback(new NoOp() {
            @Override
            public int hashCode() {
                return super.hashCode();
            }
        });
        if (null == ctor) return enhancer.create();
        return enhancer.create(ctor.getParameterTypes(), args);
    }

}
```

- 其实 Cglib 创建有构造函数的 Bean 也非常方便，在这里我们更加简化的处理了，如果你阅读 Spring 源码还会看到 CallbackFilter 等实现，不过我们目前的方式并不会影响创建。

6. 创建策略调用

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactor
y {

    private InstantiationStrategy instantiationStrategy = new CglibSubclassingInsta
ntiationStrategy();

    @Override
    protected Object createBean(String beanName, BeanDefinition beanDefinition, Obj
ect[] args) throws BeansException {
        Object bean = null;
        try {
            bean = createBeanInstance(beanDefinition, beanName, args);
        } catch (Exception e) {
            throw new BeansException("Instantiation of bean failed", e);
        }

        addSingleton(beanName, bean);
        return bean;
    }

    protected Object createBeanInstance(BeanDefinition beanDefinition, String beanN
ame, Object[] args) {
        Constructor constructorToUse = null;
        Class<?> beanClass = beanDefinition.getBeanClass();
        Constructor<?>[] declaredConstructors = beanClass.getDeclaredConstructors()
;
        for (Constructor ctor : declaredConstructors) {
            if (null != args && ctor.getParameterTypes().length == args.length) {
                constructorToUse = ctor;
                break;
            }
        }
        return getInstantiationStrategy().instantiate(beanDefinition, beanName, con
structorToUse, args);
    }

}
```

- 首先在 AbstractAutowireCapableBeanFactory 抽象类中定义了一个创建对象的实例化策略属性类 `InstantiationStrategy instantiationStrategy`，这里我们选择了 Cglib 的实现类。
- 接下来抽取 `createBeanInstance` 方法，在这个方法中需要注意 Constructor 代表了你有多少个构造函数，通过 beanClass.getDeclaredConstructors() 方式可以获取到你所有的构造函数，是一个集合。
- 接下来就需要循环比对出构造函数集合与入参信息 `args` 的匹配情况，这里我们对比的方式比较简单，只是一个数量对比，而实际 Spring 源码中还需要比对入参类型，否则相同数量不同入参类型的情况，就会抛异常了。

五、测试

1. 事先准备

cn.bugstack.springframework.test.bean.UserService

```
public class UserService {

    private String name;

    public UserService(String name) {
        this.name = name;
    }

    public void queryUserInfo() {
        System.out.println("查询用户信息：" + name);
    }

    @Override
    public String toString() {
        final StringBuilder sb = new StringBuilder("");
        sb.append("").append(name);
        return sb.toString();
    }
}
```

- 这里唯一多在 UserService 中添加的就是一个有 name 入参的构造函数，方便我们验证这样的对象是否能被实例化。

2. 测试用例

cn.bugstack.springframework.test.ApiTest

```
@Test
public void test_BeanFactory() {
    // 1.初始化 BeanFactory
    DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();

    // 2. 注入 bean
    BeanDefinition beanDefinition = new BeanDefinition(UserService.class);
    beanFactory.registerBeanDefinition("userService", beanDefinition);

    // 3.获取 bean
    UserService userService = (UserService) beanFactory.getBean("userService", "小傅
哥");
    userService.queryUserInfo();
}
```

- 在此次的单元测试中除了包括；Bean 工厂、注册 Bean、获取 Bean，三个步骤，还额外增加了一次对象的获取和调用。这里主要测试验证单例对象的是否正确的存放到了缓存中。
- 此外与上一章节测试过程中不同的是，我们把 UserService.class 传递给了 BeanDefinition 而不是像上一章节那样直接 new UserService() 操作。

3. 测试结果

```
查询用户信息：小傅哥

Process finished with exit code 0
```

- 从测试结果来看，最大的变化就是可以满足带有构造函数的对象，可以被实例化了。
- 你可以尝试分别使用两种不同的实例化策略，来进行实例化。SimpleInstantiationStrategy、CglibSubclassingInstantiationStrategy

4. 操作案例

这里我们再把几种不同方式的实例化操作，放到单元测试中，方便大家比对学习。

4.1 无构造函数

```
@Test
public void test_newInstance() throws IllegalAccessException, InstantiationExceptio
n {
    UserService userService = UserService.class.newInstance();
    System.out.println(userService);
}
```

- 这种方式的实例化也是我们在上一章节实现 Spring Bean 容器时直接使用的方式

4.2 验证有构造函数实例化

```
@Test
public void test_constructor() throws Exception {
    Class<UserService> userServiceClass = UserService.class;
    Constructor<UserService> declaredConstructor = userServiceClass.getDeclaredCons
tructor(String.class);
    UserService userService = declaredConstructor.newInstance("小傅哥");
    System.out.println(userService);
}
```

- 从最简单的操作来看，如果有构造函数的类需要实例化时，则需要使用 `getDeclaredConstructor` 获取构造函数，之后在通过传递参数进行实例化。

4.3 获取构造函数信息

```
@Test
public void test_parameterTypes() throws Exception {
    Class<UserService> beanClass = UserService.class;
    Constructor<?>[] declaredConstructors = beanClass.getDeclaredConstructors();
    Constructor<?> constructor = declaredConstructors[0];
    Constructor<UserService> declaredConstructor = beanClass.getDeclaredConstructor
(constructor.getParameterTypes());
    UserService userService = declaredConstructor.newInstance("小傅哥");
    System.out.println(userService);
```

- 这个案例中其实最核心的点在于获取一个类中所有的构造函数，其实也就是这个方法的使用 `beanClass.getDeclaredConstructors()`

4.4 Cglib 实例化

```
@Test
public void test_cglib() {
    Enhancer enhancer = new Enhancer();
    enhancer.setSuperclass(UserService.class);
    enhancer.setCallback(new NoOp() {
        @Override
        public int hashCode() {
            return super.hashCode();
        }
    });
    Object obj = enhancer.create(new Class[]{String.class}, new Object[]{"小傅哥"});
    System.out.println(obj);
}
```

- 此案例演示使用非常简单，但关于 Cglib 在 Spring 容器中的使用非常多，也可以深入的学习一下 Cglib 的扩展知识。

六、总结

- 本章节的主要以完善实例化操作，增加 InstantiationStrategy 实例化策略接口，并新增了两个实例化类。这部分类的名称与实现方式基本是 Spring 框架的一个缩小版，大家在学习过程中也可以从 Spring 源码找到对应的代码。
- 从我们不断的完善增加需求可以看到的，当你的代码结构设计的较为合理的时候，就可以非常容易且方便的进行扩展不同属性的类职责，而不会因为需求的增加导致类结构混乱。所以在我们自己业务需求实现的过程中，也要尽可能的去考虑一个良好的扩展性以及拆分好类的职责。
- 动手是学习起来最快的方式，不要让眼睛是感觉看会了，但上手操作就废了。也希望有需要的读者可以亲手操作一下，把你的想法也融入到可落地实现的代码里，看看想的和做的是否一致。

第 05 章：注入属性和依赖对象

一、功能完善

超卖、掉单、幂等，你的程序总是不抗揍！

想想，运营已经对外宣传了七八天的活动，满心欢喜的等着最后一天页面上线对外了，突然出现了一堆异常、资损、闪退，而用户流量稍纵即逝，最后想死的心都有！

就编程开发来讲，丢三落四、乱码七糟，可能这就是大部分初级程序员日常开发的真实写照，在即使有测试人员验证的情况下，也会出现带 Bug 上线的现象，只不过是当时没有发现而已！*因为是人写代码，就一定会有错误，即使是老码农*

就程序 Bug 来讲，会包括产品 PRD 流程上的 Bug、运营配置活动时候的 Bug、研发开发时功能实现的 Bug、测试验证时漏掉流程的 Bug、上线过程中运维服务相关配置的 Bug，而这些其实都可以通过制定的流程规范和一定的研发经验积累，慢慢尽可能减少。

而另外一类是沟通留下的 Bug，通常情况下业务提需求、产品定方案、研发做实现，最终还要有 UI、测试、运营、架构等等各个环节的人员参与到一个项目的承接、开发到上线运行，而在这一群人需要保持一个统一的信息传播其实是很难的。比如在项目开发中期，运营给产品说了一个新增的需求，产品觉得功能也不大，随即找到对应的前端研发加个逻辑，但没想到可能也影响到了后端的开发和测试的用例。最后功能虽然是上线了，可并不在整个产研测的需求覆盖度范围里，也就隐形的埋下了一个坑。

所以，如果你想让你的程序很抗揍，接的住农夫三拳，那么你要做的就不只是一个单纯的搬砖码农！

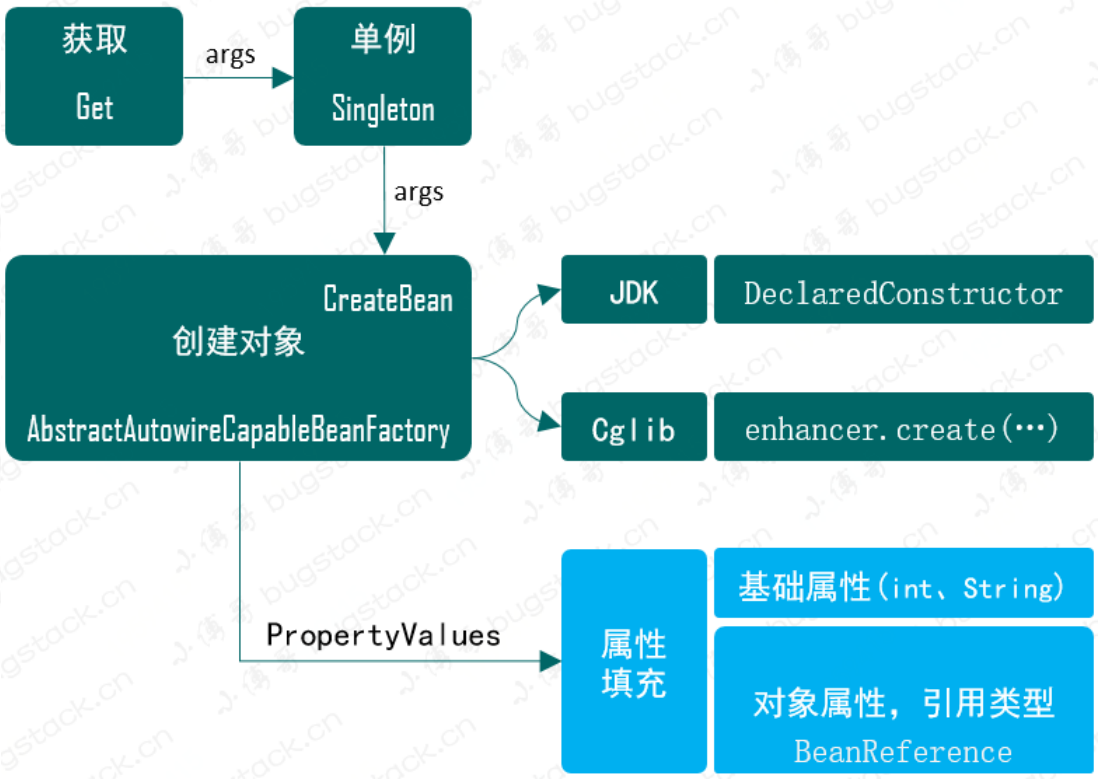
二、目标

首先我们回顾下这几章节都完成了什么，包括：实现一个容器、定义和注册 Bean、实例化 Bean，按照是否包含构造函数实现不同的实例化策略，那么在创建对象实例化这我们还缺少什么？其实还缺少一个关于类中是否有属性的问题，如果有类中包含属性那么在实例化的时候就需要把属性信息填充上，这样才是一个完整的对象创建。

对于属性的填充不只是 int、Long、String，还包括还没有实例化的对象属性，都需要在 Bean 创建时进行填充操作。*不过这里我们暂时不会考虑 Bean 的循环依赖，否则会把整个功能实现撑大，这样新人学习时就把握不住了，待后续陆续先把核心功能实现后，再逐步完善*

三、设计

鉴于属性填充是在 Bean 使用 `newInstance` 或者 `Cglib` 创建后，开始补全属性信息，那么就可以在类 `AbstractAutowireCapableBeanFactory` 的 createBean 方法中添加补全属性方法。*这部分大家在实习的过程中也可以对照 Spring 源码学习，这里的实现也是 Spring 的简化版，后续对照学习会更加易于理解*



- 属性填充要在类实例化创建之后，也就是需要在 `AbstractAutowireCapableBeanFactory` 的 createBean 方法中添加 `applyPropertyValues` 操作。
- 由于我们需要在创建 Bean 时候填充属性操作，那么就需要在 bean 定义 BeanDefinition 类中，添加 PropertyValues 信息。
- 另外是填充属性信息还包括了 Bean 的对象类型，也就是需要再定义一个 BeanReference，里面其实就是一个简单的 Bean 名称，在具体的实例化操作时进行递归创建和填充，与 Spring 源码实现一样。*Spring 源码中 BeanReference 是一个接口*

四、实现

1. 工程结构

```
small-spring-step-04
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework.beans
    │           ├── factory
    │           │   ├── config
    │           │   │   ├── BeanDefinition.java
    │           │   │   ├── BeanReference.java
    │           │   │   └── SingletonBeanRegistry.java
    │           │   ├── support
    │           │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   ├── AbstractBeanFactory.java
    │           │   │   ├── BeanDefinitionRegistry.java
    │           │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   ├── DefaultListableBeanFactory.java
    │           │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   ├── InstantiationStrategy.java
    │           │   │   └── SimpleInstantiationStrategy.java
    │           │   └── BeanFactory.java
    │           ├── BeansException.java
    │           ├── PropertyValue.java
    │           └── PropertyValues.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── UserDao.java
```

```
│   └── UserService.java
└── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

Spring Bean 容器类关系，如图 5-2

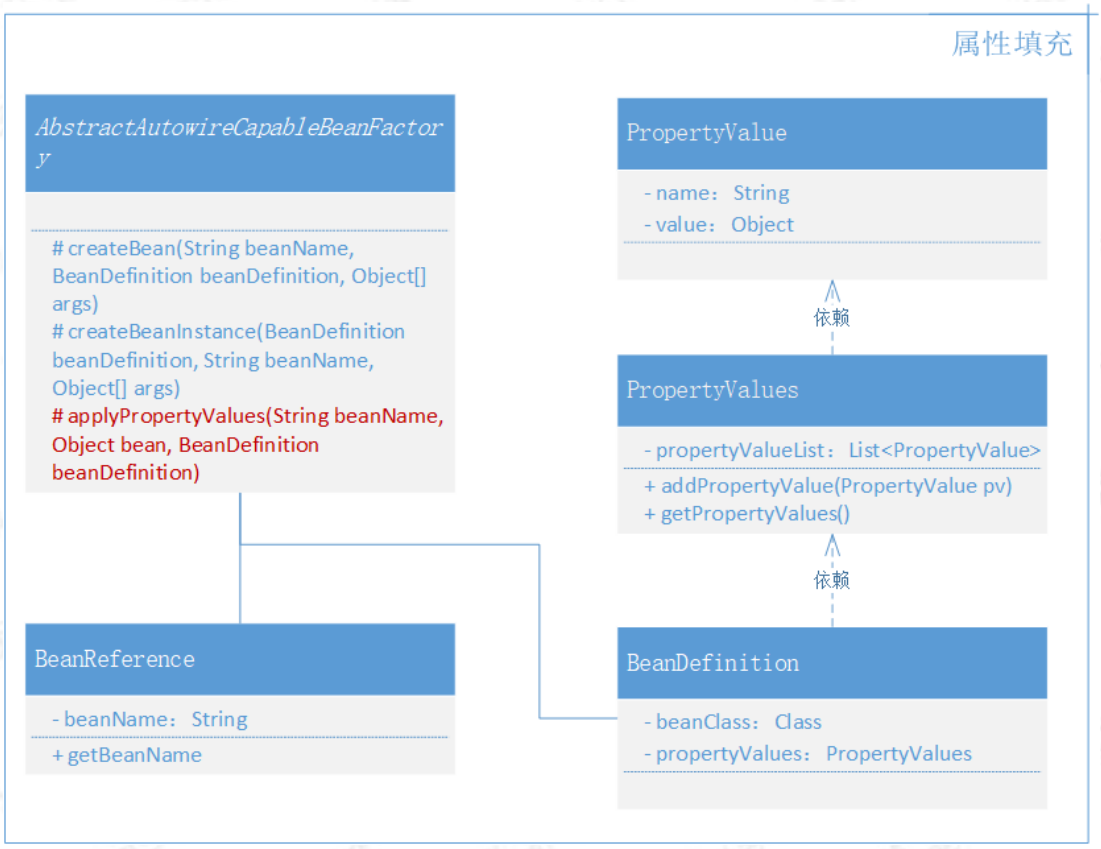


图 5-2

- 本章节中需要新增加 3 个类，BeanReference(类引用)、PropertyValue(属性值)、PropertyValues(属性集合)，分别用于类和其他类型属性填充操作。
- 另外改动的类主要是 AbstractAutowireCapableBeanFactory，在 createBean 中补全属性填充部分。

2. 定义属性

cn.bugstack.springframework.beans.PropertyValue

```
public class PropertyValue {

    private final String name;

    private final Object value;

    public PropertyValue(String name, Object value) {
```

```
        this.name = name;
        this.value = value;
    }

    // ...get/set
}
```

cn.bugstack.springframework.beans.PropertyValues

```
public class PropertyValues {

    private final List<PropertyValue> propertyValueList = new ArrayList<>();

    public void addPropertyValue(PropertyValue pv) {
        this.propertyValueList.add(pv);
    }

    public PropertyValue[] getPropertyValues() {
        return this.propertyValueList.toArray(new PropertyValue[0]);
    }

    public PropertyValue getPropertyValue(String propertyName) {
        for (PropertyValue pv : this.propertyValueList) {
            if (pv.getName().equals(propertyName)) {
                return pv;
            }
        }
        return null;
    }

}
```

- 这两个类的作用就是创建出一个用于传递类中属性信息的类，因为属性可能会有很多，所以还需要定义一个集合包装下。

3. Bean 定义补全

cn.bugstack.springframework.beans.factory.config.BeanDefinition

```
public class BeanDefinition {

    private Class beanClass;

    private PropertyValues propertyValues;
```

```
    public BeanDefinition(Class beanClass) {
        this.beanClass = beanClass;
        this.propertyValues = new PropertyValues();
    }

    public BeanDefinition(Class beanClass, PropertyValues propertyValues) {
        this.beanClass = beanClass;
        this.propertyValues = propertyValues != null ? propertyValues : new Propert
yValues();
    }

    // ...get/set
}
```

- 在 Bean 注册的过程中是需要传递 Bean 的信息，在几个前面章节的测试中都有所体现 `new BeanDefinition(UserService.class, propertyValues);`
- 所以为了把属性一定交给 Bean 定义，所以这里填充了 PropertyValues 属性，同时把两个构造函数做了一些简单的优化，避免后面 for 循环时还得判断属性填充是否为空。

4. Bean 属性填充

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactor
y {

    private InstantiationStrategy instantiationStrategy = new CglibSubclassingInsta
ntiationStrategy();

    @Override
    protected Object createBean(String beanName, BeanDefinition beanDefinition, Obj
ect[] args) throws BeansException {
        Object bean = null;
        try {
            bean = createBeanInstance(beanDefinition, beanName, args);
            // 给 Bean 填充属性
            applyPropertyValues(beanName, bean, beanDefinition);
        } catch (Exception e) {
            throw new BeansException("Instantiation of bean failed", e);
```

```
        }

        addSingleton(beanName, bean);
        return bean;
    }

    protected Object createBeanInstance(BeanDefinition beanDefinition, String beanN
ame, Object[] args) {
        Constructor constructorToUse = null;
        Class<?> beanClass = beanDefinition.getBeanClass();
        Constructor<?>[] declaredConstructors = beanClass.getDeclaredConstructors()
;
        for (Constructor ctor : declaredConstructors) {
            if (null != args && ctor.getParameterTypes().length == args.length) {
                constructorToUse = ctor;
                break;
            }
        }
        return getInstantiationStrategy().instantiate(beanDefinition, beanName, con
structorToUse, args);
    }

    /**
     * Bean 属性填充
     */
    protected void applyPropertyValues(String beanName, Object bean, BeanDefinition
 beanDefinition) {
        try {
            PropertyValues propertyValues = beanDefinition.getPropertyValues();
            for (PropertyValue propertyValue : propertyValues.getPropertyValues())
{

                String name = propertyValue.getName();
                Object value = propertyValue.getValue();

                if (value instanceof BeanReference) {
                    // A 依赖 B，获取 B 的实例化
                    BeanReference beanReference = (BeanReference) value;
                    value = getBean(beanReference.getBeanName());
                }
                // 属性填充
                BeanUtil.setFieldValue(bean, name, value);
            }
        } catch (Exception e) {
```

```
            throw new BeansException("Error setting property values: " + beanName);
        }
    }

    public InstantiationStrategy getInstantiationStrategy() {
        return instantiationStrategy;
    }

    public void setInstantiationStrategy(InstantiationStrategy instantiationStrateg
y) {
        this.instantiationStrategy = instantiationStrategy;
    }

}
```

- 这个类的内容稍微有点长，主要包括三个方法：createBean、createBeanInstance、applyPropertyValues，这里我们主要关注 createBean 的方法中调用的 applyPropertyValues 方法。
- 在 applyPropertyValues 中，通过获取 `beanDefinition.getPropertyValues()` 循环进行属性填充操作，如果遇到的是 BeanReference，那么就需要递归获取 Bean 实例，调用 getBean 方法。
- 当把依赖的 Bean 对象创建完成后，会递归回现在属性填充中。这里需要注意我们并没有去处理循环依赖的问题，这部分内容较大，后续补充。
  BeanUtil.setFieldValue(bean, name, value) 是 hutool-all 工具类中的方法，你也可以自己实现

五、测试

1. 事先准备

cn.bugstack.springframework.test.bean.UserDao

```
public class UserDao {

    private static Map<String, String> hashMap = new HashMap<>();

    static {
        hashMap.put("10001", "小傅哥");
        hashMap.put("10002", "八杯水");
        hashMap.put("10003", "阿毛");
    }
```

```
    public String queryUserName(String uId) {
        return hashMap.get(uId);
    }

}
```

cn.bugstack.springframework.test.bean.UserService

```
public class UserService {

    private String uId;

    private UserDao userDao;

    public void queryUserInfo() {
        System.out.println("查询用户信息：" + userDao.queryUserName(uId));
    }

    // ...get/set
}
```

- Dao、Service，是我们平常开发经常使用的场景。在 UserService 中注入 UserDao，这样就能体现出 Bean 属性的依赖了。

2. 测试用例

```
@Test
public void test_BeanFactory() {
    // 1.初始化 BeanFactory
    DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();

    // 2. UserDao 注册
    beanFactory.registerBeanDefinition("userDao", new BeanDefinition(UserDao.class)
);

    // 3. UserService 设置属性[uId、userDao]
    PropertyValues propertyValues = new PropertyValues();
    propertyValues.addPropertyValue(new PropertyValue("uId", "10001"));
    propertyValues.addPropertyValue(new PropertyValue("userDao",new BeanReference("
userDao")));

    // 4. UserService 注入 bean
    BeanDefinition beanDefinition = new BeanDefinition(UserService.class, propertyV
alues);
```

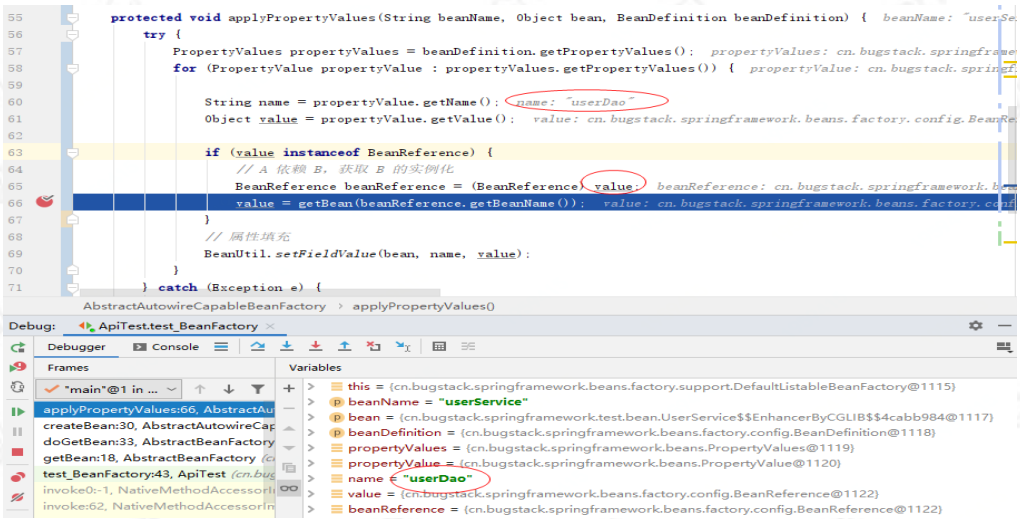
```
        beanFactory.registerBeanDefinition("userService", beanDefinition);

        // 5. UserService 获取bean
        UserService userService = (UserService) beanFactory.getBean("userService");
        userService.queryUserInfo();
    }
```

- 与直接获取 Bean 对象不同，这次我们还需要先把 userDao 注入到 Bean 容器中。`beanFactory.registerBeanDefinition("userDao", new BeanDefinition(UserDao.class));`
- 接下来就是属性填充的操作了，一种是普通属性 `new PropertyValue("uId", "10001")`，另外一种是对象属性 `new PropertyValue("userDao",new BeanReference("userDao"))`
- 接下来的操作就简单了，只不过是正常获取 userService 对象，调用方法即可。

3. 测试结果

```
查询用户信息：小傅哥

Process finished with exit code 0
```

- 从测试结果看我们的属性填充已经起作用了，因为只有属性填充后，才能调用到 Dao 方法，如：`userDao.queryUserName(uId)`
- 那么我们在看看 Debug 调试的情况下，有没有进入到实现的 Bean 属性填充中，如下：



  - 好，就是截图这里，我们看到已经开始进行属性填充操作了，当发现属性是 BeanReference 时，则需要获取创建 Bean 实例。

六、总结

- 在本章节中我们把 AbstractAutowireCapableBeanFactory 类中的创建对象功能又做了扩充，依赖于是否有构造函数的实例化策略完成后，开始补充 Bean 属性信息。当遇到 Bean 属性为 Bean 对象时，需要递归处理。最后在属性填充时需要用到反射操作，也可以使用一些工具类处理。
- 每一个章节的功能点我们都在循序渐进的实现，这样可以让新人更好的接受关于 Spring 中的设计思路。尤其是在一些已经开发好的类上，怎么扩充新的功能时候的设计更为重要。学习编程有的时候学习思路设计要比仅仅是做简单实现，更能提升编程思维。
- 到这一章节关于 Bean 的创建操作就开发完成了，接下来需要整个框架的基础上完成资源属性的加载，就是我们需要去动 Xml 配置了，让我们这小框架越来越像 Spring。另外在框架实现的过程中所有的类名都会参考 Spring 源码，以及相应的设计实现步骤也是与 Spring 源码中对应，只不过会简化一些流程，但你可以拿相同的类名，去搜到每一个功能在 Spring 源码中的实现。

第 06 章：资源加载器解析文件注册对象

一、扩展框架

你写的代码，能接的住产品加需求吗？

接，是能接的，接几次也行，哪怕就一个类一片的 if...else 也可以！但接完成什么样可就不一定了，会不会出事故也不是能控制住的。

那出事故时，你说因为我写 if...else 多了导致代码烂了，但可是你先动的手啊：你说的需求还得加、你说的老板让上线、你说的合同都签了，搬砖码农的我没办法，才以堆代码平需求，需求太多不好搞，我才以搬砖平需求！*诸侯不服，我才以兵服诸侯，你不服，我就打到你服！*

但代码烂了有时候并不是因为需求加的快、也不是着急上线。因为往往在承接产品需求的前几次，一个功能逻辑的设计并不会太复杂，也不会有多急迫，甚至会留出让你做设计、做评审、做开发的时间，如果这个时候仍不能把以后可能会发生的事情评估到需求里，那么导致代码的混乱从一开始就已经埋下了，以后只能越来越乱！

承接需求并能把它做好，这来自于对需求的理解，产品场景开发的经验以及对代码实践落地的把控能力等综合多方面因素的结果。就像你现在做的开发中，你的代码有哪些是经常变化的，有哪些是固定通用的，有哪些是负责逻辑拼装的、有哪些是来做核心实现的。那么现在如果你的核心共用层做了频繁变化的业务层包装，那么肯定的说，你的代码即将越来越乱，甚至可能埋下事故的风险！

在我们实现的 Spring 框架中，每一个章节都会结合上一章节继续扩展功能，就像每一次产品都在加需求一样，那么在学习的过程中可以承上启下的对照和参考，看看每一个模块的添加都是用什么逻辑和技术细节实现的。这些内容的学习，会非常有利于你以后在设计和实现，自己承接产品需求时做的具体开发，代码的质量也会越来越高，越来越有扩展性和可维护性。

二、目标

在完成 Spring 的框架雏形后，现在我们可以通过单元测试进行手动操作 Bean 对象的定义、注册和属性填充，以及最终获取对象调用方法。但这里会有一个问题，就是如果实际使用这个 Spring 框架，是不太可能让用户通过手动方式创建的，而是最好能通过配置文件的方式简化创建过程。需要完成如下操作：

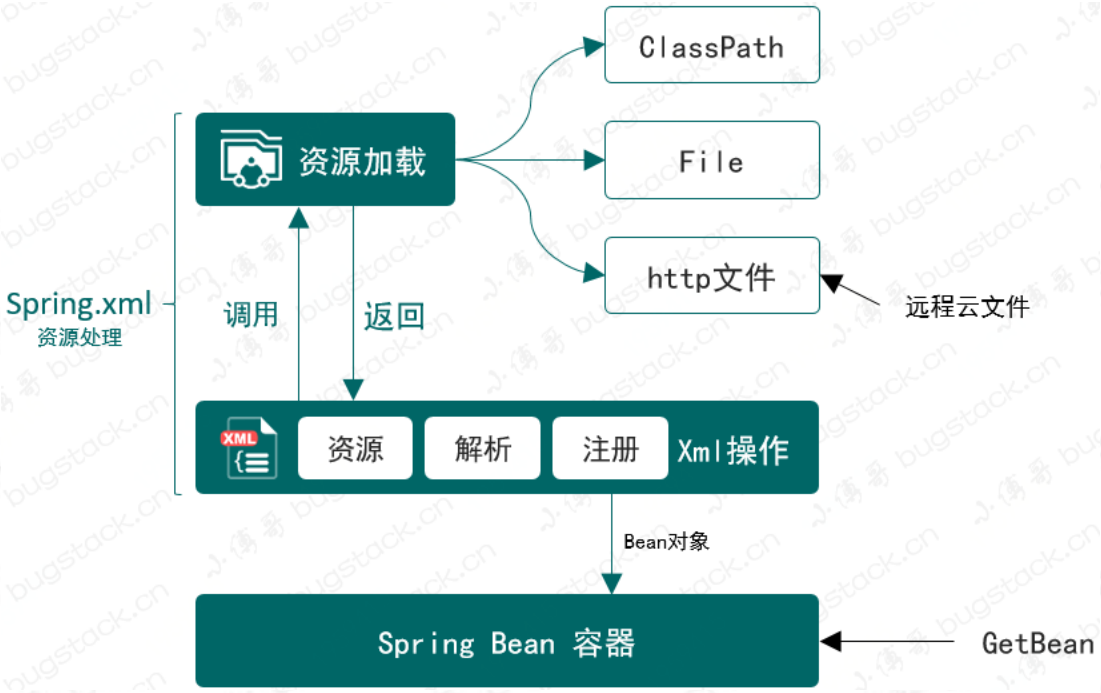
```
@Test
public void test_BeanFactory() {
    // 1.初始化 BeanFactory
    DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();

    // 2. UserDao 注册
    beanFactory.registerBeanDefinition( beanName: "userDao", new BeanDefinition(UserDao.class));

    // 3. UserService 设置属性[uId、userDao]
    PropertyValues propertyValues = new PropertyValues();
    propertyValues.addPropertyValue(new PropertyValue( name: "uId", value: "10001"));
    propertyValues.addPropertyValue(new PropertyValue( name: "userDao",new BeanReference( beanName: "userDao")));

    // 4. UserService 注入bean
    BeanDefinition beanDefinition = new BeanDefinition(UserService.class, propertyValues);
    beanFactory.registerBeanDefinition( beanName: "userService", beanDefinition);

    // 5. UserService 获取bean
    UserService userService = (UserService) beanFactory.getBean( name: "userService");
    userService.queryUserInfo();
}
```

把这一部分的操作，放到配置文件中进行处理。

- 如图中我们需要把步骤：2、3、4 整合到 Spring 框架中，通过 Spring 配置文件的方式将 Bean 对象实例化。
- 接下来我们就需要在现有的 Spring 框架中，添加能解决 Spring 配置的读取、解析、注册 Bean 的操作。

三、设计

依照本章节的需求背景，我们需要在现有的 Spring 框架雏形中添加一个资源解析器，也就是能读取 classpath、本地文件和云文件的配置内容。这些配置内容就是像使用 Spring 时配置的 Spring.xml 一样，里面会包括 Bean 对象的描述和属性信息。 在读取配置文件信息后，接下来就是对配置文件中的 Bean 描述信息解析后进行注册操作，把 Bean 对象注册到 Spring 容器中。整体设计结构如下图：



- 资源加载器属于相对独立的部分，它位于 Spring 框架核心包下的 IO 实现内容，主要用于处理 Class、本地和云环境中的文件信息。
- 当资源可以加载后，接下来就是解析和注册 Bean 到 Spring 中的操作，这部分实现需要和 DefaultListableBeanFactory 核心类结合起来，因为你所有的解析后的注册动作，都会把 Bean 定义信息放入到这个类中。
- 那么在实现的时候就设计好接口的实现层级关系，包括我们需要定义出 Bean 定义的读取接口 `BeanDefinitionReader` 以及做好对应的实现类，在实现类中完成对 Bean 对象的解析和注册。

四、实现

1. 工程结构

```
small-spring-step-05
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanReference.java
```

```
    │           │   │   ├── ConfigurableBeanFactory.java
    │           │   │   └── SingletonBeanRegistry.java
    │           │   ├── support
    │           │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   ├── AbstractBeanFactory.java
    │           │   │   ├── BeanDefinitionReader.java
    │           │   │   ├── BeanDefinitionRegistry.java
    │           │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   ├── DefaultListableBeanFactory.java
    │           │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   ├── InstantiationStrategy.java
    │           │   │   └── SimpleInstantiationStrategy.java
    │           │   ├── support
    │           │   │   └── XmlBeanDefinitionReader.java
    │           │   ├── BeanFactory.java
    │           │   ├── ConfigurableListableBeanFactory.java
    │           │   ├── HierarchicalBeanFactory.java
    │           │   └── ListableBeanFactory.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
    │           │   └── PropertyValues.java
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           └── utils
    │               └── ClassUtils.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── UserDao.java
                │   └── UserService.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

Spring Bean 容器资源加载和使用类关系，如图 6-3

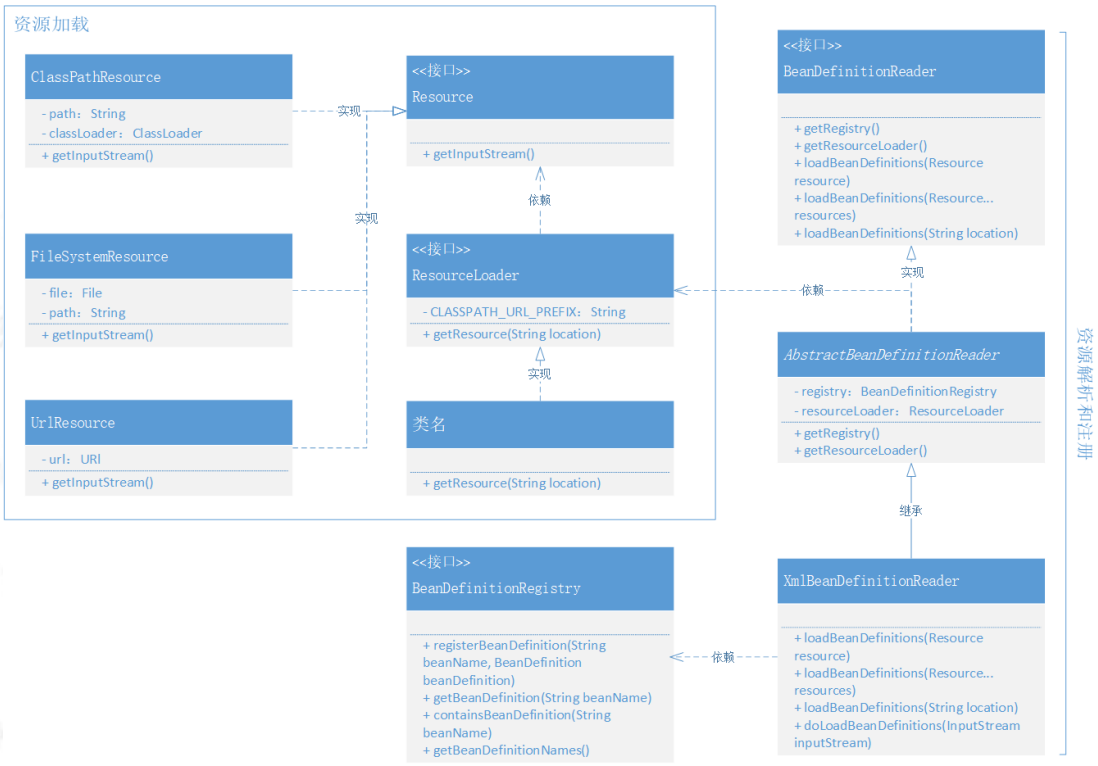


图 6-3

- 本章节为了能把 Bean 的定义、注册和初始化交给 Spring.xml 配置化处理，那么就需要实现两大块内容，分别是：资源加载器、xml 资源处理类，实现过程主要以对接口 `Resource`、`ResourceLoader` 的实现，而另外 `BeanDefinitionReader` 接口则是对资源的具体使用，将配置信息注册到 Spring 容器中去。
- 在 Resource 的资源加载器的实现中包括了，ClassPath、系统文件、云配置文件，这三部分与 Spring 源码中的设计和实现保持一致，最终在 DefaultResourceLoader 中做具体的调用。
- 接口：BeanDefinitionReader、抽象类：AbstractBeanDefinitionReader、实现类：XmlBeanDefinitionReader，这三部分内容主要是合理清晰的处理了资源读取后的注册 Bean 容器操作。*接口管定义，抽象类处理非接口功能外的注册 Bean 组件填充，最终实现类即可只关心具体的业务实现*

另外本章节还参考 Spring 源码，做了相应接口的集成和实现的关系，虽然这些接口目前还并没有太大的作用，但随着框架的逐步完善，它们也会发挥作用。如图 6-4

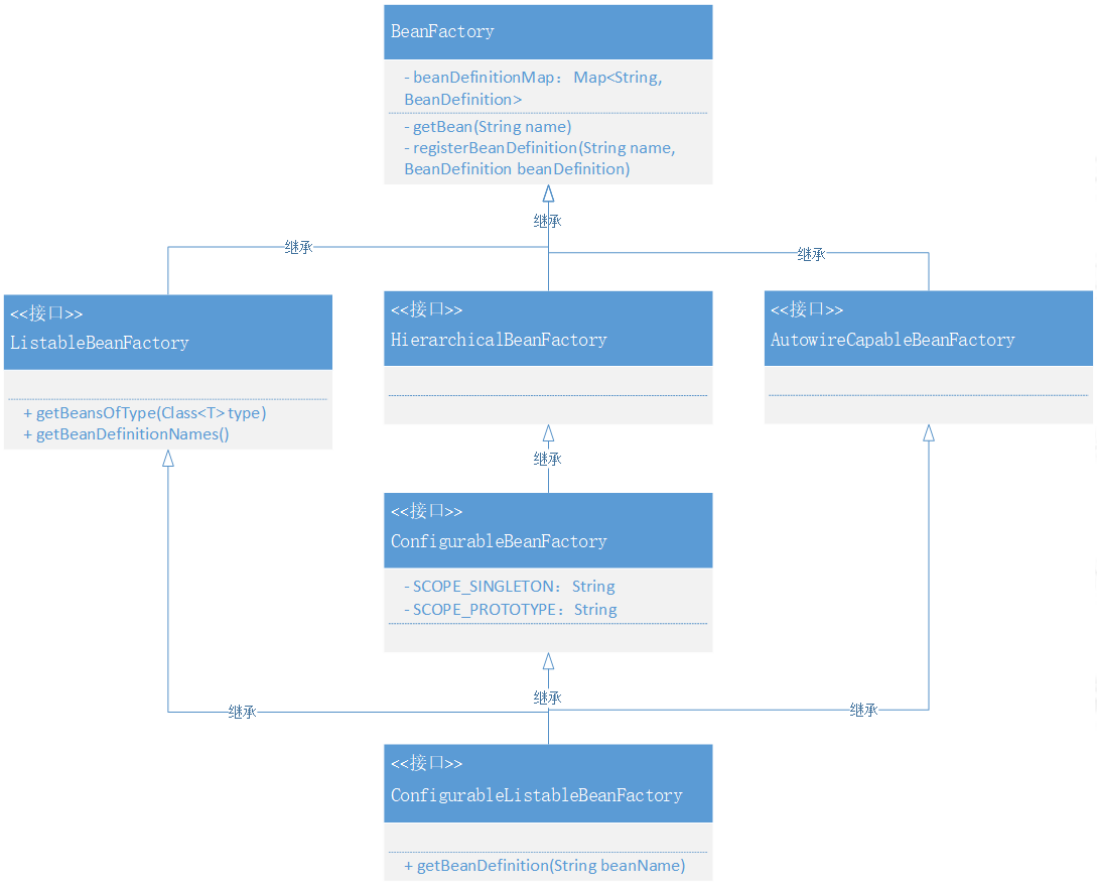


图 6-4

- BeanFactory，已经存在的 Bean 工厂接口用于获取 Bean 对象，这次新增加了按照类型获取 Bean 的方法：`<T> T getBean(String name, Class<T> requiredType)`
- ListableBeanFactory，是一个扩展 Bean 工厂接口的接口，新增加了 `getBeansOfType`、`getBeanDefinitionNames()` 方法，在 Spring 源码中还有其他扩展方法。
- HierarchicalBeanFactory，在 Spring 源码中它提供了可以获取父类 BeanFactory 方法，属于是一种扩展工厂的层次子接口。*Sub-interface implemented by bean factories that can be part of a hierarchy.*
- AutowireCapableBeanFactory，是一个自动化处理 Bean 工厂配置的接口，目前案例工程中还没有做相应的实现，后续逐步完善。
- ConfigurableBeanFactory，可获取 BeanPostProcessor、BeanClassLoader 等的一个配置化接口。
- ConfigurableListableBeanFactory，提供分析和修改 Bean 以及预先实例化的操作接口，不过目前只有一个 getBeanDefinition 方法。

2. 资源加载接口定义和实现

cn.bugstack.springframework.core.io.Resource

```
public interface Resource {

    InputStream getInputStream() throws IOException;

}
```

- 在 Spring 框架下创建 core.io 核心包，在这个包中主要用于处理资源加载流。
- 定义 Resource 接口，提供获取 InputStream 流的方法，接下来再分别实现三种不同的流文件操作：classPath、FileSystem、URL

ClassPath： cn.bugstack.springframework.core.io.ClassPathResource

```
public class ClassPathResource implements Resource {

    private final String path;

    private ClassLoader classLoader;

    public ClassPathResource(String path) {
        this(path, (ClassLoader) null);
    }

    public ClassPathResource(String path, ClassLoader classLoader) {
        Assert.notNull(path, "Path must not be null");
        this.path = path;
        this.classLoader = (classLoader != null ? classLoader : ClassUtils.getDefau
ltClassLoader());
    }

    @Override
    public InputStream getInputStream() throws IOException {
        InputStream is = classLoader.getResourceAsStream(path);
        if (is == null) {
            throw new FileNotFoundException(
                    this.path + " cannot be opened because it does not exist");
        }
        return is;
    }
}
```

- 这一部分的实现是用于通过 `ClassLoader` 读取 `ClassPath` 下的文件信息，具体的读取过程主要是：`classLoader.getResourceAsStream(path)`

FileSystem： cn.bugstack.springframework.core.io.FileSystemResource

```
public class FileSystemResource implements Resource {

    private final File file;

    private final String path;

    public FileSystemResource(File file) {
        this.file = file;
        this.path = file.getPath();
    }

    public FileSystemResource(String path) {
        this.file = new File(path);
        this.path = path;
    }

    @Override
    public InputStream getInputStream() throws IOException {
        return new FileInputStream(this.file);
    }

    public final String getPath() {
        return this.path;
    }

}
```

- 通过指定文件路径的方式读取文件信息，这部分大家肯定还是非常熟悉的，经常会读取一些 txt、excel 文件输出到控制台。

Url：cn.bugstack.springframework.core.io.UrlResource

```
public class UrlResource implements Resource{

    private final URL url;

    public UrlResource(URL url) {
        Assert.notNull(url,"URL must not be null");
        this.url = url;
    }

    @Override
    public InputStream getInputStream() throws IOException {
        URLConnection con = this.url.openConnection();
        try {
```

```
            return con.getInputStream();
        }
        catch (IOException ex){
            if (con instanceof HttpURLConnection){
                ((HttpURLConnection) con).disconnect();
            }
            throw ex;
        }
    }

}
```

- 通过 HTTP 的方式读取云服务的文件，我们也可以把配置文件放到 GitHub 或者 Gitee 上。

3. 包装资源加载器

按照资源加载的不同方式，资源加载器可以把这些方式集中到统一的类服务下进行处理，外部用户只需要传递资源地址即可，简化使用。

定义接口：cn.bugstack.springframework.core.io.ResourceLoader

```
public interface ResourceLoader {

    /**
     * Pseudo URL prefix for loading from the class path: "classpath:"
     */
    String CLASSPATH_URL_PREFIX = "classpath:";

    Resource getResource(String location);

}
```

- 定义获取资源接口，里面传递 location 地址即可。

实现接口：cn.bugstack.springframework.core.io.DefaultResourceLoader

```
public class DefaultResourceLoader implements ResourceLoader {

    @Override
    public Resource getResource(String location) {
        Assert.notNull(location, "Location must not be null");
        if (location.startsWith(CLASSPATH_URL_PREFIX)) {
```

```
            return new ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.le
ngth()));
        }
        else {
            try {
                URL url = new URL(location);
                return new UrlResource(url);
            } catch (MalformedURLException e) {
                return new FileSystemResource(location);
            }
        }
    }

}
```

- 在获取资源的实现中，主要是把三种不同类型的资源处理方式进行了包装，分为：判断是否为 ClassPath、URL 以及文件。
- 虽然 DefaultResourceLoader 类实现的过程简单，但这也是设计模式约定的具体结果，像是这里不会让外部调用放知道过多的细节，而是仅关心具体调用结果即可。

4. Bean 定义读取接口

cn.bugstack.springframework.beans.factory.support.BeanDefinitionReader

```
public interface BeanDefinitionReader {

    BeanDefinitionRegistry getRegistry();

    ResourceLoader getResourceLoader();

    void loadBeanDefinitions(Resource resource) throws BeansException;

    void loadBeanDefinitions(Resource... resources) throws BeansException;

    void loadBeanDefinitions(String location) throws BeansException;

}
```

- 这是一个 *Simple interface for bean definition readers.* 其实里面无非定义了几个方法，包括：getRegistry()、getResourceLoader()，以及三个加载 Bean 定义的方法。

- 这里需要注意 getRegistry()、getResourceLoader()，都是用于提供给后面三个方法的工具，加载和注册，这两个方法的实现会包装到抽象类中，以免污染具体的接口实现方法。

5. Bean 定义抽象类实现

cn.bugstack.springframework.beans.factory.support.AbstractBeanDefinitionReader

```
public abstract class AbstractBeanDefinitionReader implements BeanDefinitionReader
{

    private final BeanDefinitionRegistry registry;

    private ResourceLoader resourceLoader;

    protected AbstractBeanDefinitionReader(BeanDefinitionRegistry registry) {
        this(registry, new DefaultResourceLoader());
    }

    public AbstractBeanDefinitionReader(BeanDefinitionRegistry registry, ResourceLo
ader resourceLoader) {
        this.registry = registry;
        this.resourceLoader = resourceLoader;
    }

    @Override
    public BeanDefinitionRegistry getRegistry() {
        return registry;
    }

    @Override
    public ResourceLoader getResourceLoader() {
        return resourceLoader;
    }

}
```

- 抽象类把 BeanDefinitionReader 接口的前两个方法全部实现完了，并提供了构造函数，让外部的调用使用方，把 Bean 定义注入类，传递进来。
- 这样在接口 BeanDefinitionReader 的具体实现类中，就可以把解析后的 XML 文件中的 Bean 信息，注册到 Spring 容器去了。*以前我们是通过单元测试使用，调用 BeanDefinitionRegistry 完成 Bean 的注册，现在可以放到 XML 中操作了*

6. 解析 XML 处理 Bean 注册

cn.bugstack.springframework.beans.factory.xml.XmlBeanDefinitionReader

```
public class XmlBeanDefinitionReader extends AbstractBeanDefinitionReader {

    public XmlBeanDefinitionReader(BeanDefinitionRegistry registry) {
        super(registry);
    }

    public XmlBeanDefinitionReader(BeanDefinitionRegistry registry, ResourceLoader resourceLoader) {
        super(registry, resourceLoader);
    }

    @Override
    public void loadBeanDefinitions(Resource resource) throws BeansException {
        try {
            try (InputStream inputStream = resource.getInputStream()) {
                doLoadBeanDefinitions(inputStream);
            }
        } catch (IOException | ClassNotFoundException e) {
            throw new BeansException("IOException parsing XML document from " + resource, e);
        }
    }

    @Override
    public void loadBeanDefinitions(Resource... resources) throws BeansException {
        for (Resource resource : resources) {
            loadBeanDefinitions(resource);
        }
    }

    @Override
    public void loadBeanDefinitions(String location) throws BeansException {
        ResourceLoader resourceLoader = getResourceLoader();
        Resource resource = resourceLoader.getResource(location);
        loadBeanDefinitions(resource);
    }

    protected void doLoadBeanDefinitions(InputStream inputStream) throws ClassNotFoundException {
```

```
        Document doc = XmlUtil.readXML(inputStream);
        Element root = doc.getDocumentElement();
        NodeList childNodes = root.getChildNodes();

        for (int i = 0; i < childNodes.getLength(); i++) {
            // 判断元素
            if (!(childNodes.item(i) instanceof Element)) continue;
            // 判断对象
            if (!"bean".equals(childNodes.item(i).getNodeName())) continue;

            // 解析标签
            Element bean = (Element) childNodes.item(i);
            String id = bean.getAttribute("id");
            String name = bean.getAttribute("name");
            String className = bean.getAttribute("class");
            // 获取 Class，方便获取类中的名称
            Class<?> clazz = Class.forName(className);
            // 优先级 id > name
            String beanName = StrUtil.isNotEmpty(id) ? id : name;
            if (StrUtil.isEmpty(beanName)) {
                beanName = StrUtil.lowerFirst(clazz.getSimpleName());
            }

            // 定义Bean
            BeanDefinition beanDefinition = new BeanDefinition(clazz);
            // 读取属性并填充
            for (int j = 0; j < bean.getChildNodes().getLength(); j++) {
                if (!(bean.getChildNodes().item(j) instanceof Element)) continue;
                if (!"property".equals(bean.getChildNodes().item(j).getNodeName()))
 continue;
                // 解析标签：property
                Element property = (Element) bean.getChildNodes().item(j);
                String attrName = property.getAttribute("name");
                String attrValue = property.getAttribute("value");
                String attrRef = property.getAttribute("ref");
                // 获取属性值：引入对象、值对象
                Object value = StrUtil.isNotEmpty(attrRef) ? new BeanReference(attr
Ref) : attrValue;
                // 创建属性信息
                PropertyValue propertyValue = new PropertyValue(attrName, value);
                beanDefinition.getPropertyValues().addPropertyValue(propertyValue);
            }
            if (getRegistry().containsBeanDefinition(beanName)) {
                throw new BeansException("Duplicate beanName[" + beanName + "] is n
```

```
ot allowed");
                }
                // 注册 BeanDefinition
                getRegistry().registerBeanDefinition(beanName, beanDefinition);
            }
        }

}
```

XmlBeanDefinitionReader 类最核心的内容就是对 XML 文件的解析，把我们本来在代码中的操作放到了通过解析 XML 自动注册的方式。

- loadBeanDefinitions 方法，处理资源加载，这里新增加了一个内部方法：`doLoadBeanDefinitions`，它主要负责解析 xml
- 在 doLoadBeanDefinitions 方法中，主要是对 xml 的读取 `XmlUtil.readXML(inputStream)` 和元素 Element 解析。在解析的过程中通过循环操作，以此获取 Bean 配置以及配置中的 id、name、class、value、ref 信息。
- 最终把读取出来的配置信息，创建成 BeanDefinition 以及 PropertyValue，最终把完整的 Bean 定义内容注册到 Bean 容器：`getRegistry().registerBeanDefinition(beanName, beanDefinition)`

五、测试

1. 事先准备

cn.bugstack.springframework.test.bean.UserDao

```
public class UserDao {

    private static Map<String, String> hashMap = new HashMap<>();

    static {
        hashMap.put("10001", "小傅哥");
        hashMap.put("10002", "八杯水");
        hashMap.put("10003", "阿毛");
    }

    public String queryUserName(String uId) {
        return hashMap.get(uId);
    }
```

```
}
```

cn.bugstack.springframework.test.bean.UserService

```
public class UserService {

    private String uId;

    private UserDao userDao;

    public void queryUserInfo() {
        return userDao.queryUserName(uId);
    }

    // ...get/set
}
```

- Dao、Service，是我们平常开发经常使用的场景。在 UserService 中注入 UserDao，这样就能体现出 Bean 属性的依赖了。

2. 配置文件

important.properties

```
# Config File
system.key=OLpj9823dZ
```

spring.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>

    <bean id="userDao" class="cn.bugstack.springframework.test.bean.UserDao"/>

    <bean id="userService" class="cn.bugstack.springframework.test.bean.UserService
">
        <property name="uId" value="10001"/>
        <property name="userDao" ref="userDao"/>
    </bean>

</beans>
```

- 这里有两份配置文件，一份用于测试资源加载器，另外 spring.xml 用于测试整体的 Bean 注册功能。

3. 单元测试(资源加载)

案例

```
private DefaultResourceLoader resourceLoader;

@Before
public void init() {
    resourceLoader = new DefaultResourceLoader();
}

@Test
public void test_classpath() throws IOException {
    Resource resource = resourceLoader.getResource("classpath:important.properties"
);
    InputStream inputStream = resource.getInputStream();
    String content = IoUtil.readUtf8(inputStream);
    System.out.println(content);
}

@Test
public void test_file() throws IOException {
    Resource resource = resourceLoader.getResource("src/test/resources/important.pr
operties");
    InputStream inputStream = resource.getInputStream();
    String content = IoUtil.readUtf8(inputStream);
    System.out.println(content);
}

@Test
public void test_url() throws IOException {
    Resource resource = resourceLoader.getResource("https://github.com/fuzhengwei/s
mall-spring/important.properties"
    InputStream inputStream = resource.getInputStream();
    String content = IoUtil.readUtf8(inputStream);
    System.out.println(content);
}
```

测试结果

```
# Config File
system.key=OLpj9823dZ

Process finished with exit code 0
```

- 这三个方法：test_classpath、test_file、test_url，分别用于测试加载 ClassPath、FileSystem、Url 文件，*URL 文件在 Github，可能加载时会慢*

4. 单元测试(配置文件注册 Bean)

案例

```
@Test
public void test_xml() {
    // 1.初始化 BeanFactory
    DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();

    // 2. 读取配置文件&注册 Bean
    XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(beanFactory);
    reader.loadBeanDefinitions("classpath:spring.xml");

    // 3. 获取 Bean 对象调用方法
    UserService userService = beanFactory.getBean("userService", UserService.class)
;
    String result = userService.queryUserInfo();
    System.out.println("测试结果：" + result);
}
```

测试结果

```
测试结果：小傅哥

Process finished with exit code 0
```

- 在上面的测试案例中可以看到，我们把以前通过手动注册 Bean 以及配置属性信息的内容，交给了 `new XmlBeanDefinitionReader(beanFactory)` 类读取 Spring.xml 的方式来处理，并通过了测试验证。

六、总结

- 此时的工程结构已经越来越有 Spring 框架的味道了，以配置文件为入口解析和注册 Bean 信息，最终再通过 Bean 工厂获取 Bean 以及做相应的调用操作。
- 关于案例中每一个步骤的实现小傅哥这里都会尽可能参照 Spring 源码的接口定义、抽象类实现、名称规范、代码结构等，做相应的简化处理。这样大家在学习的过程中也可以通过类名或者接口和整个结构体学习 Spring 源码，这样学习起来就容易多了。
- 看完绝对不等于会，你只有动起手来从一个小小的工程框架结构，敲到现在以及以后不断的变大、变多、变强时，才能真的掌握这里面的知识。另外每一个章节的功

能实现都会涉及到很多的代码设计思路，要认真去领悟。*当然实践起来是最好的领悟方式！*

第 07 章：应用上下文

一、不能写死

你这代码，可不能写死了呀！

依照项目落地经验来看，我们在承接紧急的产品需求时候，通常会选择在原有同类项目中进行扩展，如果没有相关类型项目的储备，也可能会选择临时搭建出一个工程来实现产品的需求。但这个时候就会遇到非常现实的问题，选择完整的设计和开发就可能满足不了上线时间，临时拼凑式的完成需求又可能不具备上线后响应产品的临时调整。

上线后的调整有哪些呢？项目刚一上线，运营了还不到半天，老板发现自己的配置的活动好像金额配置的太小了，用户都不来，割不到韭菜呀。赶紧半夜联系产品，来来来，你给我这改改，那修修，把人均优惠 1 万元放大大的，把可能两字缩小放在后面。再把优惠的奖金池配置从 10 元调整 11 元，快快快，赶紧修改，你修改了咱们能赚 1 个亿！！！

好家伙，项目是临时开发堆出来的，没有后台系统、没有配置中心、没有模块拆分，老板一句句改改改，产品来传达催促，最后背锅的可就是研发了。*你这不能写死，这优惠配置得抽出来，这文案也后台下发吧，这接口入参也写死了，再写一个新接口吧！* 一顿操作猛如虎，研发搬砖修接口，运营折腾好几宿，最后 PV150！

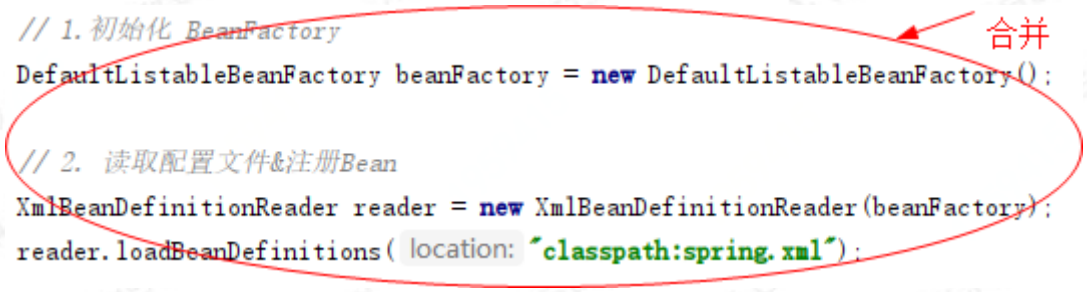
无论业务、产品、运营如何，但就研发自身来讲，尽可能的要不避免临时堆出一个服务来，尤其是在团队建设初期或者运营思路经常调整的情况下，更要注重设计细节和实现方案。哪怕去报风险延期，也不要让自己背上一个明知是烂坑还要接的活。

而本章节说到不把代码写死，就是因为我们需要继续在手写 Spring 框架中继续扩展新的功能，如一个 Bean 的定义和实例化的过程前后，是否可以满足我们进行自定义扩展，对 Bean 对象执行一些修改、增强、记录等操作呢？ *这个过程基本就是你在使用 Spring 容器框架时候做的一些中间件扩展开发。*

二、目标

如果你在自己的实际工作中开发过基于 Spring 的技术组件，或者学习过关于 SpringBoot 中间件设计和开发 等内容。那么你一定会继承或者实现了 Spring 对外暴露的类或接口，在接口的实现中获取了 BeanFactory 以及 Bean 对象的获取等内容，并对这些内容做一些操作，例如：修改 Bean 的信息，添加日志打印、处理数据库路由对数据源的切换、给 RPC 服务连接注册中心等。

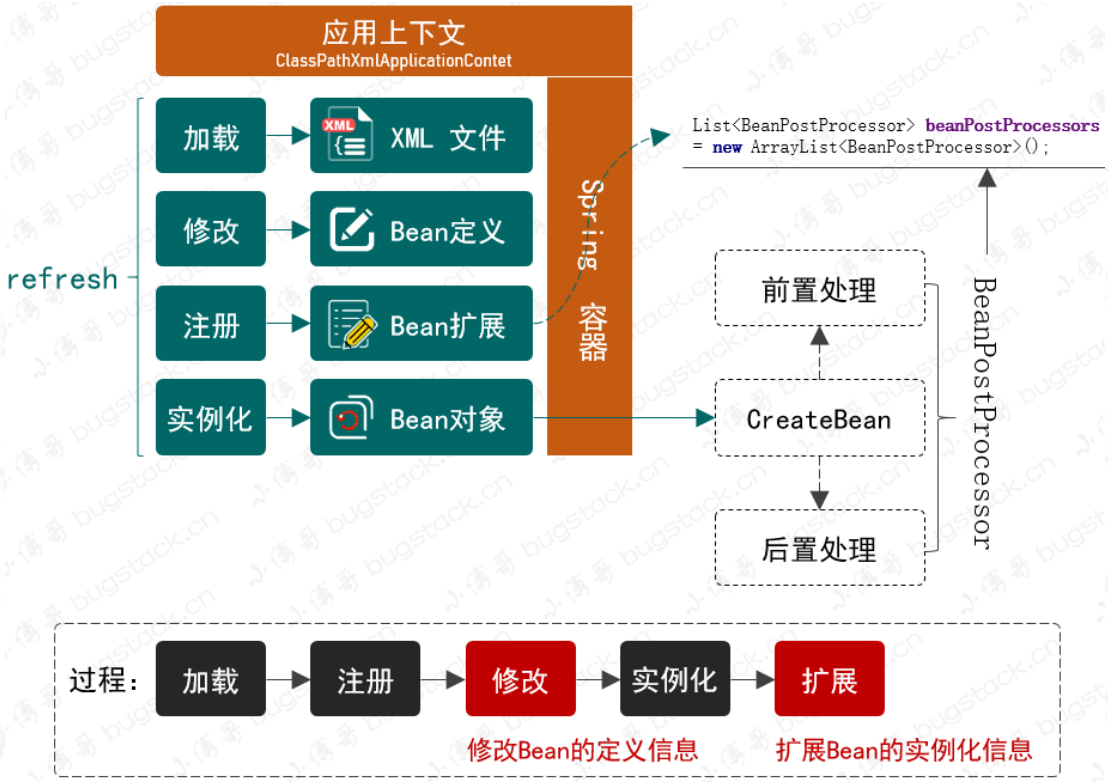
在对容器中 Bean 的实例化过程添加扩展机制的同时，还需要把目前关于 Spring.xml 初始化和加载策略进行优化，因为我们不太可能让面向 Spring 本身开发的 `DefaultListableBeanFactory` 服务，直接给予用户使用。修改点如下：



- DefaultListableBeanFactory、XmlBeanDefinitionReader，是我们在目前 Spring 框架中对于服务功能测试的使用方式，它能很好的体现出 Spring 是如何对 xml 加载以及注册 Bean 对象的操作过程，但这种方式是面向 Spring 本身的，还不具备一定的扩展性。
- 就像我们现在需要提供出一个可以在 Bean 初始化过程中，完成对 Bean 对象的扩展时，就很难做到自动化处理。所以我们要把 Bean 对象扩展机制功能和对 Spring 框架上下文的包装融合起来，对外提供完整的服务。

三、设计

为了能满足于在 Bean 对象从注册到实例化的过程中执行用户的自定义操作，就需要在 Bean 的定义和初始化过程中插入接口类，这个接口再有外部去实现自己需要的服务。那么在结合对 Spring 框架上下文的处理能力，就可以满足我们的目标需求了。整体设计结构如下图：



- 满足于对 Bean 对象扩展的两个接口，其实也是 Spring 框架中非常具有重量级的两个接口：`BeanFactoryPostProcess` 和 `BeanPostProcessor`，也几乎是大家在使用 Spring 框架额外新增开发自己组建需求的两个必备接口。
- BeanFactoryPostProcessor，是由 Spring 框架组建提供的容器扩展机制，允许在 Bean 对象注册后但未实例化之前，对 Bean 的定义信息 `BeanDefinition` 执行修改操作。
- BeanPostProcessor，也是 Spring 提供的扩展机制，不过 BeanPostProcessor 是在 Bean 对象实例化之后修改 Bean 对象，也可以替换 Bean 对象。这部分与后面要实现的 AOP 有着密切的关系。
- 同时如果只是添加这两个接口，不做任何包装，那么对于使用者来说还是非常麻烦的。我们希望于开发 Spring 的上下文操作类，把相应的 XML 加载 、注册、实例化以及新增的修改和扩展都融合进去，让 Spring 可以自动扫描到我们的新增服务，便于用户使用。

四、实现

1. 工程结构

```
small-spring-step-06
└── src
    ├── main
    │   └── java
```

```
    │       └── cn.bugstack.springframework
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   │   ├── BeanPostProcessor.java
    │           │   │   │   ├── BeanReference.java
    │           │   │   │   ├── ConfigurableBeanFactory.java
    │           │   │   │   └── SingletonBeanRegistry.java
    │           │   │   ├── support
    │           │   │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   │   ├── AbstractBeanFactory.java
    │           │   │   │   ├── BeanDefinitionReader.java
    │           │   │   │   ├── BeanDefinitionRegistry.java
    │           │   │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   │   ├── DefaultListableBeanFactory.java
    │           │   │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   │   ├── InstantiationStrategy.java
    │           │   │   │   └── SimpleInstantiationStrategy.java
    │           │   │   ├── support
    │           │   │   │   └── XmlBeanDefinitionReader.java
    │           │   │   ├── BeanFactory.java
    │           │   │   ├── ConfigurableListableBeanFactory.java
    │           │   │   ├── HierarchicalBeanFactory.java
    │           │   │   └── ListableBeanFactory.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
    │           │   └── PropertyValues.java
    │           ├── context
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   └── ClassPathXmlApplicationContext.java
    │           │   ├── ApplicationContext.java
    │           │   └── ConfigurableApplicationContext.java
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
```

```
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           └── utils
    │               └── ClassUtils.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── UserDao.java
                │   └── UserService.java
                ├── common
                │   ├── MyBeanFactoryPostProcessor.java
                │   └── MyBeanPostProcessor.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

Spring 应用上下文和对 Bean 对象扩展机制的类关系，如图 7-3

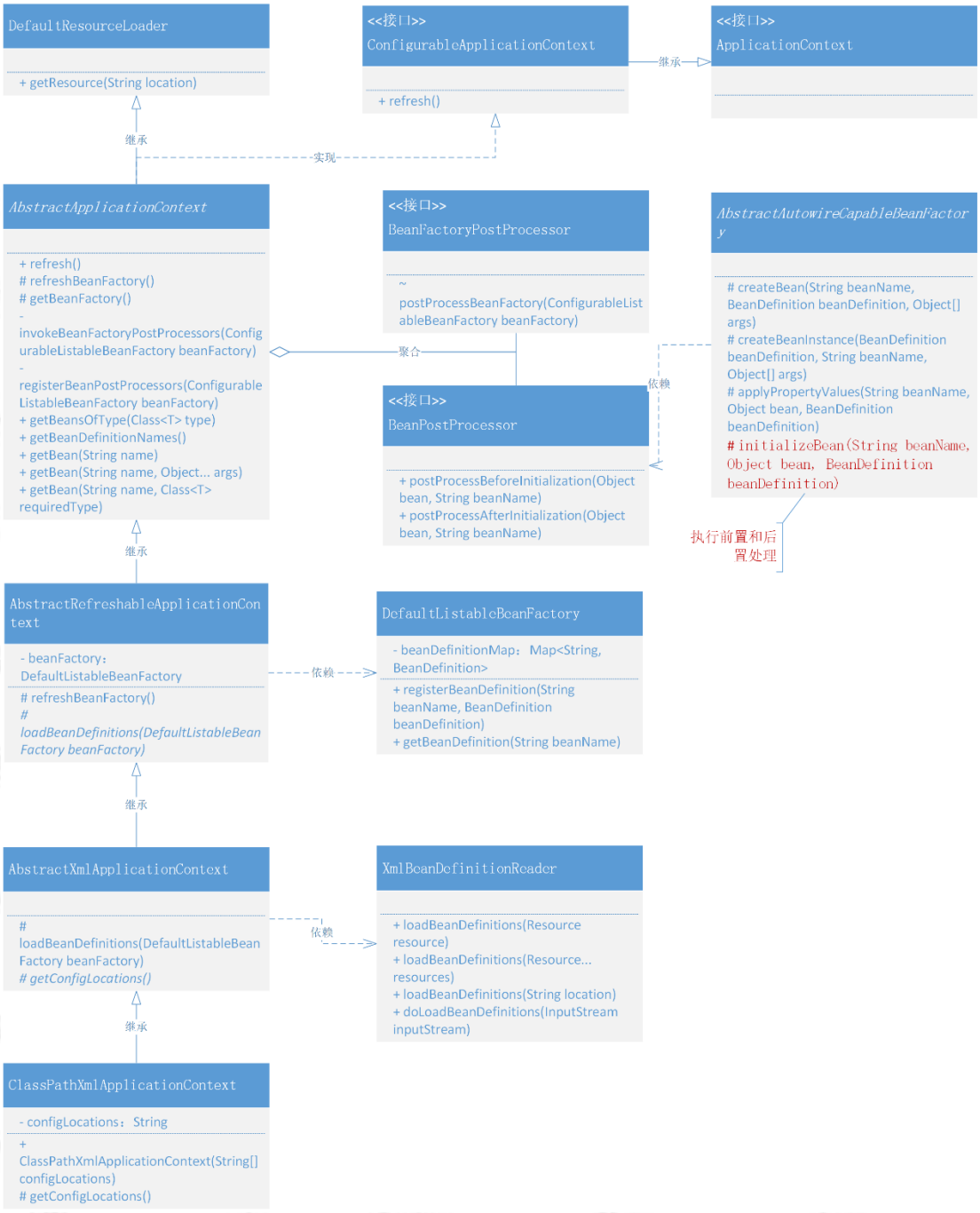


图 7-3

- 在整个类图中主要体现出来的是关于 Spring 应用上下文以及对 Bean 对象扩展机制的实现。
- 以继承了 ListableBeanFactory 接口的 ApplicationContext 接口开始，扩展出一系列应用上下文的抽象实现类，并最终完成 `ClassPathXmlApplicationContext` 类的实现。而这个类就是最后交给用户使用的类。
- 同时在实现应用上下文的过程中，通过定义接口：`BeanFactoryPostProcessor`、`BeanPostProcessor` 两个接口，把关于对 Bean 的扩展机制串联进去了。

2. 定义 BeanFactoryPostProcessor

cn.bugstack.springframework.beans.factory.config.BeanFactoryPostProcessor

```
public interface BeanFactoryPostProcessor {

    /**
     * 在所有的 BeanDefinition 加载完成后，实例化 Bean 对象之前，提供修改 BeanDefinition 属性的机制
     *
     * @param beanFactory
     * @throws BeansException
     */
    void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;

}
```

- 在 Spring 源码中有这样一段描述 `Allows for custom modification of an application context's bean definitions,adapting the bean property values of the context's underlying bean factory.` 其实也就是说这个接口是满足于在所有的 BeanDefinition 加载完成后，实例化 Bean 对象之前，提供修改 BeanDefinition 属性的机制。

3. 定义 BeanPostProcessor

cn.bugstack.springframework.beans.factory.config.BeanPostProcessor

```
public interface BeanPostProcessor {

    /**
     * 在 Bean 对象执行初始化方法之前，执行此方法
     *
     * @param bean
     * @param beanName
     * @return
     * @throws BeansException
     */
    Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException;

    /**
```

```
     * 在 Bean 对象执行初始化方法之后，执行此方法
     *
     * @param bean
     * @param beanName
     * @return
     * @throws BeansException
     */
    Object postProcessAfterInitialization(Object bean, String beanName) throws Bean
sException;

}
```

- 在 Spring 源码中有这样一段描述 `Factory hook that allows for custom modification of new bean instances,e.g. checking for marker interfaces or wrapping them with proxies.`也就是提供了修改新实例化 Bean 对象的扩展点。
- 另外此接口提供了两个方法：`postProcessBeforeInitialization` 用于在 Bean 对象执行初始化方法之前，执行此方法、`postProcessAfterInitialization` 用于在 Bean 对象执行初始化方法之后，执行此方法。

4. 定义上下文接口

cn.bugstack.springframework.context.ApplicationContext

```
public interface ApplicationContext extends ListableBeanFactory {
}
```

- context 是本次实现应用上下文功能新增的服务包
- ApplicationContext，继承于 ListableBeanFactory，也就继承了关于 BeanFactory 方法，比如一些 getBean 的方法。另外 ApplicationContext 本身是 Central 接口，但目前还不需要添加一些获取 ID 和父类上下文，所以暂时没有接口方法的定义。

cn.bugstack.springframework.context.ConfigurableApplicationContext

```
public interface ConfigurableApplicationContext extends ApplicationContext {

    /**
     * 刷新容器
     *
     * @throws BeansException
     */
    void refresh() throws BeansException;
```

```
}
```

- ConfigurableApplicationContext 继承自 ApplicationContext，并提供了 refresh 这个核心方法。*如果你有看过一些 Spring 源码，那么一定会看到这个方法。* 接下来也是需要在上下文的实现中完成刷新容器的操作过程。

5. 应用上下文抽象类实现

cn.bugstack.springframework.context.support.AbstractApplicationContext

```
public abstract class AbstractApplicationContext extends DefaultResourceLoader implements ConfigurableApplicationContext {

    @Override
    public void refresh() throws BeansException {
        // 1. 创建 BeanFactory，并加载 BeanDefinition
        refreshBeanFactory();

        // 2. 获取 BeanFactory
        ConfigurableListableBeanFactory beanFactory = getBeanFactory();

        // 3. 在 Bean 实例化之前，执行 BeanFactoryPostProcessor (Invoke factory processors registered as beans in the context.)
        invokeBeanFactoryPostProcessors(beanFactory);

        // 4. BeanPostProcessor 需要提前于其他 Bean 对象实例化之前执行注册操作
        registerBeanPostProcessors(beanFactory);

        // 5. 提前实例化单例Bean对象
        beanFactory.preInstantiateSingletons();
    }

    protected abstract void refreshBeanFactory() throws BeansException;

    protected abstract ConfigurableListableBeanFactory getBeanFactory();

    private void invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory) {
        Map<String, BeanFactoryPostProcessor> beanFactoryPostProcessorMap = beanFactory.getBeansOfType(BeanFactoryPostProcessor.class);
```

```
        for (BeanFactoryPostProcessor beanFactoryPostProcessor : beanFactoryPostPro
cessorMap.values()) {
            beanFactoryPostProcessor.postProcessBeanFactory(beanFactory);
        }
    }

    private void registerBeanPostProcessors(ConfigurableListableBeanFactory beanFac
tory) {
        Map<String, BeanPostProcessor> beanPostProcessorMap = beanFactory.getBeansO
fType(BeanPostProcessor.class);
        for (BeanPostProcessor beanPostProcessor : beanPostProcessorMap.values()) {
            beanFactory.addBeanPostProcessor(beanPostProcessor);
        }
    }

    //... getBean、getBeansOfType、getBeanDefinitionNames 方法

}
```

- AbstractApplicationContext 继承 DefaultResourceLoader 是为了处理 `spring.xml` 配置资源的加载。
- 之后是在 refresh() 定义实现过程，包括：
    1. 创建 BeanFactory，并加载 BeanDefinition
    2. 获取 BeanFactory
    3. 在 Bean 实例化之前，执行 BeanFactoryPostProcessor (Invoke factory processors registered as beans in the context.)
    4. BeanPostProcessor 需要提前于其他 Bean 对象实例化之前执行注册操作
    5. 提前实例化单例 Bean 对象
- 另外把定义出来的抽象方法，refreshBeanFactory()、getBeanFactory() 由后面的继承此抽象类的其他抽象类实现。

6. 获取 Bean 工厂和加载资源

cn.bugstack.springframework.context.support.AbstractRefreshableApplicationContext

```
public abstract class AbstractRefreshableApplicationContext extends AbstractApplicationContext {

    private DefaultListableBeanFactory beanFactory;

    @Override
    protected void refreshBeanFactory() throws BeansException {
        DefaultListableBeanFactory beanFactory = createBeanFactory();
        loadBeanDefinitions(beanFactory);
        this.beanFactory = beanFactory;
    }

    private DefaultListableBeanFactory createBeanFactory() {
        return new DefaultListableBeanFactory();
    }

    protected abstract void loadBeanDefinitions(DefaultListableBeanFactory beanFactory);

    @Override
    protected ConfigurableListableBeanFactory getBeanFactory() {
        return beanFactory;
    }

}
```

- 在 refreshBeanFactory() 中主要是获取了 `DefaultListableBeanFactory` 的实例化以及对资源配置的加载操作 `loadBeanDefinitions(beanFactory)`，在加载完成后即可完成对 spring.xml 配置文件中 Bean 对象的定义和注册，同时也包括实现了接口 BeanFactoryPostProcessor、BeanPostProcessor 的配置 Bean 信息。
- 但此时资源加载还只是定义了一个抽象类方法 `loadBeanDefinitions(DefaultListableBeanFactory beanFactory)`，继续由其他抽象类继承实现。

7. 上下文中对配置信息的加载

cn.bugstack.springframework.context.support.AbstractXmlApplicationContext

```
public abstract class AbstractXmlApplicationContext extends AbstractRefreshableApplicationContext {

    @Override
    protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) {
        XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory, this);
        String[] configLocations = getConfigLocations();
        if (null != configLocations){
            beanDefinitionReader.loadBeanDefinitions(configLocations);
        }
    }

    protected abstract String[] getConfigLocations();

}
```

- 在 AbstractXmlApplicationContext 抽象类的 loadBeanDefinitions 方法实现中，使用 XmlBeanDefinitionReader 类，处理了关于 XML 文件配置信息的操作。
- 同时这里又留下了一个抽象类方法，getConfigLocations()，此方法是为了从入口上下文类，拿到配置信息的地址描述。

8. 应用上下文实现类(ClassPathXmlApplicationContext)

cn.bugstack.springframework.context.support.ClassPathXmlApplicationContext

```
public class ClassPathXmlApplicationContext extends AbstractXmlApplicationContext {

    private String[] configLocations;

    public ClassPathXmlApplicationContext() {
    }

    /**
     * 从 XML 中加载 BeanDefinition，并刷新上下文
     *
     * @param configLocations
```

```
     * @throws BeansException
     */
    public ClassPathXmlApplicationContext(String configLocations) throws BeansExcep
tion {
        this(new String[]{configLocations});
    }

    /**
     * 从 XML 中加载 BeanDefinition，并刷新上下文
     * @param configLocations
     * @throws BeansException
     */
    public ClassPathXmlApplicationContext(String[] configLocations) throws BeansExc
eption {
        this.configLocations = configLocations;
        refresh();
    }

    @Override
    protected String[] getConfigLocations() {
        return configLocations;
    }

}
```

- ClassPathXmlApplicationContext，是具体对外给用户提供的应用上下文方法。
- 在继承了 AbstractXmlApplicationContext 以及层层抽象类的功能分离实现后，在此类 ClassPathXmlApplicationContext 的实现中就简单多了，主要是对继承抽象类中方法的调用和提供了配置文件地址信息。

9. 在 Bean 创建时完成前置和后置处理

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactor
y implements AutowireCapableBeanFactory {

    private InstantiationStrategy instantiationStrategy = new CglibSubclassingInsta
ntiationStrategy();

    @Override
    protected Object createBean(String beanName, BeanDefinition beanDefinition, Obj
```

```
ect[] args) throws BeansException {
        Object bean = null;
        try {
            bean = createBeanInstance(beanDefinition, beanName, args);
            // 给 Bean 填充属性
            applyPropertyValues(beanName, bean, beanDefinition);
            // 执行 Bean 的初始化方法和 BeanPostProcessor 的前置和后置处理方法
            bean = initializeBean(beanName, bean, beanDefinition);
        } catch (Exception e) {
            throw new BeansException("Instantiation of bean failed", e);
        }

        addSingleton(beanName, bean);
        return bean;
    }

    public InstantiationStrategy getInstantiationStrategy() {
        return instantiationStrategy;
    }

    public void setInstantiationStrategy(InstantiationStrategy instantiationStrateg
y) {
        this.instantiationStrategy = instantiationStrategy;
    }

    private Object initializeBean(String beanName, Object bean, BeanDefinition bean
Definition) {
        // 1. 执行 BeanPostProcessor Before 处理
        Object wrappedBean = applyBeanPostProcessorsBeforeInitialization(bean, bean
Name);

        // 待完成内容：invokeInitMethods(beanName, wrappedBean, beanDefinition);
        invokeInitMethods(beanName, wrappedBean, beanDefinition);

        // 2. 执行 BeanPostProcessor After 处理
        wrappedBean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
        return wrappedBean;
    }

    private void invokeInitMethods(String beanName, Object wrappedBean, BeanDefinit
ion beanDefinition) {

    }
```

```
    @Override
    public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean,
String beanName) throws BeansException {
        Object result = existingBean;
        for (BeanPostProcessor processor : getBeanPostProcessors()) {
            Object current = processor.postProcessBeforeInitialization(result, bean
Name);
            if (null == current) return result;
            result = current;
        }
        return result;
    }

    @Override
    public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, S
tring beanName) throws BeansException {
        Object result = existingBean;
        for (BeanPostProcessor processor : getBeanPostProcessors()) {
            Object current = processor.postProcessAfterInitialization(result, beanN
ame);
            if (null == current) return result;
            result = current;
        }
        return result;
    }

}
```

- 实现 BeanPostProcessor 接口后，会涉及到两个接口方法，`postProcessBeforeInitialization`、`postProcessAfterInitialization`，分别作用于 Bean 对象执行初始化前后的额外处理。
- 也就是需要在创建 Bean 对象时，在 createBean 方法中添加 `initializeBean(beanName, bean, beanDefinition);` 操作。而这个操作主要主要是对于方法 `applyBeanPostProcessorsBeforeInitialization`、`applyBeanPostProcessorsAfterInitialization` 的使用。
- 另外需要提一下，applyBeanPostProcessorsBeforeInitialization、applyBeanPostProcessorsAfterInitialization 两个方法是在接口类 `AutowireCapableBeanFactory` 中新增加的。

五、测试

1. 事先准备

cn.bugstack.springframework.test.bean.UserDao

```
public class UserDao {

    private static Map<String, String> hashMap = new HashMap<>();

    static {
        hashMap.put("10001", "小傅哥");
        hashMap.put("10002", "八杯水");
        hashMap.put("10003", "阿毛");
    }

    public String queryUserName(String uId) {
        return hashMap.get(uId);
    }

}
```

cn.bugstack.springframework.test.bean.UserService

```
public class UserService {

    private String uId;
    private String company;
    private String location;
    private UserDao userDao;

    public void queryUserInfo() {
        return userDao.queryUserName(uId);
    }

    // ...get/set
}
```

- Dao、Service，是我们平常开发经常使用的场景。在 UserService 中注入 UserDao，这样就能体现出 Bean 属性的依赖了。
- 另外这里新增加了 company、location，两个属性信息，便于测试 BeanPostProcessor、BeanFactoryPostProcessor 两个接口对 Bean 属性信息扩展的作用。

2. 实现 BeanPostProcessor 和 BeanFactoryPostProcessor

cn.bugstack.springframework.test.common.MyBeanFactoryPostProcessor

```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
 throws BeansException {

        BeanDefinition beanDefinition = beanFactory.getBeanDefinition("userService"
);
        PropertyValues propertyValues = beanDefinition.getPropertyValues();

        propertyValues.addPropertyValue(new PropertyValue("company", "改为：字节跳动
"));
    }

}
```

cn.bugstack.springframework.test.common.MyBeanPostProcessor

```
public class MyBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) thr
ows BeansException {
        if ("userService".equals(beanName)) {
            UserService userService = (UserService) bean;
            userService.setLocation("改为：北京");
        }
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) thro
ws BeansException {
        return bean;
    }

}
```

- 如果你在 Spring 中做过一些组件的开发那么一定非常熟悉这两个类，本文的测试也是实现了这两个类，对实例化过程中的 Bean 对象做一些操作。

3. 配置文件

基础配置，无 BeanFactoryPostProcessor、BeanPostProcessor，实现类

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>

    <bean id="userDao" class="cn.bugstack.springframework.test.bean.UserDao"/>

    <bean id="userService" class="cn.bugstack.springframework.test.bean.UserService
">
        <property name="uId" value="10001"/>
        <property name="company" value="腾讯"/>
        <property name="location" value="深圳"/>
        <property name="userDao" ref="userDao"/>
    </bean>

</beans>
```

增强配置，有 BeanFactoryPostProcessor、BeanPostProcessor，实现类

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>

    <bean id="userDao" class="cn.bugstack.springframework.test.bean.UserDao"/>

    <bean id="userService" class="cn.bugstack.springframework.test.bean.UserService
">
        <property name="uId" value="10001"/>
        <property name="company" value="腾讯"/>
        <property name="location" value="深圳"/>
        <property name="userDao" ref="userDao"/>
    </bean>

    <bean class="cn.bugstack.springframework.test.common.MyBeanPostProcessor"/>
    <bean class="cn.bugstack.springframework.test.common.MyBeanFactoryPostProcessor
"/>

</beans>
```

- 这里提供了两个配置文件，一个是不包含 BeanFactoryPostProcessor、BeanPostProcessor，另外一个是包含的。之所以这样配置主要对照验证，在运用 Spring 新增加的应用上下文和不使用的时候，都是怎么操作的。

4. 不用应用上下文

```
@Test
public void test_BeanFactoryPostProcessorAndBeanPostProcessor(){
    // 1.初始化 BeanFactory
    DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();

    // 2. 读取配置文件&注册 Bean
    XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(beanFactory);
    reader.loadBeanDefinitions("classpath:spring.xml");

    // 3. BeanDefinition 加载完成 & Bean 实例化之前，修改 BeanDefinition 的属性值
    MyBeanFactoryPostProcessor beanFactoryPostProcessor = new MyBeanFactoryPostProc
essor();
    beanFactoryPostProcessor.postProcessBeanFactory(beanFactory);

    // 4. Bean 实例化之后，修改 Bean 属性信息
    MyBeanPostProcessor beanPostProcessor = new MyBeanPostProcessor();
    beanFactory.addBeanPostProcessor(beanPostProcessor);

    // 5. 获取 Bean 对象调用方法
    UserService userService = beanFactory.getBean("userService", UserService.class)
;
    String result = userService.queryUserInfo();
    System.out.println("测试结果：" + result);
}
```

- DefaultListableBeanFactory 创建 beanFactory 并使用 XmlBeanDefinitionReader 加载配置文件的方式，还是比较熟悉的。
- 接下来就是对 MyBeanFactoryPostProcessor 和 MyBeanPostProcessor 的处理，一个是在 BeanDefinition 加载完成 & Bean 实例化之前，修改 BeanDefinition 的属性值，另外一个是在 Bean 实例化之后，修改 Bean 属性信息。

测试结果

```
测试结果：小傅哥,改为：字节跳动,改为：北京

Process finished with exit code 0
```

- 通过测试结果可以看到，我们配置的属性信息已经与 spring.xml 配置文件中不一样了。

5. 使用应用上下文

```
@Test
public void test_xml() {
    // 1.初始化 BeanFactory
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplication
Context("classpath:springPostProcessor.xml");

    // 2. 获取Bean 对象调用方法
    UserService userService = applicationContext.getBean("userService", UserService
.class);
    String result = userService.queryUserInfo();
    System.out.println("测试结果：" + result);
}
```

- 另外使用新增加的 ClassPathXmlApplicationContext 应用上下文类，再操作起来就方便多了，*这才是面向用户使用的类*，在这里可以一步把配置文件交给 ClassPathXmlApplicationContext，也不需要管理一些自定义实现的 Spring 接口的类。

测试结果

```
测试结果：小傅哥,改为：字节跳动,改为：北京

Process finished with exit code 0
```

- 这与不用应用上下文的测试结果是一样，不过现在的方式更加方便了。

六、总结

- 本文主要新增了 Spring 框架中两个非常重要的接口 BeanFactoryPostProcess、BeanPostProcessor 同时还添加了关于应用上下文的实现，ApplicationContext 接口的定义是继承 BeanFactory 外新增加功能的接口，它可以满足于自动识别、资源加载、容器事件、监听器等功能，同时例如一些国际化支持、单例 Bean 自动初始化等，也是可以在这个类里实现和扩充的。
- 通过本文的实现一定会非常了解 BeanFactoryPostProcess、BeanPostProcessor，以后再做一些关于 Spring 中间件的开发时，如果需要用到 Bean 对象的获取以及修改一些属性信息，那么就可以使用这两个接口了。同时 BeanPostProcessor 也是实现 AOP 切面技术的关键所在。
- 有人问：面试问那么多，可是工作又用不到，是嘎哈么呢？，嘎哈么，那

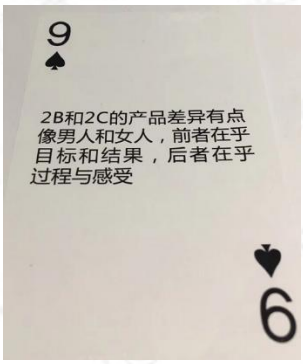
  你说你开车上桥的时候，会每次都撞两边的护栏吗，不撞是吧，那不要修了哇，直接就铺一个平板，还省材料了。其实核心技术的原理学习，是更有助于你完成更复

杂的架构设计，当你的知识能更全面覆盖所承接的需求时，也就能更好的做出合理的架构和落地。

第 08 章：初始化和销毁方法

一、易扩展性

有什么方式，能给代码留条活路？



有人说：人人都是产品经理，那你知道吗，人人也都可以是码农程序员！就像：

- 编程就是；定义属性、创建方法、调用展示
- Java 和 PHP 就像男人和女人，前者在乎架构化模块后，后者在乎那个颜色我喜欢
- 用心写，但不要不做格式化
- 初次和产品对接的三个宝：砖头、铁锹、菜刀，分别保证有用、可用、好用
- 从一行代码到一吨代码，开发越来越难，壁垒也越来越高

其实学会写代码并不难，但学会写好代码却很难。从易阅读上来说你的代码要有准确的命名和清晰的注释、从易使用上来说你的代码要具备设计模式的包装让对外的服务调用更简单、从易扩展上来说你的代码要做好业务和功能的实现分层。在易阅读、易使用、易扩展以及更多编码规范的约束下，还需要在开发完成上线后的交付结果上满足；高可用、高性能、高并发，与此同时你还会接到现有项目中层出不穷来自产品经理新增的需求。

怎么办？知道你在码砖，不知道你在盖哪个猪圈！

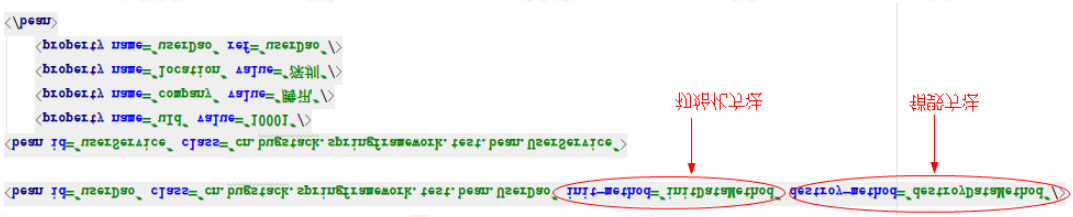
就算码的砖是盖的猪圈，也得因为猪多扩面积、改水槽、加饲料呀，所以根本没法保证你写完的代码就不会加需求。那么新加的需求如果是以破坏了你原有的封装了非常完美 500 行的 `ifelse` 咋办，拆了重盖吗？

兄嘚，给代码留条活路吧！你的代码用上了定义接口吗、接口继承接口吗、接口由抽象类实现吗、类继承的类实现了接口方法吗，而这些操作都是为了让你的程序逻辑做到分层、分区、分块，把核心逻辑层和业务封装层做好隔离，当有业务变化时候，只需要做在业务层完成装配，而底层的核心逻辑服务并不需要频繁变化，它们所增加的接口也更原子化，不具备业务语意。所以这样的实现方式才能给你的代码留条活路。*如果还不是太理解，可以多看看《重学 Java 设计模式》和现在编写的《手撸 Spring》，这里面都有大量的设计模式应用实践*

二、目标

当我们的类创建的 Bean 对象，交给 Spring 容器管理以后，这个类对象就可以被赋予更多的使用能力。就像我们在上一章节已经给类对象添加了修改注册 Bean 定义未实例化前的属性信息修改和实例化过程中的前置和后置处理，这些额外能力的实现，都可以让我们对现有工程中的类对象做相应的扩展处理。

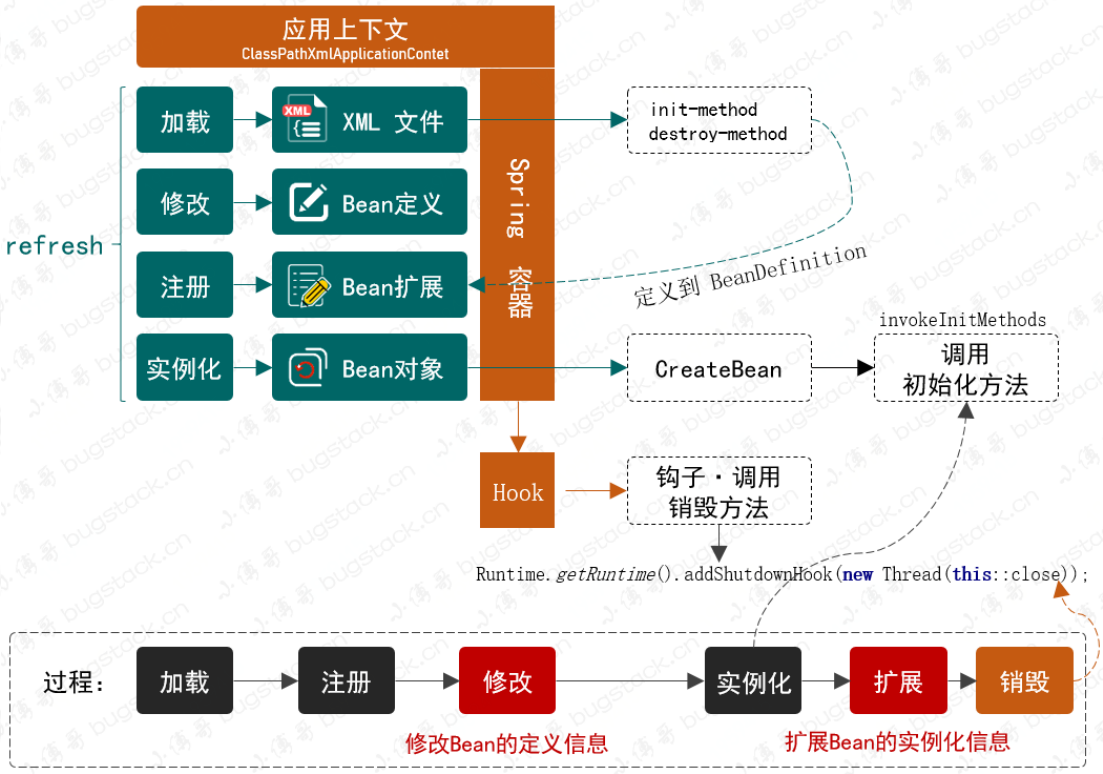
那么除此之外我们还希望可以在 Bean 初始化过程，执行一些操作。比如帮我们做一些数据的加载执行，链接注册中心暴漏 RPC 接口以及在 Web 程序关闭时执行链接断开，内存销毁等操作。*如果说没有 Spring 我们也可以通过构造函数、静态方法以及手动调用的方式实现，但这样的处理方式终究不如把诸如此类的操作都交给 Spring 容器来管理更加合适。* 因此你会看到到 spring.xml 中有如下操作：



- 需要满足用户可以在 xml 中配置初始化和销毁的方法，也可以通过实现类的方式处理，比如我们在使用 Spring 时用到的 InitializingBean, DisposableBean 两个接口。 -其实还可以有一种是注解的方式处理初始化操作，不过目前还没有实现到注解的逻辑，后续再完善此类功能。

三、设计

可能面对像 Spring 这样庞大的框架，对外暴露的接口定义使用或者 xml 配置，完成的一系列扩展性操作，都让 Spring 框架看上去很神秘。其实对于这样在 Bean 容器初始化过程中额外添加的处理操作，无非就是预先执行了一个定义好的接口方法或者是反射调用类中 xml 中配置的方法，最终你只要按照接口定义实现，就会有 Spring 容器在处理的过程中进行调用而已。整体设计结构如下图：



- 在 spring.xml 配置中添加 `init-method、destroy-method` 两个注解，在配置文件加载的过程中，把注解配置一并定义到 BeanDefinition 的属性当中。这样在 initializeBean 初始化操作的工程中，就可以通过反射的方式来调用配置在 Bean 定义属性当中的方法信息了。另外如果是接口实现的方式，那么直接可以通过 Bean 对象调用对应接口定义的方法即可，`((InitializingBean) bean).afterPropertiesSet()`，两种方式达到的效果是一样的。
- 除了在初始化做的操作外，`destroy-method` 和 `DisposableBean` 接口的定义，都会在 Bean 对象初始化完成阶段，执行注册销毁方法的信息到 DefaultSingletonBeanRegistry 类中的 disposableBeans 属性里，这是为了后续统一进行操作。*这里还有一段适配器的使用，因为反射调用和接口直接调用，是两种方式。所以需要使用适配器进行包装，下文代码讲解中参考 DisposableBeanAdapter 的具体实现* -关于销毁方法需要在虚拟机执行关闭之前进行操作，所以这里需要用到一个注册钩子的操作，如：

`Runtime.getRuntime().addShutdownHook(new Thread(() -> System.out.println("close! ")));` *这段代码你可以执行测试*，另外你可以使用手动调用 ApplicationContext.close 方法关闭容器。

四、实现

1. 工程结构

```
small-spring-step-07
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   │   ├── BeanPostProcessor.java
    │           │   │   │   ├── BeanReference.java
    │           │   │   │   ├── ConfigurableBeanFactory.java
    │           │   │   │   └── SingletonBeanRegistry.java
    │           │   │   ├── support
    │           │   │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   │   ├── AbstractBeanFactory.java
    │           │   │   │   ├── BeanDefinitionReader.java
    │           │   │   │   ├── BeanDefinitionRegistry.java
    │           │   │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   │   ├── DefaultListableBeanFactory.java
    │           │   │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   │   ├── DisposableBeanAdapter.java
    │           │   │   │   ├── InstantiationStrategy.java
    │           │   │   │   └── SimpleInstantiationStrategy.java
    │           │   │   ├── support
    │           │   │   │   └── XmlBeanDefinitionReader.java
    │           │   │   ├── BeanFactory.java
    │           │   │   ├── ConfigurableListableBeanFactory.java
    │           │   │   ├── DisposableBean.java
    │           │   │   ├── HierarchicalBeanFactory.java
```

```
    │           │   │   ├── InitializingBean.java
    │           │   │   └── ListableBeanFactory.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
    │           │   └── PropertyValues.java
    │           ├── context
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   └── ClassPathXmlApplicationContext.java
    │           │   ├── ApplicationContext.java
    │           │   └── ConfigurableApplicationContext.java
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           └── utils
    │               └── ClassUtils.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── UserDao.java
                │   └── UserService.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

Spring 应用上下文和对 Bean 对象扩展机制的类关系，如图 8-4

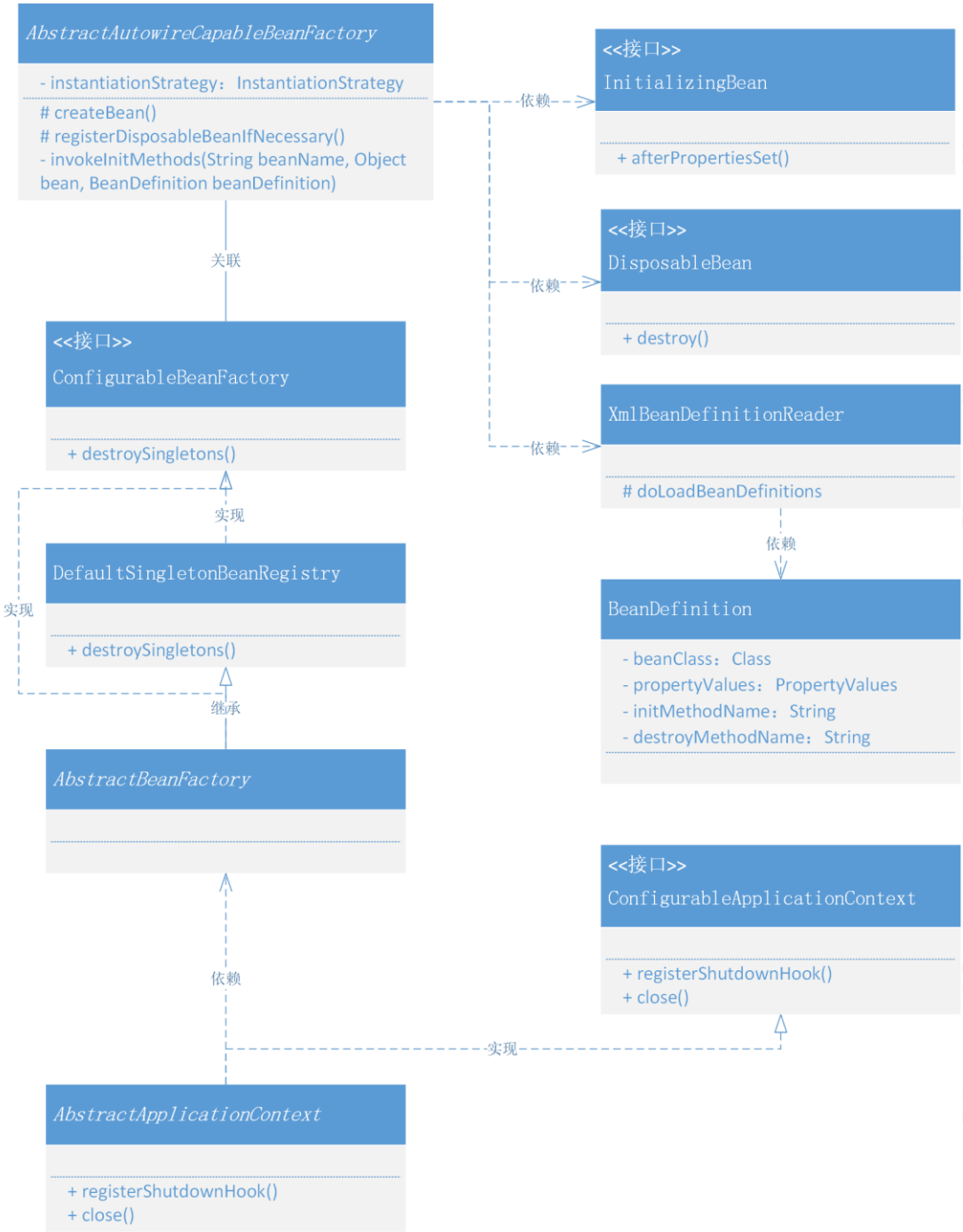


图 8-4

- 以上整个类图结构描述出来的就是本次新增 Bean 实例化过程中的初始化方法和销毁方法。
- 因为我们一共实现了两种方式的初始化和销毁方法，xml 配置和定义接口，所以这里既有 InitializingBean、DisposableBean 也有需要 XmlBeanDefinitionReader 加载 spring.xml 配置信息到 BeanDefinition 中。
- 另外接口 ConfigurableBeanFactory 定义了 destroySingletons 销毁方法，并由 AbstractBeanFactory 继承的父类 DefaultSingletonBeanRegistry 实现 ConfigurableBeanFactory 接口定义的 destroySingletons 方法。*这种方式的设计可*

能数程序员是没有用过的，都是用的谁实现接口谁完成实现类，而不是把实现接口的操作又交给继承的父类处理。所以这块还是蛮有意思的，是一种不错的隔离分层服务的设计方式

- 最后就是关于向虚拟机注册钩子，保证在虚拟机关闭之前，执行销毁操作。`Runtime.getRuntime().addShutdownHook(new Thread(() -> System.out.println("close! ")));`

2. 定义初始化和销毁方法的接口

cn.bugstack.springframework.beans.factory.InitializingBean

```
public interface InitializingBean {

    /**
     * Bean 处理了属性填充后调用
     *
     * @throws Exception
     */
    void afterPropertiesSet() throws Exception;

}
```

cn.bugstack.springframework.beans.factory.DisposableBean

```
public interface DisposableBean {

    void destroy() throws Exception;

}
```

- InitializingBean、DisposableBean，两个接口方法还是比较常用的，在一些需要结合 Spring 实现的组件中，经常会使用这两个方法来做一些参数的初始化和销毁操作。比如接口暴漏、数据库数据读取、配置文件加载等等。

3. Bean 属性定义新增初始化和销毁

cn.bugstack.springframework.beans.factory.config.BeanDefinition

```
public class BeanDefinition {

    private Class beanClass;

    private PropertyValues propertyValues;
```

```
    private String initMethodName;

    private String destroyMethodName;

    // ...get/set
}
```

- 在 BeanDefinition 新增加了两个属性：initMethodName、destroyMethodName，这两个属性是为了在 spring.xml 配置的 Bean 对象中，可以配置 `init-method="initDataMethod" destroy-method="destroyDataMethod"` 操作，最终实现接口的效果是一样的。*只不过一个是接口方法的直接调用，另外是一个在配置文件中读取到方法反射调用*

4. 执行 Bean 对象的初始化方法

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactory implements AutowireCapableBeanFactory {

    private InstantiationStrategy instantiationStrategy = new CglibSubclassingInstantiationStrategy();

    @Override
    protected Object createBean(String beanName, BeanDefinition beanDefinition, Object[] args) throws BeansException {
        Object bean = null;
        try {
            bean = createBeanInstance(beanDefinition, beanName, args);
            // 给 Bean 填充属性
            applyPropertyValues(beanName, bean, beanDefinition);
            // 执行 Bean 的初始化方法和 BeanPostProcessor 的前置和后置处理方法
            bean = initializeBean(beanName, bean, beanDefinition);
        } catch (Exception e) {
            throw new BeansException("Instantiation of bean failed", e);
        }

        // ...

        addSingleton(beanName, bean);
        return bean;
```

```
    }

    private Object initializeBean(String beanName, Object bean, BeanDefinition bean
Definition) {
        // 1. 执行 BeanPostProcessor Before 处理
        Object wrappedBean = applyBeanPostProcessorsBeforeInitialization(bean, bean
Name);

        // 执行 Bean 对象的初始化方法
        try {
            invokeInitMethods(beanName, wrappedBean, beanDefinition);
        } catch (Exception e) {
            throw new BeansException("Invocation of init method of bean[" + beanNam
e + "] failed", e);
        }

        // 2. 执行 BeanPostProcessor After 处理
        wrappedBean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
        return wrappedBean;
    }

    private void invokeInitMethods(String beanName, Object bean, BeanDefinition bea
nDefinition) throws Exception {
        // 1. 实现接口 InitializingBean
        if (bean instanceof InitializingBean) {
            ((InitializingBean) bean).afterPropertiesSet();
        }

        // 2. 配置信息 init-method {判断是为了避免二次执行销毁}
        String initMethodName = beanDefinition.getInitMethodName();
        if (StrUtil.isNotEmpty(initMethodName)) {
            Method initMethod = beanDefinition.getBeanClass().getMethod(initMethodN
ame);
            if (null == initMethod) {
                throw new BeansException("Could not find an init method named '" +
initMethodName + "' on bean with name '" + beanName + "'");
            }
            initMethod.invoke(bean);
        }
    }

}
```

- 抽象类 AbstractAutowireCapableBeanFactory 中的 createBean 是用来创建 Bean 对象的方法，在这个方法中我们之前已经扩展了 BeanFactoryPostProcessor、BeanPostProcessor 操作，这里我们继续完善执行 Bean 对象的初始化方法的处理动作。
- 在方法 invokeInitMethods 中，主要分为两块来执行实现了 InitializingBean 接口的操作，处理 afterPropertiesSet 方法。另外一个是判断配置信息 init-method 是否存在，执行反射调用 initMethod.invoke(bean)。这两种方式都可以在 Bean 对象初始化过程中进行处理加载 Bean 对象中的初始化操作，让使用者可以额外新增加自己想要的动作。

5. 定义销毁方法适配器(接口和配置)

cn.bugstack.springframework.beans.factory.support.DisposableBeanAdapter

```
public class DisposableBeanAdapter implements DisposableBean {

    private final Object bean;
    private final String beanName;
    private String destroyMethodName;

    public DisposableBeanAdapter(Object bean, String beanName, BeanDefinition beanD
efinition) {
        this.bean = bean;
        this.beanName = beanName;
        this.destroyMethodName = beanDefinition.getDestroyMethodName();
    }

    @Override
    public void destroy() throws Exception {
        // 1. 实现接口 DisposableBean
        if (bean instanceof DisposableBean) {
            ((DisposableBean) bean).destroy();
        }

        // 2. 配置信息 destroy-method {判断是为了避免二次执行销毁}
        if (StrUtil.isNotEmpty(destroyMethodName) && !(bean instanceof DisposableBe
an && "destroy".equals(this.destroyMethodName))) {
            Method destroyMethod = bean.getClass().getMethod(destroyMethodName);
            if (null == destroyMethod) {
                throw new BeansException("Couldn't find a destroy method named '" +
 destroyMethodName + "' on bean with name '" + beanName + "'");
            }
```

```
            destroyMethod.invoke(bean);
        }

    }

}
```

- 可能你会想这里怎么有一个适配器的类呢，因为销毁方法有两种甚至多种方式，目前有实现接口 DisposableBean、配置信息 destroy-method，两种方式。而这两种方式的销毁动作是由 AbstractApplicationContext 在注册虚拟机钩子后看，虚拟机关闭前执行的操作动作。
- 那么在销毁执行时不太希望还得关注都销毁那些类型的方法，它的使用上更希望是有一个统一的接口进行销毁，所以这里就新增了适配类，做统一处理。

6. 创建 Bean 时注册销毁方法对象

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactor
y implements AutowireCapableBeanFactory {

    private InstantiationStrategy instantiationStrategy = new CglibSubclassingInsta
ntiationStrategy();

    @Override
    protected Object createBean(String beanName, BeanDefinition beanDefinition, Obj
ect[] args) throws BeansException {
        Object bean = null;
        try {
            bean = createBeanInstance(beanDefinition, beanName, args);
            // 给 Bean 填充属性
            applyPropertyValues(beanName, bean, beanDefinition);
            // 执行 Bean 的初始化方法和 BeanPostProcessor 的前置和后置处理方法
            bean = initializeBean(beanName, bean, beanDefinition);
        } catch (Exception e) {
            throw new BeansException("Instantiation of bean failed", e);
        }

        // 注册实现了 DisposableBean 接口的 Bean 对象
        registerDisposableBeanIfNecessary(beanName, bean, beanDefinition);
```

```
        addSingleton(beanName, bean);
        return bean;
    }

    protected void registerDisposableBeanIfNecessary(String beanName, Object bean,
BeanDefinition beanDefinition) {
        if (bean instanceof DisposableBean || StrUtil.isNotEmpty(beanDefinition.get
DestroyMethodName())) {
            registerDisposableBean(beanName, new DisposableBeanAdapter(bean, beanNa
me, beanDefinition));
        }
    }

}
```

- 在创建 Bean 对象的实例的时候，需要把销毁方法保存起来，方便后续执行销毁动作进行调用。
- 那么这个销毁方法的具体方法信息，会被注册到 DefaultSingletonBeanRegistry 中新增加的 `Map<String, DisposableBean> disposableBeans` 属性中去，因为这个接口的方法最终可以被类 AbstractApplicationContext 的 close 方法通过 `getBeanFactory().destroySingletons()` 调用。
- 在注册销毁方法的时候，会根据是接口类型和配置类型统一交给 DisposableBeanAdapter 销毁适配器类来做统一处理。*实现了某个接口的类可以被 instanceof 判断或者强转后调用接口方法*

7. 虚拟机关闭钩子注册调用销毁方法

cn.bugstack.springframework.context.ConfigurableApplicationContext

```
public interface ConfigurableApplicationContext extends ApplicationContext {

    void refresh() throws BeansException;

    void registerShutdownHook();

    void close();

}
```

- 首先我们需要在 ConfigurableApplicationContext 接口中定义注册虚拟机钩子的方法 `registerShutdownHook` 和手动执行关闭的方法 `close`。

cn.bugstack.springframework.context.support.AbstractApplicationContext

```
public abstract class AbstractApplicationContext extends DefaultResourceLoader impl
ements ConfigurableApplicationContext {

    // ...

    @Override
    public void registerShutdownHook() {
        Runtime.getRuntime().addShutdownHook(new Thread(this::close));
    }

    @Override
    public void close() {
        getBeanFactory().destroySingletons();
    }

}
```

- 这里主要体现了关于注册钩子和关闭的方法实现，上文提到过的 Runtime.getRuntime().addShutdownHook，可以尝试验证。在一些中间件和监控系统的设计中也可以用得到，比如监测服务器宕机，执行备机启动操作。

五、测试

1. 事先准备

cn.bugstack.springframework.test.bean.UserDao

```
public class UserDao {

    private static Map<String, String> hashMap = new HashMap<>();

    public void initDataMethod(){
        System.out.println("执行：init-method");
        hashMap.put("10001", "小傅哥");
        hashMap.put("10002", "八杯水");
        hashMap.put("10003", "阿毛");
    }

    public void destroyDataMethod(){
```

```
        System.out.println("执行：destroy-method");
        hashMap.clear();
    }

    public String queryUserName(String uId) {
        return hashMap.get(uId);
    }

}
```

cn.bugstack.springframework.test.bean.UserService

```
public class UserService implements InitializingBean, DisposableBean {

    private String uId;
    private String company;
    private String location;
    private UserDao userDao;

    @Override
    public void destroy() throws Exception {
        System.out.println("执行：UserService.destroy");
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("执行：UserService.afterPropertiesSet");
    }

    // ...get/set
}
```

- UserDao，修改了之前使用 static 静态块初始化数据的方式，改为提供 initDataMethod 和 destroyDataMethod 两个更优雅的操作方式进行处理。
- UserService，以实现接口 InitializingBean, DisposableBean 的两个方法 destroy()、afterPropertiesSet()，处理相应的初始化和销毁方法的动作。*afterPropertiesSet，方法名字很好，在属性设置后执行*

2. 配置文件

基础配置，无 BeanFactoryPostProcessor、BeanPostProcessor，实现类

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
```

```
    <bean id="userDao" class="cn.bugstack.springframework.test.bean.UserDao" init-
method="initDataMethod" destroy-method="destroyDataMethod"/>

    <bean id="userService" class="cn.bugstack.springframework.test.bean.UserService
">
        <property name="uId" value="10001"/>
        <property name="company" value="腾讯"/>
        <property name="location" value="深圳"/>
        <property name="userDao" ref="userDao"/>
    </bean>

</beans>
```

- 配置文件中主要是新增了，`init-method="initDataMethod" destroy-method="destroyDataMethod"`，这样两个配置。从源码的学习中可以知道，这两个配置是为了加入到 BeanDefinition 定义类之后写入到类 DefaultListableBeanFactory 中的 beanDefinitionMap 属性中去。

3. 单元测试

```
@Test
public void test_xml() {
    // 1.初始化 BeanFactory
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplication
Context("classpath:spring.xml");
    applicationContext.registerShutdownHook();

    // 2. 获取Bean对象调用方法
    UserService userService = applicationContext.getBean("userService", UserService
.class);
    String result = userService.queryUserInfo();
    System.out.println("测试结果：" + result);
}
```

- 测试方法中新增加了一个，注册钩子的动作。`applicationContext.registerShutdownHook();`

测试结果

```
执行：init-method
执行：UserService.afterPropertiesSet
测试结果：小傅哥,腾讯,深圳
执行：UserService.destroy
```

```
执行：destroy-method

Process finished with exit code 0
```

- 从测试结果可以看到，我们的新增加的初始和销毁方法已经可以如期输出结果了。

六、总结

- 本文主要完成了关于初始和销毁在使用接口定义 `implements InitializingBean, DisposableBean` 和在 spring.xml 中配置 `init-method="initDataMethod" destroy-method="destroyDataMethod"` 的两种具体在 `AbstractAutowireCapableBeanFactory` 完成初始方法和 `AbstractApplicationContext` 处理销毁动作的具体实现过程。
- 通过本文的实现内容，可以看到目前这个 Spring 框架对 Bean 的操作越来越完善了，可扩展性也不断的增强。你既可以在 Bean 注册完成实例化前进行 BeanFactoryPostProcessor 操作，也可以在 Bean 实例化过程中执行前置和后置操作，现在又可以执行 Bean 的初始化方法和销毁方法。所以一个简单的 Bean 对象，已经被赋予了各种扩展能力。
- 在学习和动手实践 Spring 框架学习的过程中，特别要注意的是它对接口和抽象类的把握和使用，尤其遇到类似，A 继承 B 实现 C 时，C 的接口方法由 A 继承的父类 B 实现，这样的操作都蛮有意思的。也是可以复用到通常的业务系统开发中进行处理一些复杂逻辑的功能分层，做到程序的可扩展、易维护等特性。

第 09 章：Aware 感知容器对象

一、设计能力

同事写的代码，我竟丝毫看不懂！

大佬的代码，就像 **“赖蛤蟆泡青蛙，长的丑玩的花”**：一个类实现了多个接口、继承的类又继承了其他类、接口还可以和接口继承、实现接口的抽象类再由类实现抽象类方法、类 A 继承的类 B 实现了类 A 实现的接口 C，等等。

看上去复杂又难懂的代码，却又能一次次满足需求的高效迭代和顺利扩展，而像螺丝钉一样搬砖的你，只是在大佬写的代码里，完成某个接口下的一小块功能，甚至写完了也不知道怎么就被调用运行了，整个过程像看 Spring 源码一样神奇，跳来跳去的摸不着头绪！

其实这主要是因为你的代码是否运用了设计模式，当然设计模式也没那么神奇，就像你们两家都是 120 平米的房子，他家有三室两厅一厨一卫，南北通透，全阳采光。但你家就不一样了，你家是锅碗瓢盆、卫浴马桶、沙发茶几还有那 1.8 的双人床，在 120 平米的房子里敞开了放，没有动静隔离，也没有干湿分离，纯自由发挥。*所以你的代码看上去就乱的很！*

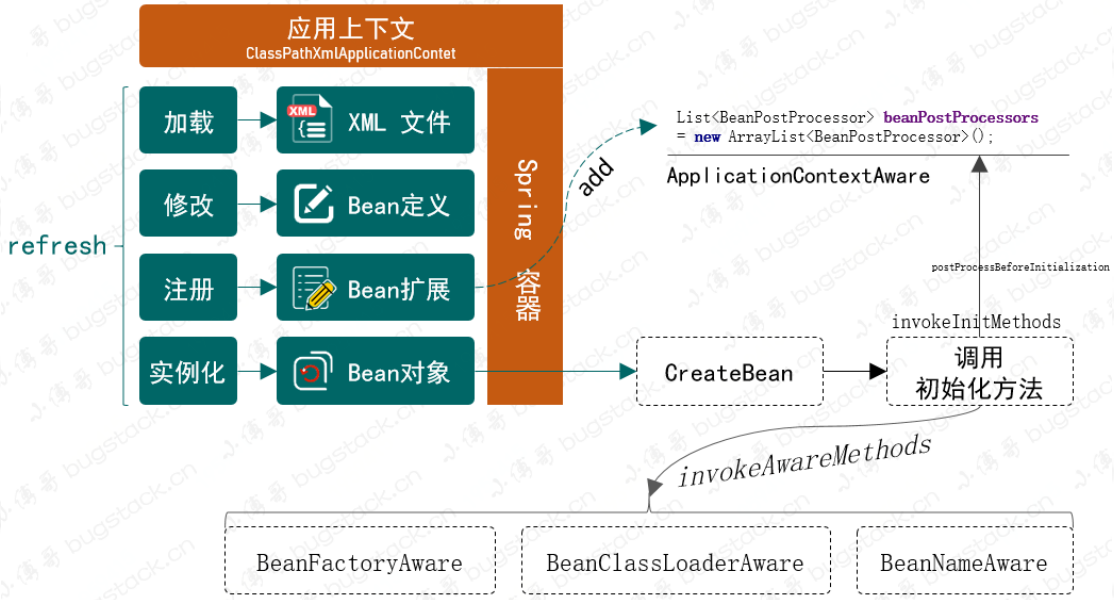
二、目标

目前已实现的 Spring 框架，在 Bean 操作上能提供出的能力，包括：Bean 对象的定义和注册，以及在操作 Bean 对象过程中执行的，BeanFactoryPostProcessor、BeanPostProcessor、InitializingBean、DisposableBean，以及在 XML 新增的一些配置处理，让我们可以 Bean 对象有更强的操作性。
那么，如果我们想获得 Spring 框架提供的 BeanFactory、ApplicationContext、BeanClassLoader 等这些能力做一些扩展框架的使用时该怎么操作呢。所以我们本章节希望在 Spring 框架中提供一种能感知容器操作的接口，如果谁实现了这样的一个接口，就可以获取接口入参中的各类能力。

三、设计

如果说我希望拿到 Spring 框架中一些提供的资源，那么首先需要考虑以一个什么方式去获取，之后你定义出来的获取方式，在 Spring 框架中该怎么去承接，实现了这两项内容，就可以扩展出你需要的一些属于 Spring 框架本身的能力了。

在关于 Bean 对象实例化阶段我们操作过一些额外定义、属性、初始化和销毁的操作，其实我们如果像获取 Spring 一些如 BeanFactory、ApplicationContext 时，也可以通过此类方式进行实现。那么我们需要定义一个标记性的接口，这个接口不需要有方法，它只起到标记作用就可以，而具体的功能由继承此接口的其他功能性接口定义具体方法，最终这个接口就可以通过 `instanceof` 进行判断和调用了。整体设计结构如下图：



- 定义接口 Aware，在 Spring 框架中它是一种感知标记性接口，具体的子类定义和实现能感知容器中的相关对象。*也就是通过这个桥梁，向具体的实现类中提供容器服务*
- 继承 Aware 的接口包括：BeanFactoryAware、BeanClassLoaderAware、BeanNameAware 和 ApplicationContextAware，当然在 Spring 源码中还有一些其他关于注解的，不过目前我们还是用不到。
- 在具体的接口实现过程中你可以看到，一部分(*BeanFactoryAware*、*BeanClassLoaderAware*、*BeanNameAware*)在 factory 的 support 文件夹下，另外 ApplicationContextAware 是在 context 的 support 中，这是因为不同的内容获取需要在不同的包下提供。所以，在 AbstractApplicationContext 的具体实现中会用到向 beanFactory 添加 BeanPostProcessor 内容的 `ApplicationContextAwareProcessor` 操作，最后由 AbstractAutowireCapableBeanFactory 创建 createBean 时处理相应的调用操作。

关于 applyBeanPostProcessorsBeforeInitialization 已经在前面章节中实现过，如果忘记可以往前翻翻

四、实现

1. 工程结构

```
small-spring-step-08
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   │   ├── BeanPostProcessor.java
    │           │   │   │   ├── BeanReference.java
    │           │   │   │   ├── ConfigurableBeanFactory.java
    │           │   │   │   └── SingletonBeanRegistry.java
    │           │   │   ├── support
    │           │   │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   │   ├── AbstractBeanFactory.java
    │           │   │   │   ├── BeanDefinitionReader.java
    │           │   │   │   ├── BeanDefinitionRegistry.java
    │           │   │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   │   ├── DefaultListableBeanFactory.java
    │           │   │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   │   ├── DisposableBeanAdapter.java
    │           │   │   │   ├── InstantiationStrategy.java
    │           │   │   │   └── SimpleInstantiationStrategy.java
    │           │   │   ├── support
    │           │   │   │   └── XmlBeanDefinitionReader.java
    │           │   │   ├── Aware.java
    │           │   │   ├── BeanClassLoaderAware.java
    │           │   │   ├── BeanFactory.java
    │           │   │   ├── BeanFactoryAware.java
    │           │   │   ├── BeanNameAware.java
    │           │   │   ├── ConfigurableListableBeanFactory.java
```

```
    │           │   ├── DisposableBean.java
    │           │   ├── HierarchicalBeanFactory.java
    │           │   ├── InitializingBean.java
    │           │   └── ListableBeanFactory.java
    │           ├── BeansException.java
    │           ├── PropertyValue.java
    │           └── PropertyValues.java
    │           ├── context
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   ├── ApplicationContextAwareProcessor.java
    │           │   │   └── ClassPathXmlApplicationContext.java
    │           │   ├── ApplicationContext.java
    │           │   ├── ApplicationContextAware.java
    │           │   └── ConfigurableApplicationContext.java
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           └── utils
    │               └── ClassUtils.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── UserDao.java
                │   └── UserService.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

Spring 感知接口的设计和实现类关系，如图 9-2

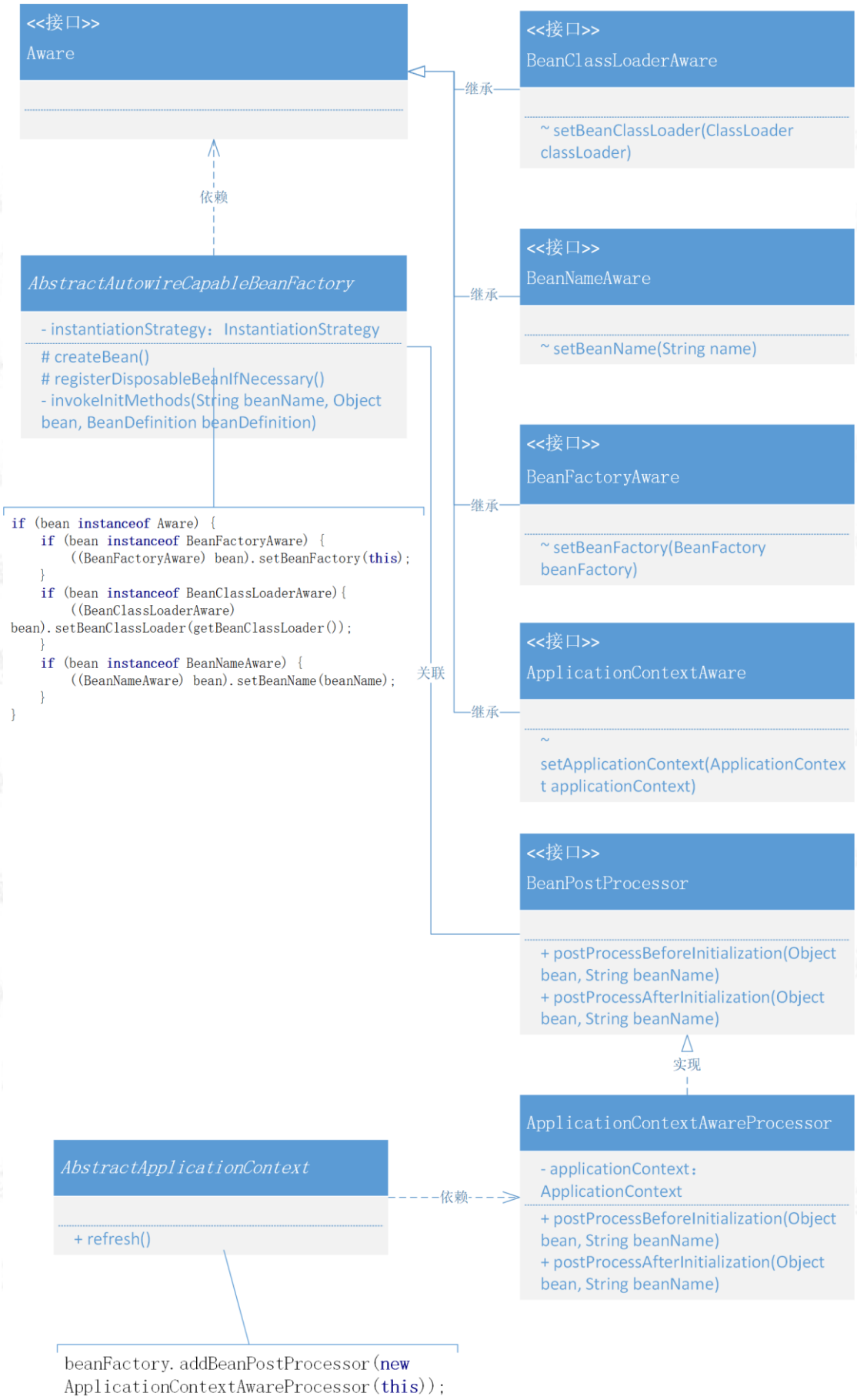

<<接口>>
Aware
<<接口>>
BeanClassLoaderAware
~ setBeanClassLoader(ClassLoader classLoader)
继承
依赖
AbstractAutowireCapableBeanFactory
- instantiationStrategy：InstantiationStrategy
createBean()
registerDisposableBeanIfNecessary()
- invokeInitMethods(String beanName, Object bean, BeanDefinition beanDefinition)
<<接口>>
BeanNameAware
~ setBeanName(String name)
继承
<<接口>>
BeanFactoryAware
~ setBeanFactory(BeanFactory beanFactory)
继承
if (bean instanceof Aware) {
if (bean instanceof BeanFactoryAware) {
((BeanFactoryAware) bean).setBeanFactory(this);
}
if (bean instanceof BeanClassLoaderAware){
((BeanClassLoaderAware) bean).setBeanClassLoader(getBeanClassLoader());
}
if (bean instanceof BeanNameAware) {
((BeanNameAware) bean).setBeanName(beanName);
}
}
关联
<<接口>>
ApplicationContextAware
~ setApplicationContext(ApplicationContext applicationContext)
继承
<<接口>>
BeanPostProcessor
+ postProcessBeforeInitialization(Object bean, String beanName)
+ postProcessAfterInitialization(Object bean, String beanName)
实现
ApplicationContextAwareProcessor
- applicationContext：ApplicationContext
+ postProcessBeforeInitialization(Object bean, String beanName)
+ postProcessAfterInitialization(Object bean, String beanName)
AbstractApplicationContext
+ refresh()
依赖
beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));

图 9-2

- 以上整个类关系就是关于 Aware 感知的定义和对容器感知的实现。
- Aware 有四个继承的接口，其他这些接口的继承都是为了继承一个标记，有了标记的存在更方便类的操作和具体判断实现。
- 另外由于 ApplicationContext 并不是在 AbstractAutowireCapableBeanFactory 中 createBean 方法下的内容，所以需要像容器中注册 `addBeanPostProcessor` ，再由 createBean 统一调用 applyBeanPostProcessorsBeforeInitialization 时进行操作。

2. 定义标记接口

cn.bugstack.springframework.beans.factory.Aware

```
/**
 * Marker superinterface indicating that a bean is eligible to be
 * notified by the Spring container of a particular framework object
 * through a callback-style method.  Actual method signature is
 * determined by individual subinterfaces, but should typically
 * consist of just one void-returning method that accepts a single
 * argument.
 *
 * 标记类接口，实现该接口可以被 Spring 容器感知
 *
 */
public interface Aware {
}
```

- 在 Spring 中有特别多类似这样的标记接口的设计方式，它们的存在就像是一种标签一样，可以方便统一摘取出属于此类接口的实现类，通常会有 instanceof 一起判断使用。

3. 容器感知类

3.1 BeanFactoryAware

cn.bugstack.springframework.beans.factory.BeanFactoryAware

```
public interface BeanFactoryAware extends Aware {

    void setBeanFactory(BeanFactory beanFactory) throws BeansException;
```

```
}
```

- Interface to be implemented by beans that wish to be aware of their owning {@link BeanFactory}.
- 实现此接口，既能感知到所属的 BeanFactory

3.2 BeanClassLoaderAware

cn.bugstack.springframework.beans.factory.BeanClassLoaderAware

```
public interface BeanClassLoaderAware extends Aware{

    void setBeanClassLoader(ClassLoader classLoader);

}
```

- Callback that allows a bean to be aware of the bean{@link ClassLoader class loader}; that is, the class loader used by the present bean factory to load bean classes.
- 实现此接口，既能感知到所属的 ClassLoader

3.3 BeanNameAware

cn.bugstack.springframework.beans.factory.BeanNameAware

```
public interface BeanNameAware extends Aware {

    void setBeanName(String name);

}
```

- Interface to be implemented by beans that want to be aware of their bean name in a bean factory.
- 实现此接口，既能感知到所属的 BeanName

3.4 ApplicationContextAware

cn.bugstack.springframework.context.ApplicationContextAware

```
public interface ApplicationContextAware extends Aware {

    void setApplicationContext(ApplicationContext applicationContext) throws BeansE
```

```
xception;

}
```

- Interface to be implemented by any object that wishes to be notifiedof the {@link ApplicationContext} that it runs in.
- 实现此接口，既能感知到所属的 ApplicationContext

4. 包装处理器(ApplicationContextAwareProcessor)

cn.bugstack.springframework.context.support.ApplicationContextAwareProcessor

```
public class ApplicationContextAwareProcessor implements BeanPostProcessor {

    private final ApplicationContext applicationContext;

    public ApplicationContextAwareProcessor(ApplicationContext applicationContext)
{
        this.applicationContext = applicationContext;
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) thr
ows BeansException {
        if (bean instanceof ApplicationContextAware){
            ((ApplicationContextAware) bean).setApplicationContext(applicationConte
xt);
        }
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) thro
ws BeansException {
        return bean;
    }

}
```

- 由于 ApplicationContext 的获取并不能直接在创建 Bean 时候就可以拿到，所以需要在 refresh 操作时，把 ApplicationContext 写入到一个包装的 BeanPostProcessor 中去，再由

AbstractAutowireCapableBeanFactory.applyBeanPostProcessorsBeforeInitialization 方法调用。

5. 注册 BeanPostProcessor

cn.bugstack.springframework.context.support.AbstractApplicationContext

```
public abstract class AbstractApplicationContext extends DefaultResourceLoader implements ConfigurableApplicationContext {

    @Override
    public void refresh() throws BeansException {
        // 1. 创建 BeanFactory，并加载 BeanDefinition
        refreshBeanFactory();

        // 2. 获取 BeanFactory
        ConfigurableListableBeanFactory beanFactory = getBeanFactory();

        // 3. 添加 ApplicationContextAwareProcessor，让继承自 ApplicationContextAware 的 Bean 对象都能感知所属的 ApplicationContext
        beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));

        // 4. 在 Bean 实例化之前，执行 BeanFactoryPostProcessor (Invoke factory processors registered as beans in the context.)
        invokeBeanFactoryPostProcessors(beanFactory);

        // 5. BeanPostProcessor 需要提前于其他 Bean 对象实例化之前执行注册操作
        registerBeanPostProcessors(beanFactory);

        // 6. 提前实例化单例 Bean 对象
        beanFactory.preInstantiateSingletons();
    }

  // ...
}
```

- refresh() 方法就是整个 Spring 容器的操作过程，与上一章节对比，本次新增加了关于 addBeanPostProcessor 的操作。
- 添加 ApplicationContextAwareProcessor，让继承自 ApplicationContextAware 的 Bean 对象都能感知所属的 ApplicationContext。

6. 感知调用操作

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactor
y implements AutowireCapableBeanFactory {

    private InstantiationStrategy instantiationStrategy = new CglibSubclassingInsta
ntiationStrategy();

    @Override
    protected Object createBean(String beanName, BeanDefinition beanDefinition, Obj
ect[] args) throws BeansException {
        Object bean = null;
        try {
            bean = createBeanInstance(beanDefinition, beanName, args);
            // 给 Bean 填充属性
            applyPropertyValues(beanName, bean, beanDefinition);
            // 执行 Bean 的初始化方法和 BeanPostProcessor 的前置和后置处理方法
            bean = initializeBean(beanName, bean, beanDefinition);
        } catch (Exception e) {
            throw new BeansException("Instantiation of bean failed", e);
        }

        // 注册实现了 DisposableBean 接口的 Bean 对象
        registerDisposableBeanIfNecessary(beanName, bean, beanDefinition);

        addSingleton(beanName, bean);
        return bean;
    }

    private Object initializeBean(String beanName, Object bean, BeanDefinition bean
Definition) {

        // invokeAwareMethods
        if (bean instanceof Aware) {
            if (bean instanceof BeanFactoryAware) {
                ((BeanFactoryAware) bean).setBeanFactory(this);
            }
            if (bean instanceof BeanClassLoaderAware){
                ((BeanClassLoaderAware) bean).setBeanClassLoader(getBeanClassLoader
());
            }
```

```
            if (bean instanceof BeanNameAware) {
                ((BeanNameAware) bean).setBeanName(beanName);
            }
        }

        // 1. 执行 BeanPostProcessor Before 处理
        Object wrappedBean = applyBeanPostProcessorsBeforeInitialization(bean, beanName);

        // 执行 Bean 对象的初始化方法
        try {
            invokeInitMethods(beanName, wrappedBean, beanDefinition);
        } catch (Exception e) {
            throw new BeansException("Invocation of init method of bean[" + beanName + "] failed", e);
        }

        // 2. 执行 BeanPostProcessor After 处理
        wrappedBean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
        return wrappedBean;
    }


    @Override
    public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName) throws BeansException {
        Object result = existingBean;
        for (BeanPostProcessor processor : getBeanPostProcessors()) {
            Object current = processor.postProcessBeforeInitialization(result, beanName);
            if (null == current) return result;
            result = current;
        }
        return result;
    }

    @Override
    public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName) throws BeansException {
        Object result = existingBean;
        for (BeanPostProcessor processor : getBeanPostProcessors()) {
            Object current = processor.postProcessAfterInitialization(result, beanName);
```

```
            if (null == current) return result;
            result = current;
        }
        return result;
    }

}
```

- 这里我们去掉了一些类的内容，只保留关于本次 Aware 感知接口的操作。
- 首先在 initializeBean 中，通过判断 `bean instanceof Aware`，调用了三个接口方法，`BeanFactoryAware.setBeanFactory(this)`、`BeanClassLoaderAware.setBeanClassLoader(getBeanClassLoader())`、`BeanNameAware.setBeanName(beanName)`，这样就能通知到已经实现了此接口的类。
- 另外我们还向 BeanPostProcessor 中添加了 `ApplicationContextAwareProcessor`，此时在这个方法中也会被调用到具体的类实现，得到一个 ApplicationContex 属性。

五、测试

1. 事先准备

cn.bugstack.springframework.test.bean.UserDao

```
public class UserDao {

    private static Map<String, String> hashMap = new HashMap<>();

    public void initDataMethod(){
        System.out.println("执行：init-method");
        hashMap.put("10001", "小傅哥");
        hashMap.put("10002", "八杯水");
        hashMap.put("10003", "阿毛");
    }

    public void destroyDataMethod(){
        System.out.println("执行：destroy-method");
        hashMap.clear();
    }

    public String queryUserName(String uId) {
        return hashMap.get(uId);
```

```
    }

}
```

cn.bugstack.springframework.test.bean.UserService

```
public class UserService implements BeanNameAware, BeanClassLoaderAware, Applicatio
nContextAware, BeanFactoryAware {

    private ApplicationContext applicationContext;
    private BeanFactory beanFactory;

    private String uId;
    private String company;
    private String location;
    private UserDao userDao;

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        this.beanFactory = beanFactory;
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws
 BeansException {
        this.applicationContext = applicationContext;
    }

    @Override
    public void setBeanName(String name) {
        System.out.println("Bean Name is: " + name);
    }

    @Override
    public void setBeanClassLoader(ClassLoader classLoader) {
        System.out.println("ClassLoader: " + classLoader);
    }

    // ...get/set
}
```

- UserDao 本次并没有什么改变，还是提供了关于初始化的方法，并在 Spring.xml 中提供 init-method、destroy-method 配置信息。

- UserService 新增加，BeanNameAware, BeanClassLoaderAware, ApplicationContextAware, BeanFactoryAware，四个感知的实现类，并在类中实现相应的接口方法。

2. 配置文件

基础配置，无 BeanFactoryPostProcessor、BeanPostProcessor，实现类

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>

    <bean id="userDao" class="cn.bugstack.springframework.test.bean.UserDao" init-method="initDataMethod" destroy-method="destroyDataMethod"/>

    <bean id="userService" class="cn.bugstack.springframework.test.bean.UserService">
        <property name="uId" value="10001"/>
        <property name="company" value="腾讯"/>
        <property name="location" value="深圳"/>
        <property name="userDao" ref="userDao"/>
    </bean>

</beans>
```

- 本章节中并没有额外新增加配置信息，与上一章节内容相同。

3. 单元测试

```
@Test
public void test_xml() {
    // 1.初始化 BeanFactory
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("classpath:spring.xml");
    applicationContext.registerShutdownHook();

    // 2. 获取Bean对象调用方法
    UserService userService = applicationContext.getBean("userService", UserService.class);
    String result = userService.queryUserInfo();
    System.out.println("测试结果：" + result);
    System.out.println("ApplicationContextAware：" +userService.getApplicationContext());
```

```
    System.out.println("BeanFactoryAware: "+userService.getBeanFactory());
}
```

- 测试方法中主要是添加了一写关于新增 Aware 实现的调用，其他不需要调用的也打印了相应的日志信息，可以在测试结果中看到。

测试结果

```
执行：init-method
ClassLoader：sun.misc.Launcher$AppClassLoader@14dad5dc
Bean Name is：userService
测试结果：小傅哥,腾讯,深圳
ApplicationContextAware：
cn.bugstack.springframework.context.support.ClassPathXmlApplicationContext@5ba23b66
BeanFactoryAware：
cn.bugstack.springframework.beans.factory.support.DefaultListableBeanFactory@2ff4f0
0f
执行：destroy-method


Process finished with exit code 0
```

- 从测试结果可以看到，本章节新增加的感知接口对应的具体实现(BeanNameAware, BeanClassLoaderAware, ApplicationContextAware, BeanFactoryAware)，已经可以如期输出结果了。

六、总结

- 目前关于 Spring 框架的实现中，某些功能点已经越来趋向于完整，尤其是 Bean 对象的生命周期，已经有了很多的体现。整体总结如图 9-3

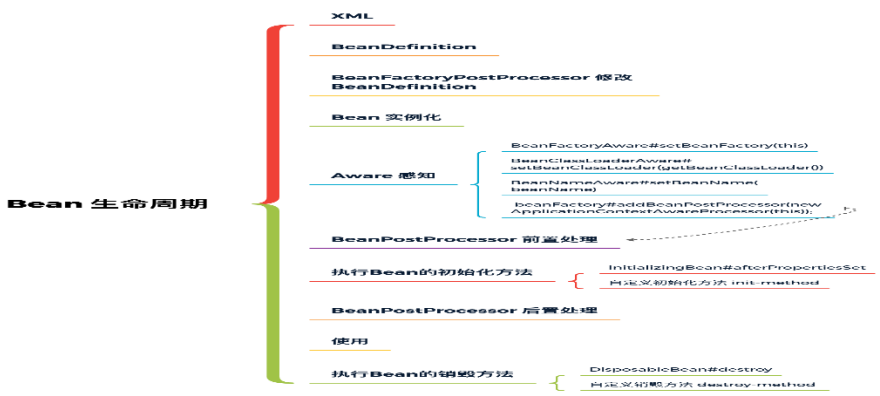


图 9-3

- 本章节关于 Aware 的感知接口的四个继承接口 BeanNameAware, BeanClassLoaderAware, ApplicationContextAware, BeanFactoryAware 的实现，又扩展了 Spring 的功能。如果你有做过关于 Spring 中间件的开发那么一定会大量用到这些类，现在你不只是用过，而且还知道他们都是什么时候触达的，在以后想排查类的实例化顺序也可以有一个清晰的思路了。
- 每一章节内容的实现都是在以设计模式为核心的结构上填充各项模块的功能，单纯的操作编写代码并不会有太多收获，一定是要理解为什么这么设计，这么设计的好处是什么，怎么就那么多接口和抽象类的应用，这些才是 Spring 框架学习的核心所在。

第 10 章：对象作用域和 FactoryBean

一、先见之明

老司机，你的砖怎么搬的那么快？

是有劲？是技巧？是后门？总之，那个老司机的代码总是可以很快的完成产品每次新增的需求，就像他俩是一家似的！而你就不一样了，不只产品经理还有运营、测试的小姐姐，都得给你买吃的，求着你赶紧把 Bug 修修，否则都来不及上线了。

那为啥别人的代码总是可以很快的扩展新功能，而你的代码从来不能被重构只能被重写，小需求小改、大需求大改，没需求呢？没需求自己跑着跑着也能崩溃，半夜被运维薅起来：“你这怎么又有数据库慢查询，把别人业务都拖拉胯了！”

有人说 30 岁的人都，还和刚毕业的做一样的活，是没进步的！ 这太扯淡了，同样是同样的活，但做出来的结果可不一定是一样的，有人能用 `ifelse` 把产品功能凑出来，也有人可以把需求拆解成各个功能模块，定义接口、抽象类、实现和继承，运用设计模式构建出一套新增需求时候能快速实现，出现问题能准确定位的代码逻辑。这就像有人问：“树上有十只鸟，一枪打死一只，还有几只？”，你会想到什么？枪声大吗、鸟笼了吗、鸟被绑树上了吗、有鸟残疾的吗、鸟被打死了吗、打鸟的人眼睛好使吗、算肚子里怀孕的鸟吗、打鸟犯法吗、边上树还有其他鸟吗等等，这些都是一个职业技术人在一个行业磨练出来的经验，不是 1 天 2 天看几本书，刷几个洗脑文章能吸收的。

二、目标

交给 Spring 管理的 Bean 对象，一定就是我们用类创建出来的 Bean 吗？创建出来的 Bean 就永远是单例的吗，没有可能是原型模式吗？

在集合 Spring 框架下，我们使用的 MyBatis 框架中，它的核心作用是可以满足用户不需要实现 Dao 接口类，就可以通过 xml 或者注解配置的方式完成对数据库执行 CRUD 操作，那么在实现这样的 ORM 框架中，是怎么把一个数据库

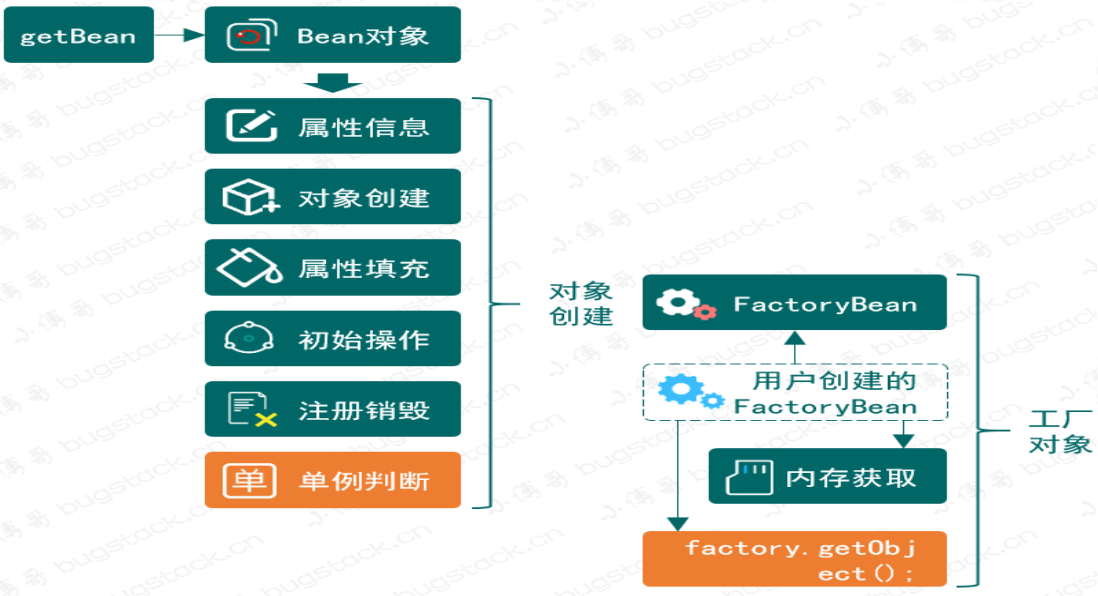
操作的 Bean 对象交给 Spring 管理的呢。

因为我们在使用 Spring、MyBatis 框架的时候都可以知道，并没有手动的去创建任何操作数据库的 Bean 对象，有的仅仅是一个接口定义，而这个接口定义竟然可以被注入到其他需要使用 Dao 的属性中去了，那么这一过程最核心待解决的问题，就是需要完成把复杂且以代理方式动态变化的对象，注册到 Spring 容器中。而为了满足这样的一个扩展组件开发的需求，就需要我们在现有手写的 Spring 框架中，添加这一能力。

三、方案

关于提供一个能让使用者定义复杂的 Bean 对象，功能点非常不错，意义也非常大，因为这样做了之后 Spring 的生态种子孵化箱就此提供了，谁家的框架都可以在此标准上完成自己服务的接入。

但这样的功能逻辑设计上并不复杂，因为整个 Spring 框架在开发的过程中就已经提供了各项扩展能力的接茬，你只需要在合适的位置提供一个接茬的处理接口调用和相应的功能逻辑实现即可，像这里的目标实现就是对外提供一个可以二次从 FactoryBean 的 getObject 方法中获取对象的功能即可，这样所有实现此接口的对象类，就可以扩充自己的对象功能了。*MyBatis 就是实现了一个 MapperFactoryBean 类，在 getObject 方法中提供 SqlSession 对执行 CRUD 方法的操作* 整体设计结构如下图：



- 整个的实现过程包括了两部分，一个解决单例还是原型对象，另外一个处理 FactoryBean 类型对象创建过程中关于获取具体调用对象的 `getObject` 操作。
- `SCOPE_SINGLETON`、`SCOPE_PROTOTYPE`，对象类型的创建获取方式，主要区分在于 `AbstractAutowireCapableBeanFactory#createBean` 创建完成对象后是否放入到内存中，如果不放入则每次获取都会重新创建。
- createBean 执行对象创建、属性填充、依赖加载、前置后置处理、初始化等操作后，就要开始做执行判断整个对象是否是一个 FactoryBean 对象，如果是这样的对象，就需要再继续执行获取 FactoryBean 具体对象中的 `getObject` 对象了。整个 getBean 过程中都会新增一个单例类型的判断 `factory.isSingleton()`，用于决定是否使用内存存放对象信息。

四、实现

1. 工程结构

```
small-spring-step-09
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   │   ├── BeanPostProcessor.java
    │           │   │   │   ├── BeanReference.java
    │           │   │   │   ├── ConfigurableBeanFactory.java
    │           │   │   │   └── SingletonBeanRegistry.java
    │           │   │   ├── support
    │           │   │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   │   ├── AbstractBeanFactory.java
    │           │   │   │   ├── BeanDefinitionReader.java
    │           │   │   │   ├── BeanDefinitionRegistry.java
    │           │   │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   │   ├── DefaultListableBeanFactory.java
    │           │   │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   │   ├── DisposableBeanAdapter.java
    │           │   │   │   ├── FactoryBeanRegistrySupport.java
    │           │   │   │   ├── InstantiationStrategy.java
```

```
    │           │   │   │   └── SimpleInstantiationStrategy.java
    │           │   │   ├── support
    │           │   │   │   └── XmlBeanDefinitionReader.java
    │           │   │   ├── Aware.java
    │           │   │   ├── BeanClassLoaderAware.java
    │           │   │   ├── BeanFactory.java
    │           │   │   ├── BeanFactoryAware.java
    │           │   │   ├── BeanNameAware.java
    │           │   │   ├── ConfigurableListableBeanFactory.java
    │           │   │   ├── DisposableBean.java
    │           │   │   ├── FactoryBean.java
    │           │   │   ├── HierarchicalBeanFactory.java
    │           │   │   ├── InitializingBean.java
    │           │   │   └── ListableBeanFactory.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
    │           │   └── PropertyValues.java
    │           ├── context
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   ├── ApplicationContextAwareProcessor.java
    │           │   │   └── ClassPathXmlApplicationContext.java
    │           │   ├── ApplicationContext.java
    │           │   ├── ApplicationContextAware.java
    │           │   └── ConfigurableApplicationContext.java
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           └── utils
    │               └── ClassUtils.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── UserDao.java
                │   └── UserService.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

Spring 单例、原型以及 FactoryBean 功能实现类关系，如图 10-2

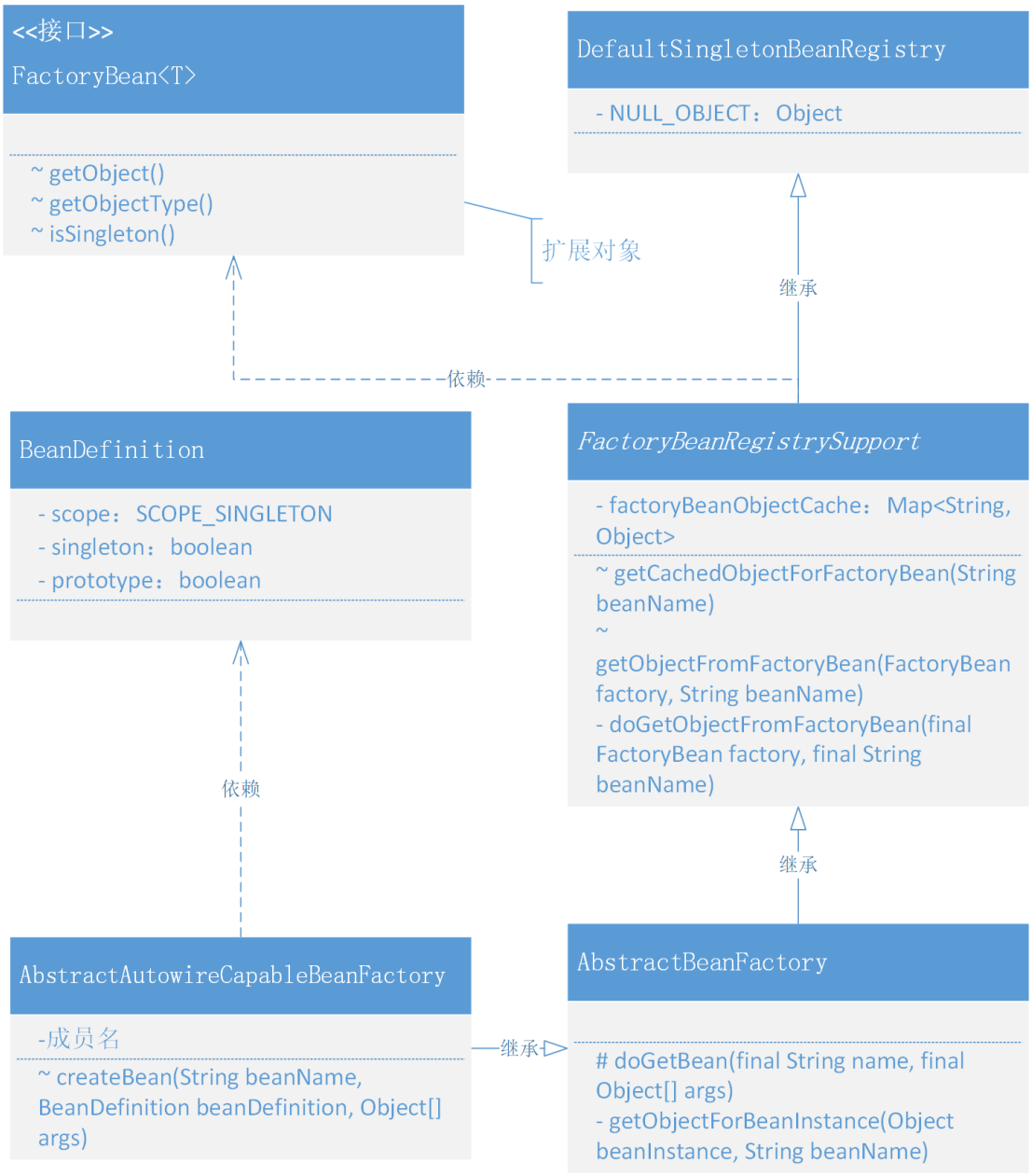


图 10-2

- 以上整个类关系图展示的就是添加 Bean 的实例化是单例还是原型模式以及 FactoryBean 的实现。
- 其实整个实现的过程并不复杂，只是在现有的 AbstractAutowireCapableBeanFactory 类以及继承的抽象类 AbstractBeanFactory 中进行扩展。
- 不过这次我们把 AbstractBeanFactory 继承的 DefaultSingletonBeanRegistry 类，中间加了一层 FactoryBeanRegistrySupport，这个类在 Spring 框架中主要是处理关于 FactoryBean 注册的支撑操作。

2. Bean 的作用范围定义和 xml 解析

cn.bugstack.springframework.beans.factory.config.BeanDefinition

```
public class BeanDefinition {

    String SCOPE_SINGLETON = ConfigurableBeanFactory.SCOPE_SINGLETON;

    String SCOPE_PROTOTYPE = ConfigurableBeanFactory.SCOPE_PROTOTYPE;

    private Class beanClass;

    private PropertyValues propertyValues;

    private String initMethodName;

    private String destroyMethodName;

    private String scope = SCOPE_SINGLETON;

    private boolean singleton = true;

    private boolean prototype = false;

    // ...get/set
}
```

- singleton、prototype，是本次在 BeanDefinition 类中新增加的两个属性信息，用于把从 spring.xml 中解析到的 Bean 对象作用范围填充到属性中。

cn.bugstack.springframework.beans.factory.xml.XmlBeanDefinitionReader

```
public class XmlBeanDefinitionReader extends AbstractBeanDefinitionReader {

    protected void doLoadBeanDefinitions(InputStream inputStream) throws ClassNotFoundException {

        for (int i = 0; i < childNodes.getLength(); i++) {
            // 判断元素
            if (!(childNodes.item(i) instanceof Element)) continue;
            // 判断对象
            if (!"bean".equals(childNodes.item(i).getNodeName())) continue;
```

```
            // 解析标签
            Element bean = (Element) childNodes.item(i);
            String id = bean.getAttribute("id");
            String name = bean.getAttribute("name");
            String className = bean.getAttribute("class");
            String initMethod = bean.getAttribute("init-method");
            String destroyMethodName = bean.getAttribute("destroy-method");
            String beanScope = bean.getAttribute("scope");

            // 获取 Class，方便获取类中的名称
            Class<?> clazz = Class.forName(className);
            // 优先级 id > name
            String beanName = StrUtil.isNotEmpty(id) ? id : name;
            if (StrUtil.isEmpty(beanName)) {
                beanName = StrUtil.lowerFirst(clazz.getSimpleName());
            }

            // 定义Bean
            BeanDefinition beanDefinition = new BeanDefinition(clazz);
            beanDefinition.setInitMethodName(initMethod);
            beanDefinition.setDestroyMethodName(destroyMethodName);

            if (StrUtil.isNotEmpty(beanScope)) {
                beanDefinition.setScope(beanScope);
            }

            // ...

            // 注册 BeanDefinition
            getRegistry().registerBeanDefinition(beanName, beanDefinition);
        }
    }

}
```

- 在解析 XML 处理类 XmlBeanDefinitionReader 中，新增加了关于 Bean 对象配置中 scope 的解析，并把这个属性信息填充到 Bean 定义中。`beanDefinition.setScope(beanScope)`

3. 创建和修改对象时候判断单例和原型模式

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactor
y implements AutowireCapableBeanFactory {

    private InstantiationStrategy instantiationStrategy = new CglibSubclassingInsta
ntiationStrategy();

    @Override
    protected Object createBean(String beanName, BeanDefinition beanDefinition, Obj
ect[] args) throws BeansException {
        Object bean = null;
        try {
            bean = createBeanInstance(beanDefinition, beanName, args);
            // 给 Bean 填充属性
            applyPropertyValues(beanName, bean, beanDefinition);
            // 执行 Bean 的初始化方法和 BeanPostProcessor 的前置和后置处理方法
            bean = initializeBean(beanName, bean, beanDefinition);
        } catch (Exception e) {
            throw new BeansException("Instantiation of bean failed", e);
        }

        // 注册实现了 DisposableBean 接口的 Bean 对象
        registerDisposableBeanIfNecessary(beanName, bean, beanDefinition);

        // 判断 SCOPE_SINGLETON、SCOPE_PROTOTYPE
        if (beanDefinition.isSingleton()) {
            addSingleton(beanName, bean);
        }
        return bean;
    }

    protected void registerDisposableBeanIfNecessary(String beanName, Object bean,
BeanDefinition beanDefinition) {
        // 非 Singleton 类型的 Bean 不执行销毁方法
        if (!beanDefinition.isSingleton()) return;

        if (bean instanceof DisposableBean || StrUtil.isNotEmpty(beanDefinition.get
DestroyMethodName())) {
            registerDisposableBean(beanName, new DisposableBeanAdapter(bean, beanNa
me, beanDefinition));
        }
    }

    // ... 其他功能
}
```

- 单例模式和原型模式的区别就在于是否存放到内存中，如果是原型模式那么就不会存放到内存中，每次获取都重新创建对象，另外非 Singleton 类型的 Bean 不需要执行销毁方法。
- 所以这里的代码会有两处修改，一处是 createBean 中判断是否添加到 addSingleton(beanName, bean);，另外一处是 registerDisposableBeanIfNecessary 销毁注册中的判断 `if (!beanDefinition.isSingleton()) return;`。

4. 定义 FactoryBean 接口

cn.bugstack.springframework.beans.factory.FactoryBean

```
public interface FactoryBean<T> {

    T getObject() throws Exception;

    Class<?> getObjectType();

    boolean isSingleton();

}
```

- FactoryBean 中需要提供 3 个方法，获取对象、对象类型，以及是否是单例对象，如果是单例对象依然会被放到内存中。

5. 实现一个 FactoryBean 注册服务

cn.bugstack.springframework.beans.factory.support.FactoryBeanRegistrySupport

```
public abstract class FactoryBeanRegistrySupport extends DefaultSingletonBeanRegistry {

    /**
     * Cache of singleton objects created by FactoryBeans: FactoryBean name -> object
     */
    private final Map<String, Object> factoryBeanObjectCache = new ConcurrentHashMap<String, Object>();

    protected Object getCachedObjectForFactoryBean(String beanName) {
        Object object = this.factoryBeanObjectCache.get(beanName);
        return (object != NULL_OBJECT ? object : null);
```

```
    }

    protected Object getObjectFromFactoryBean(FactoryBean factory, String beanName)
{
        if (factory.isSingleton()) {
            Object object = this.factoryBeanObjectCache.get(beanName);
            if (object == null) {
                object = doGetObjectFromFactoryBean(factory, beanName);
                this.factoryBeanObjectCache.put(beanName, (object != null ? object
: NULL_OBJECT));
            }
            return (object != NULL_OBJECT ? object : null);
        } else {
            return doGetObjectFromFactoryBean(factory, beanName);
        }
    }

    private Object doGetObjectFromFactoryBean(final FactoryBean factory, final Stri
ng beanName){
        try {
            return factory.getObject();
        } catch (Exception e) {
            throw new BeansException("FactoryBean threw exception on object[" + bea
nName + "] creation", e);
        }
    }

}
```

- FactoryBeanRegistrySupport 类主要处理的就是关于 FactoryBean 此类对象的注册操作，之所以放到这样一个单独的类里，就是希望做到不同领域模块下只负责各自需要完成的功能，避免因为扩展导致类膨胀到难以维护。
- 同样这里也定义了缓存操作 factoryBeanObjectCache，用于存放单例类型的对象，避免重复创建。*在日常使用用，基本也都是创建的单例对象*
- doGetObjectFromFactoryBean 是具体的获取 FactoryBean#getObject() 方法，因为既有缓存的处理也有对象的获取，所以额外提供了 getObjectFromFactoryBean 进行逻辑包装，这部分的操作方式是不和你日常做的业务逻辑开发非常相似。*从 Redis 取数据，如果为空就从数据库获取并写入 Redis*

6. 扩展 AbstractBeanFactory 创建对象逻辑

cn.bugstack.springframework.beans.factory.support.AbstractBeanFactory

```
public abstract class AbstractBeanFactory extends FactoryBeanRegistrySupport implem
ents ConfigurableBeanFactory {

    protected <T> T doGetBean(final String name, final Object[] args) {
        Object sharedInstance = getSingleton(name);
        if (sharedInstance != null) {
            // 如果是 FactoryBean，则需要调用 FactoryBean#getObject
            return (T) getObjectForBeanInstance(sharedInstance, name);
        }

        BeanDefinition beanDefinition = getBeanDefinition(name);
        Object bean = createBean(name, beanDefinition, args);
        return (T) getObjectForBeanInstance(bean, name);
    }

    private Object getObjectForBeanInstance(Object beanInstance, String beanName) {
        if (!(beanInstance instanceof FactoryBean)) {
            return beanInstance;
        }

        Object object = getCachedObjectForFactoryBean(beanName);

        if (object == null) {
            FactoryBean<?> factoryBean = (FactoryBean<?>) beanInstance;
            object = getObjectFromFactoryBean(factoryBean, beanName);
        }

        return object;
    }

    // ...
}
```

- 首先这里把 AbstractBeanFactory 原来继承的 DefaultSingletonBeanRegistry，修改为继承 FactoryBeanRegistrySupport。因为需要扩展出创建 FactoryBean 对象的能力，所以这就想一个链条服务上，截出一个段来处理额外的服务，并把链条再链接上。
- 此处新增加的功能主要是在 doGetBean 方法中，添加了调用 `(T) getObjectForBeanInstance(sharedInstance, name)` 对获取 FactoryBean 的操作。
- 在 getObjectForBeanInstance 方法中做具体的 instanceof 判断，另外还会从 FactoryBean 的缓存中获取对象，如果不存在则调用 FactoryBeanRegistrySupport#getObjectFromFactoryBean，执行具体的操作。

五、测试

1. 事先准备

cn.bugstack.springframework.test.bean.IUserDao

```
public interface IUserDao {

    String queryUserName(String uId);

}
```

- 这个章节我们删掉 UserDao，定义一个 IUserDao 接口，之所这样做是为了通过 FactoryBean 做一个自定义对象的代理操作。

cn.bugstack.springframework.test.bean.UserService

```
public class UserService {

    private String uId;
    private String company;
    private String location;
    private IUserDao userDao;

    public String queryUserInfo() {
        return userDao.queryUserName(uId) + "," + company + "," + location;
    }

    // ...get/set
}
```

- 在 UserService 新修改了一个原有 UserDao 属性为 IUserDao，后面我们会给这个属性注入代理对象。

2. 定义 FactoryBean 对象

cn.bugstack.springframework.test.bean.ProxyBeanFactory

```
public class ProxyBeanFactory implements FactoryBean<IUserDao> {

    @Override
    public IUserDao getObject() throws Exception {
```

```
        InvocationHandler handler = (proxy, method, args) -> {

            Map<String, String> hashMap = new HashMap<>();
            hashMap.put("10001", "小傅哥");
            hashMap.put("10002", "八杯水");
            hashMap.put("10003", "阿毛");

            return "你被代理了 " + method.getName() + ": 
" + hashMap.get(args[0].toString());
        };
        return (IUserDao) Proxy.newProxyInstance(Thread.currentThread().getContextC
lassLoader(), new Class[]{IUserDao.class}, handler);
    }

    @Override
    public Class<?> getObjectType() {
        return IUserDao.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }

}
```

- 这是一个实现接口 FactoryBean 的代理类 ProxyBeanFactory 名称，主要是模拟了 UserDao 的原有功能，类似于 MyBatis 框架中的代理操作。
- getObject() 中提供的就是一个 InvocationHandler 的代理对象，当有方法调用的时候，则执行代理对象的功能。

3. 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>

    <bean id="userService" class="cn.bugstack.springframework.test.bean.UserService
" scope="prototype">
        <property name="uId" value="10001"/>
        <property name="company" value="腾讯"/>
        <property name="location" value="深圳"/>
        <property name="userDao" ref="proxyUserDao"/>
    </bean>
```

```
    <bean id="proxyUserDao" class="cn.bugstack.springframework.test.bean.ProxyBeanF
actory"/>

</beans>
```

- 在配置文件中，我们把 proxyUserDao 这个代理对象，注入到 userService 的 userDao 中。*与上一章节相比，去掉了 UserDao 实现类，转而用代理类替换*

4. 单元测试(单例&原型)

```
@Test
public void test_prototype() {
    // 1.初始化 BeanFactory
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplication
Context("classpath:spring.xml");
    applicationContext.registerShutdownHook();

    // 2. 获取 Bean 对象调用方法
    UserService userService01 = applicationContext.getBean("userService", UserServi
ce.class);
    UserService userService02 = applicationContext.getBean("userService", UserServi
ce.class);

    // 3. 配置 scope="prototype/singleton"
    System.out.println(userService01);
    System.out.println(userService02);

    // 4. 打印十六进制哈希
    System.out.println(userService01 + " 十六进制哈希：
" + Integer.toHexString(userService01.hashCode()));
    System.out.println(ClassLayout.parseInstance(userService01).toPrintable());

}
```

- 在 spring.xml 配置文件中，设置了 scope="prototype" 这样就每次获取到的对象都应该是一个新的对象。
- 这里判断对象是否为一个会看到打印的类对象的哈希值，所以我们把十六进制哈希打印出来。

测试结果

```
cn.bugstack.springframework.test.bean.UserService$$EnhancerByCGLIB$$4cabb984@1b0375
b3
```

```
cn.bugstack.springframework.test.bean.UserService$$EnhancerByCGLIB$$4cabb984@2f7c72
60
cn.bugstack.springframework.test.bean.UserService$$EnhancerByCGLIB$$4cabb984@1b0375
b3 十六进制哈希：1b0375b3
cn.bugstack.springframework.test.bean.UserService$$EnhancerByCGLIB$$4cabb984 object
 internals:
 OFFSET  SIZE                                               TYPE DESCRIPTION
                                    VALUE
      0     4                                                    (object header)
                                    01 b3 75 03 (00000001 10110011 01110101 00000
011) (58045185)
      4     4                                                    (object header)
                                    1b 00 00 00 (00011011 00000000 00000000 00000
000) (27)
      8     4                                                    (object header)
                                    9f e1 01 f8 (10011111 11100001 00000001 11111
000) (-134094433)
     12     4                                    java.lang.String UserService.uId
                                    (object)
     16     4                                    java.lang.String UserService.company
                                    (object)
     20     4                                    java.lang.String UserService.location
                                    (object)
     24     4   cn.bugstack.springframework.test.bean.IUserDao UserService.userDao
                                    (object)
     28     1                                            boolean UserService$$Enhance
rByCGLIB$$4cabb984.CGLIB$BOUND      true
     29     3                                                    (alignment/padding g
ap)
     32     4                         net.sf.cglib.proxy.NoOp UserService$$Enhance
rByCGLIB$$4cabb984.CGLIB$CALLBACK_0 (object)
     36     4                                                    (loss due to the nex
t object alignment)
Instance size: 40 bytes
Space losses: 3 bytes internal + 4 bytes external = 7 bytes total


Process finished with exit code 0
```

```
cn.bugstack.springframework.test.bean.UserService$$EnhancerByCGLIB$$4cabb984@1b0375b3 十六进制哈
cn.bugstack.springframework.test.bean.UserService$$EnhancerByCGLIB$$4cabb984 object internals:
 OFFSET  SIZE                                      TYPE DESCRIPTION
      0     4                                           (object header)
      4     4                                           (object header)
      8     4                                           (object header)
```

- 对象后面的这一小段字符串就是 16 进制哈希值，在对象头哈希值存放的结果上看，也有对应的数值。只不过这个结果是倒过来的。
- 另外可以看到 cabb984@1b0375b3、cabb984@2f7c7260，这两个对象的结尾 16 进制哈希值并不一样，所以我们的原型模式是生效的。

5. 单元测试(代理对象)

```
@Test
public void test_factory_bean() {
    // 1.初始化 BeanFactory
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplication
Context("classpath:spring.xml");
    applicationContext.registerShutdownHook();

    // 2. 调用代理方法
    UserService userService = applicationContext.getBean("userService", UserService
.class);
    System.out.println("测试结果：" + userService.queryUserInfo());
}
```

- 关于 FactoryBean 的调用并没有太多不一样，因为所有的不同都已经被 spring.xml 配置进去了。当然你可以直接调用 spring.xml 配置的对象 `cn.bugstack.springframework.test.bean.ProxyBeanFactory`

测试结果

```
测试结果：你被代理了 queryUserName：小傅哥,腾讯,深圳

Process finished with exit code 0
```

- 从测试结果来看，我们的代理类 ProxyBeanFactory 已经完美替换掉了 UserDao 的功能。
- 虽然看上去这一点实现并不复杂，甚至有点简单。但就是这样一点点核心内容的设计了，解决了所有需要和 Spring 结合的其他框架交互链接问题。*如果对此类内容感兴趣，也可以阅读小傅哥《中间件设计和开发》*

六、总结

- 在 Spring 框架整个开发的过程中，前期的各个功能接口类扩展的像膨胀了似的，但到后期在完善功能时，就没有那么难了，反而深入理解后会觉得功能的补充，都比较简单。只需要再所遇领域范围内进行扩展相应的服务实现即可。
- 当你仔细阅读完关于 FactoryBean 的实现以及测试过程的使用，以后再需要使用 FactoryBean 开发相应的组件时候，一定会非常清楚它是如何创建自己的复杂 Bean 对象以及在什么时候初始化和调用的。遇到问题也可以快速的排查、定位和解决。
- 如果你在学习的过程中感觉这些类、接口、实现、继承，穿梭的很复杂，一时半会脑子还反应不过来。那么你最好的方式是动手去画画这些类关系图，梳理下实现的结构，看看每个类在干什么。*看只能是知道，动手才能学会！*

第 11 章：容器事件和事件监听器

一、降低耦合

能解耦，是多么重要的一件事情！

摔杯为号、看我眼色行事、见南面火起，这是在嘎哈么？这其实是在通过事物传播进行解耦引线和炸弹，仅仅是这样的一个解耦，它放到了多少村夫莽汉，劫了法场，篡了兵权！

这样的解耦场景在互联网开发的设计中使用的也是非常频繁，如：这里需要一个注册完成事件推送消息、用户下单我会发送一个 MQ、收到我的支付消息就可以发货了等等，都是依靠事件订阅和发布以及 MQ 消息这样的组件，来处理系统之间的调用解耦，最终通过解耦的方式来提升整体系统架构的负载能力。

其实解耦思路可以理解为设计模式中观察者模式的具体使用效果，在观察者模式中当对象间存在一对多关系时，则使用观察者模式，它是一种定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。*这让我想起了我每个月的车牌摇号，都会推送给我一条本月没中签的消息！！！*

二、目标

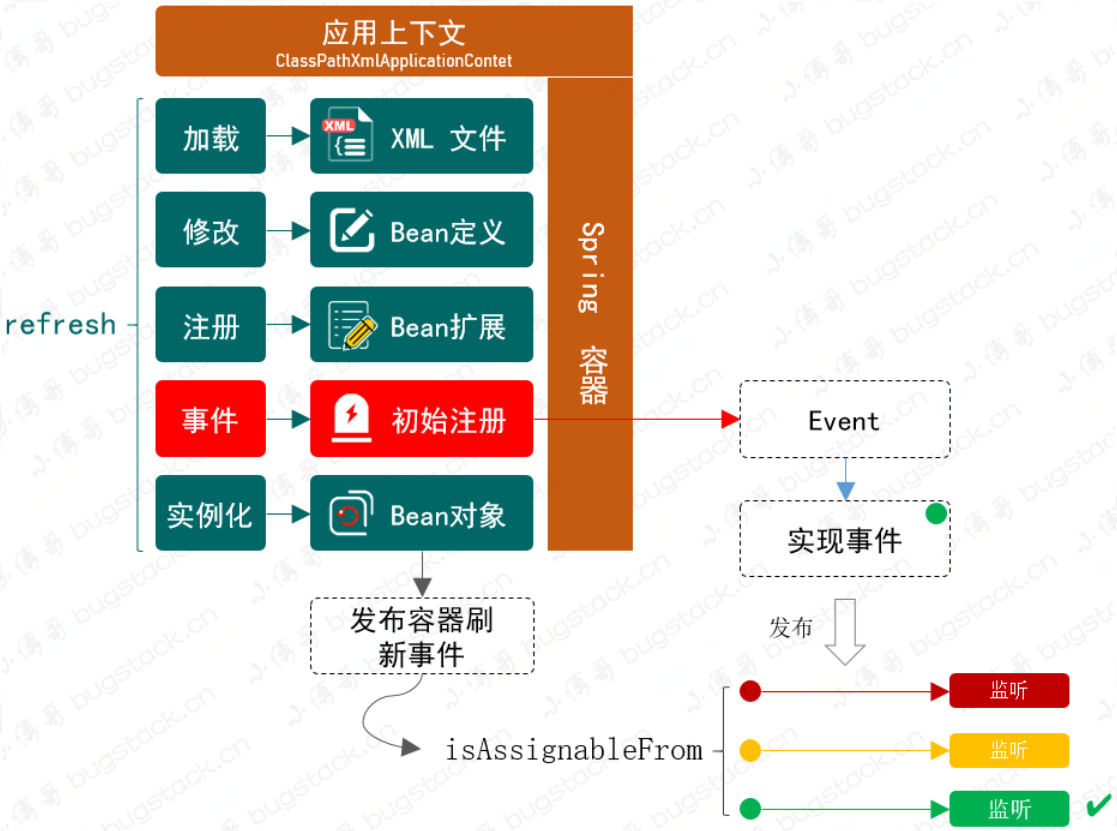
在 Spring 中有一个 Event 事件功能，它可以提供事件的定义、发布以及监听事件来完成一些自定义的动作。比如你可以定义一个新用户注册的事件，当有用户执行注册完成后，在事件监听中给用户发送一些优惠券和短信提醒，这样的操作就可以把属于基本功能的注册和对应的策略服务分开，降低系统的耦合。以后在扩展注册服务，比如需要添加风控策略、添加实名认证、判断用户属性等都不会影响到依赖注册成功后执行的动作。

那么在本章节我们需要以观察者模式的方式，设计和实现 Spring Event 的容器事件和事件监听器功能，最终可以让我们在现有实现的 Spring 框架中可以定义、监听和发布自己的事件信息。

三、方案

其实事件的设计本身就是一种观察者模式的实现，它所要解决的就是一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

在功能实现上我们需要定义出事件类、事件监听、事件发布，而这些类的功能需要结合到 Spring 的 AbstractApplicationContext#refresh()，以便于处理事件初始化和注册事件监听器的操作。整体设计结构如下图：



- 在整个功能实现过程中，仍然需要在面向用户的应用上下文 `AbstractApplicationContext` 中添加相关事件内容，包括：初始化事件发布者、注册事件监听器、发布容器刷新完成事件。
- 使用观察者模式定义事件类、监听类、发布类，同时还需要完成一个广播器的功能，接收到事件推送时进行分析处理符合监听事件接受者感兴趣的事件，也就是使用 isAssignableFrom 进行判断。
- isAssignableFrom 和 instanceof 相似，不过 isAssignableFrom 是用来判断子类和父类的关系的，或者接口的实现类和接口的关系的，默认所有的类的终极父类都是 Object。如果 A.isAssignableFrom(B)结果是 true，证明 B 可以转换成为 A,也就是 A 可以由 B 转换而来。

四、实现

1. 工程结构

```
small-spring-step-10
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   │   ├── BeanPostProcessor.java
    │           │   │   │   ├── BeanReference.java
    │           │   │   │   ├── ConfigurableBeanFactory.java
    │           │   │   │   └── SingletonBeanRegistry.java
    │           │   │   ├── support
    │           │   │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   │   ├── AbstractBeanFactory.java
    │           │   │   │   ├── BeanDefinitionReader.java
    │           │   │   │   ├── BeanDefinitionRegistry.java
    │           │   │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   │   ├── DefaultListableBeanFactory.java
    │           │   │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   │   ├── DisposableBeanAdapter.java
    │           │   │   │   ├── FactoryBeanRegistrySupport.java
    │           │   │   │   ├── InstantiationStrategy.java
    │           │   │   │   └── SimpleInstantiationStrategy.java
    │           │   │   ├── support
    │           │   │   │   └── XmlBeanDefinitionReader.java
    │           │   │   ├── Aware.java
    │           │   │   ├── BeanClassLoaderAware.java
    │           │   │   ├── BeanFactory.java
    │           │   │   ├── BeanFactoryAware.java
    │           │   │   ├── BeanNameAware.java
    │           │   │   ├── ConfigurableListableBeanFactory.java
    │           │   │   ├── DisposableBean.java
    │           │   │   ├── FactoryBean.java
```

```
    │           │   │   ├── HierarchicalBeanFactory.java
    │           │   │   ├── InitializingBean.java
    │           │   │   └── ListableBeanFactory.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
    │           │   └── PropertyValues.java
    │           ├── context
    │           │   ├── event
    │           │   │   ├── AbstractApplicationEventMulticaster.java
    │           │   │   ├── ApplicationContextEvent.java
    │           │   │   ├── ApplicationEventMulticaster.java
    │           │   │   ├── ContextClosedEvent.java
    │           │   │   ├── ContextRefreshedEvent.java
    │           │   │   └── SimpleApplicationEventMulticaster.java
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   ├── ApplicationContextAwareProcessor.java
    │           │   │   └── ClassPathXmlApplicationContext.java
    │           │   ├── ApplicationContext.java
    │           │   ├── ApplicationContextAware.java
    │           │   ├── ApplicationEvent.java
    │           │   ├── ApplicationEventPublisher.java
    │           │   ├── ApplicationListener.java
    │           │   └── ConfigurableApplicationContext.java
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           └── utils
    │               └── ClassUtils.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── event
                │   ├── ContextClosedEventListener.java
                │   ├── ContextRefreshedEventListener.java
                │   ├── CustomEvent.java
                │   └── CustomEventListener.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

容器事件和事件监听器实现类关系，如图 11-2

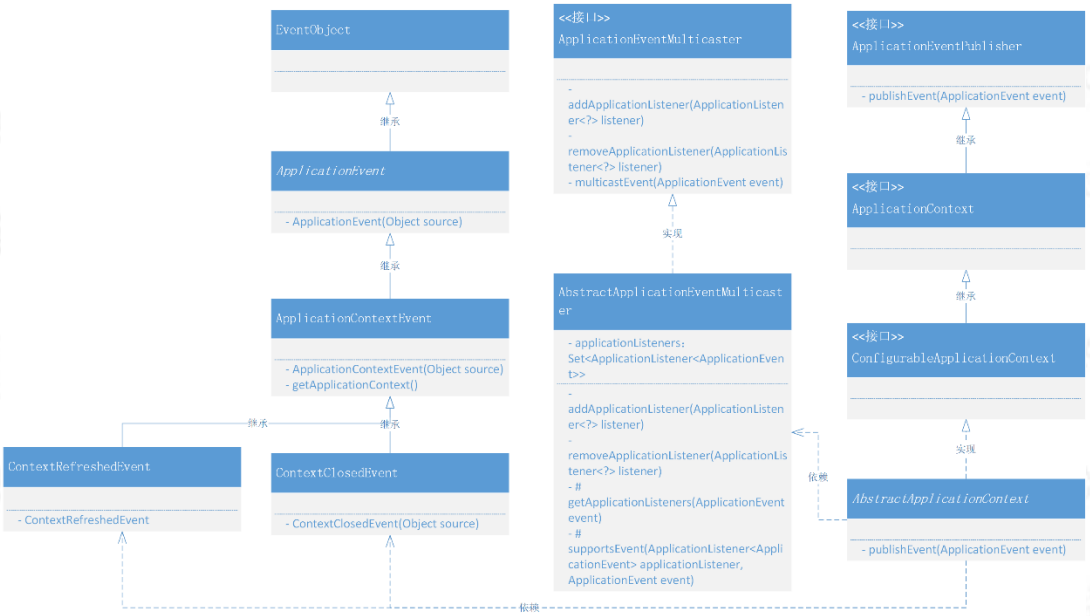


图 10-2

- 以上整个类关系图以围绕实现 event 事件定义、发布、监听功能实现和把事件的相关内容使用 AbstractApplicationContext#refresh 进行注册和处理操作。
- 在实现的过程中主要以扩展 spring context 包为主，事件的实现也是在这个包下进行扩展的，当然也可以看出来目前所有的实现内容，仍然是以 IOC 为主。
- ApplicationContext 容器继承事件发布功能接口 ApplicationEventPublisher，并在实现类中提供事件监听功能。
- ApplicationEventMulticaster 接口是注册监听器和发布事件的广播器，提供添加、移除和发布事件方法。
- 最后是发布容器关闭事件，这个仍然需要扩展到 AbstractApplicationContext#close 方法中，由注册到虚拟机的钩子实现。

2. 定义和实现事件

cn.bugstack.springframework.context.ApplicationEvent

```
public abstract class ApplicationEvent extends EventObject {

    /**
     * Constructs a prototypical Event.
     *
     * @param source The object on which the Event initially occurred.
     * @throws IllegalArgumentException if source is null.
     */
    public ApplicationEvent(Object source) {
```

```
        super(source);
    }

}
```

- 以继承 java.util.EventObject 定义出具备事件功能的抽象类 ApplicationEvent，后续所有事件的类都需要继承这个类。

cn.bugstack.springframework.context.event.ApplicationContextEvent

```
public class ApplicationContextEvent extends ApplicationEvent {

    /**
     * Constructs a prototypical Event.
     *
     * @param source The object on which the Event initially occurred.
     * @throws IllegalArgumentException if source is null.
     */
    public ApplicationContextEvent(Object source) {
        super(source);
    }

    /**
     * Get the <code>ApplicationContext</code> that the event was raised for.
     */
    public final ApplicationContext getApplicationContext() {
        return (ApplicationContext) getSource();
    }

}
```

cn.bugstack.springframework.context.event.ContextClosedEvent

```
public class ContextClosedEvent extends ApplicationContextEvent{

    /**
     * Constructs a prototypical Event.
     *
     * @param source The object on which the Event initially occurred.
     * @throws IllegalArgumentException if source is null.
     */
    public ContextClosedEvent(Object source) {
        super(source);
    }

}
```

cn.bugstack.springframework.context.event.ContextRefreshedEvent

```
public class ContextRefreshedEvent extends ApplicationContextEvent{
    /**
     * Constructs a prototypical Event.
     *
     * @param source The object on which the Event initially occurred.
     * @throws IllegalArgumentException if source is null.
     */
    public ContextRefreshedEvent(Object source) {
        super(source);
    }

}
```

- ApplicationContextEvent 是定义事件的抽象类，所有的事件包括关闭、刷新，以及用户自己实现的事件，都需要继承这个类。
- ContextClosedEvent、ContextRefreshedEvent，分别是 Spring 框架自己实现的两个事件类，可以用于监听刷新和关闭动作。

3. 事件广播器

cn.bugstack.springframework.context.event.ApplicationEventMulticaster

```
public interface ApplicationEventMulticaster {

    /**
     * Add a listener to be notified of all events.
     * @param listener the listener to add
     */
    void addApplicationListener(ApplicationListener<?> listener);

    /**
     * Remove a listener from the notification list.
     * @param listener the listener to remove
     */
    void removeApplicationListener(ApplicationListener<?> listener);

    /**
     * Multicast the given application event to appropriate listeners.
     * @param event the event to multicast
     */
    void multicastEvent(ApplicationEvent event);
```

```
}
```

- 在事件广播器中定义了添加监听和删除监听的方法以及一个广播事件的方法 `multicastEvent` 最终推送时间消息也会经过这个接口方法来处理谁该接收事件。

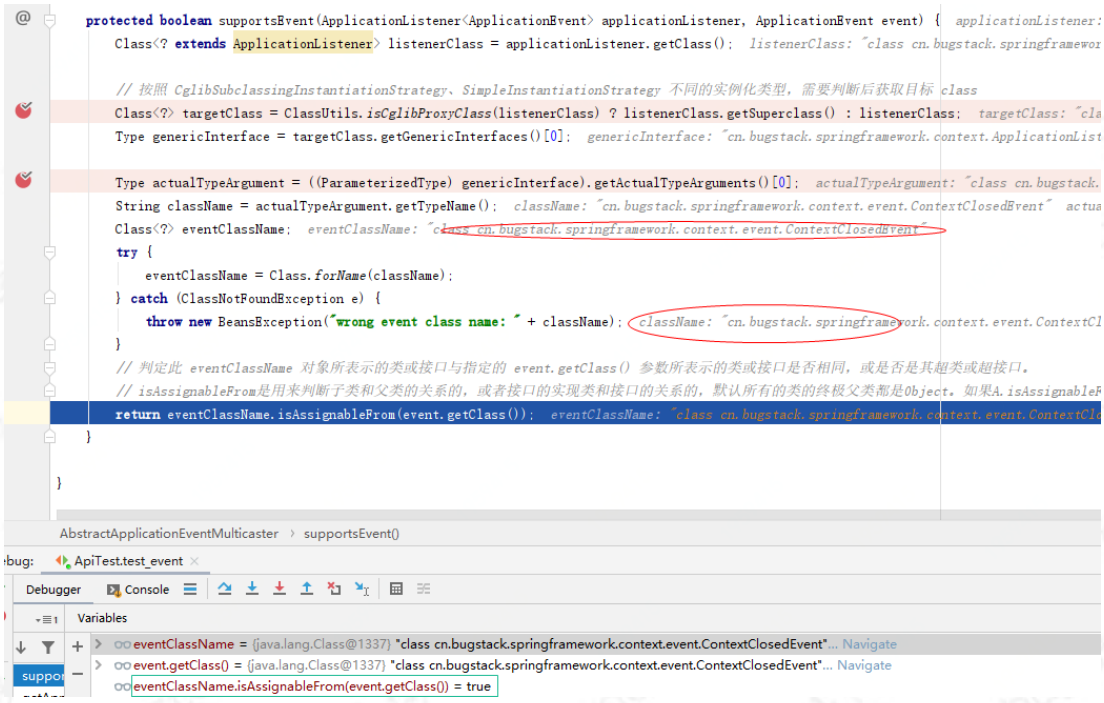
cn.bugstack.springframework.context.event.AbstractApplicationEventMulticaster

```
public abstract class AbstractApplicationEventMulticaster implements ApplicationEventMulticaster, BeanFactoryAware {

    public final Set<ApplicationListener<ApplicationEvent>> applicationListeners = new LinkedHashSet<>();

    private BeanFactory beanFactory;

    @Override
    public void addApplicationListener(ApplicationListener<?> listener) {
        applicationListeners.add((ApplicationListener<ApplicationEvent>) listener);
    }

    @Override
    public void removeApplicationListener(ApplicationListener<?> listener) {
        applicationListeners.remove(listener);
    }

    @Override
    public final void setBeanFactory(BeanFactory beanFactory) {
        this.beanFactory = beanFactory;
    }

    protected Collection<ApplicationListener> getApplicationListeners(ApplicationEvent event) {
        LinkedList<ApplicationListener> allListeners = new LinkedList<ApplicationListener>();
        for (ApplicationListener<ApplicationEvent> listener : applicationListeners) {
            if (supportsEvent(listener, event)) allListeners.add(listener);
        }
        return allListeners;
    }
```

```
    /**
     * 监听器是否对该事件感兴趣
     */
    protected boolean supportsEvent(ApplicationListener<ApplicationEvent> applicationListener, ApplicationEvent event) {
        Class<? extends ApplicationListener> listenerClass = applicationListener.getClass();

        // 按照 CglibSubclassingInstantiationStrategy、SimpleInstantiationStrategy 不同的实例化类型，需要判断后获取目标 class
        Class<?> targetClass = ClassUtils.isCglibProxyClass(listenerClass) ? listenerClass.getSuperclass() : listenerClass;
        Type genericInterface = targetClass.getGenericInterfaces()[0];

        Type actualTypeArgument = ((ParameterizedType) genericInterface).getActualTypeArguments()[0];
        String className = actualTypeArgument.getTypeName();
        Class<?> eventClassName;
        try {
            eventClassName = Class.forName(className);
        } catch (ClassNotFoundException e) {
            throw new BeansException("wrong event class name: " + className);
        }
        // 判定此 eventClassName 对象所表示的类或接口与指定的 event.getClass() 参数所表示的类或接口是否相同，或是否是其超类或超接口。
        // isAssignableFrom 是用来判断子类和父类的关系的，或者接口的实现类和接口的关系的，默认所有的类的终极父类都是 Object。如果 A.isAssignableFrom(B) 结果是 true，证明 B 可以转换成为 A，也就是 A 可以由 B 转换而来。
        return eventClassName.isAssignableFrom(event.getClass());
    }

}
```

- AbstractApplicationEventMulticaster 是对事件广播器的公用方法提取，在这个类中可以实现一些基本功能，避免所有直接实现接口放还需要处理细节。
- 除了像 addApplicationListener、removeApplicationListener，这样的通用方法，这里这个类中主要是对 getApplicationListeners 和 supportsEvent 的处理。
- getApplicationListeners 方法主要是摘取符合广播事件中的监听处理器，具体过滤动作在 supportsEvent 方法中。
- 在 supportsEvent 方法中，主要包括对 Cglib、Simple 不同实例化需要获取目标 Class，Cglib 代理类需要获取父类的 Class，普通实例化的不需要。接下来就是通过提取接口和对应的 ParameterizedType 和 eventClassName，方便最后确认是否为子类和父类的关系，以此证明此事件归这个符合的类处理。*可以参考代码中的注释*

supportsEvent 方法运行截图



- 在代码调试中可以看到，最终 eventClassName 和 event.getClass() 在 isAssignableFrom 判断下为 true
- 关于 CglibSubclassingInstantiationStrategy、SimpleInstantiationStrategy 可以尝试在 AbstractApplicationContext 类中更换验证。

4. 事件发布者的定义和实现

cn.bugstack.springframework.context.ApplicationEventPublisher

```
public interface ApplicationEventPublisher {

    /**
     * Notify all listeners registered with this application of an application
     * event. Events may be framework events (such as RequestHandledEvent)
     * or application-specific events.
     * @param event the event to publish
     */
    void publishEvent(ApplicationEvent event);

}
```

- ApplicationEventPublisher 是整个一个事件的发布接口，所有的事件都需要从这个接口发布出去。

cn.bugstack.springframework.context.support.AbstractApplicationContext

```
public abstract class AbstractApplicationContext extends DefaultResourceLoader implements ConfigurableApplicationContext {

    public static final String APPLICATION_EVENT_MULTICASTER_BEAN_NAME = "applicationEventMulticaster";

    private ApplicationEventMulticaster applicationEventMulticaster;

    @Override
    public void refresh() throws BeansException {

        // 6. 初始化事件发布者
        initApplicationEventMulticaster();

        // 7. 注册事件监听器
        registerListeners();

        // 9. 发布容器刷新完成事件
        finishRefresh();
    }

    private void initApplicationEventMulticaster() {
        ConfigurableListableBeanFactory beanFactory = getBeanFactory();
        applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);
        beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, applicationEventMulticaster);
    }

    private void registerListeners() {
        Collection<ApplicationListener> applicationListeners = getBeansOfType(ApplicationListener.class).values();
        for (ApplicationListener listener : applicationListeners) {
            applicationEventMulticaster.addApplicationListener(listener);
        }
    }

    private void finishRefresh() {
        publishEvent(new ContextRefreshedEvent(this));
    }
```

```
    @Override
    public void publishEvent(ApplicationEvent event) {
        applicationEventMulticaster.multicastEvent(event);
    }

    @Override
    public void close() {
        // 发布容器关闭事件
        publishEvent(new ContextClosedEvent(this));

        // 执行销毁单例 bean 的销毁方法
        getBeanFactory().destroySingletons();
    }

}
```

- 在抽象应用上下文 AbstractApplicationContext#refresh 中，主要新增了 初始化事件发布者、注册事件监听器、发布容器刷新完成事件，三个方法用于处理事件操作。
- 初始化事件发布者(initApplicationEventMulticaster)，主要用于实例化一个 SimpleApplicationEventMulticaster，这是一个事件广播器。
- 注册事件监听器(registerListeners)，通过 getBeansOfType 方法获取到所有从 spring.xml 中加载到的事件配置 Bean 对象。
- 发布容器刷新完成事件(finishRefresh)，发布了第一个服务器启动完成后的事件，这个事件通过 publishEvent 发布出去，其实也就是调用了 applicationEventMulticaster.multicastEvent(event); 方法。
- 最后是一个 close 方法中，新增加了发布一个容器关闭事件。`publishEvent(new ContextClosedEvent(this));`

五、测试

1. 创建一个事件和监听器

cn.bugstack.springframework.test.event.CustomEvent

```
public class CustomEvent extends ApplicationContextEvent {

    private Long id;
    private String message;
```

```
    /**
     * Constructs a prototypical Event.
     *
     * @param source The object on which the Event initially occurred.
     * @throws IllegalArgumentException if source is null.
     */
    public CustomEvent(Object source, Long id, String message) {
        super(source);
        this.id = id;
        this.message = message;
    }

    // ...get/set
}
```

- 创建一个自定义事件，在事件类的构造函数中可以添加自己的想要的入参信息。这个事件类最终会被完成的拿到监听里，所以你添加的属性都会被获得到。

cn.bugstack.springframework.test.event.CustomEventListener

```
public class CustomEventListener implements ApplicationListener<CustomEvent> {

    @Override
    public void onApplicationEvent(CustomEvent event) {
        System.out.println("收到：" + event.getSource() + "消息;时间：
" + new Date());
        System.out.println("消息：" + event.getId() + ":" + event.getMessage());
    }

}
```

- 这个是一个用于监听 CustomEvent 事件的监听器，这里你可以处理自己想要的操作，比如一些用户注册后发送优惠券和短信通知等。
- 另外是关于 `ContextRefreshedEventListener implements ApplicationListener<ContextRefreshedEvent>`、`ContextClosedEventListener implements ApplicationListener<ContextClosedEvent>` 监听器，这里就不演示了，可以参考下源码。

2. 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
```

```
    <bean class="cn.bugstack.springframework.test.event.ContextRefreshedEventListen
er"/>

    <bean class="cn.bugstack.springframework.test.event.CustomEventListener"/>

    <bean class="cn.bugstack.springframework.test.event.ContextClosedEventListener"
/>

</beans>
```

- 在 spring.xml 中配置了三个事件监听器，监听刷新、监控自定义事件、监听关闭事件。

3. 单元测试

```
public class ApiTest {

    @Test
    public void test_event() {
        ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplica
tionContext("classpath:spring.xml");
        applicationContext.publishEvent(new CustomEvent(applicationContext, 1019129
009086763L, "成功了！"));

        applicationContext.registerShutdownHook();
    }

}
```

- 通过使用 applicationContext 新增加的发布事件接口方法，发布一个自定义事件 CustomEvent，并透传了相应的参数信息。

测试结果

```
刷新事件：
cn.bugstack.springframework.test.event.ContextRefreshedEventListener$$EnhancerByCGL
IB$$440a36f5
收到：
cn.bugstack.springframework.context.support.ClassPathXmlApplicationContext@71c7db30
消息;时间：22:32:50
消息：1019129009086763:成功了！
关闭事件：
cn.bugstack.springframework.test.event.ContextClosedEventListener$$EnhancerByCGLIB$
$f4d4b18d
```

```
Process finished with exit code 0
```

- 从测试结果可以看到，我们自己定义的事件和监听，以及监听系统的事件信息，都可以在控制台完整的输出出来了。你也可以尝试增加一些其他事件行为，并调试代码学习观察者模式。

六、总结

- 在整个手写 Spring 框架的学习过程中，可以逐步看到很多设计模式的使用，比如：简单工厂 BeanFactory、工厂方法 FactoryBean、策略模式访问资源，现在又实现了一个观察者模式的具体使用。所以学习 Spring 的过程中，要更加注意关于设计模式的运用，这是你能读懂代码的核心也是学习的重点。
- 那么本章节关于观察者模式的实现过程，主要包括了事件的定义、事件的监听和发布事件，发布完成后根据匹配策略，监听器就会收到属于自己的事件内容，并做相应的处理动作，这样的观察者模式其实日常我们也经常使用，不过在结合 Spring 以后，除了设计模式的学习，还可以学到如何把相应观察者的实现和应用上下文结合。
- 所有在 Spring 学习到的技术、设计、思路都是可以和实际的业务开发结合起来的，而这些看似比较多的代码模块，其实也是按照各自职责一点点的扩充进去的。在自己的学习过程中，可以先动手尝试完成这些框架功能，在一点点通过调试的方式与 Spring 源码进行对照参考，最终也就慢慢掌握这些设计和编码能力了。

代理篇： AOP

第 12 章：基于 JDK、CGlib 实现 AOP 切面

一、技术能力

为什么，你的代码总是糊到猪圈上？

怎么办，知道你在互联网，不知道你在哪个大厂。知道你在加班，不知道你在和哪个产品争辩。知道你在偷懒，不知道你要摸鱼到几点。知道你在搬砖，不知道你在盖哪个猪圈。

当你特别辛苦夜以继日的完成着，每天、每周、每月重复性的工作时，你能获得的成长是最小，得到的回报也是少的。*留着最多的汗、拿着最少的钱*
可能你一激动开始看源码，但不知道看完的源码能用到什么地方。看设计模式，看的时候懂，但改自己的代码又下不去手。其实一方面是本身技术栈的知识面不足，另外一方面是自己储备的代码也不够。

最终也就导致根本没法把一些列的知识串联起来，就像你看了 HashMap，但也联想不到分库分表组件中的数据散列也会用到了 HashMap 中的扰动函数思想和泊松分布验证、看了 Spring 源码，也读不出来 Mybatis 是如何解决只定义 Dao 接口就能使用配置或者注解对数据库进行 CRUD 操作、看来 JDK 的动态代理，也想不到 AOP 是如何设计的。所以成体系学习，加强技术栈知识的完整性，才能更好的用上这些学习到的编码能力。

二、目标

到本章节我们将要从 IOC 的实现，转入到关于 AOP(Aspect Oriented Programming) 内容的开发。在软件行业，AOP 意为：面向切面编程，通过预编译的方式和运行期间动态代理实现程序功能功能的统一维护。其实 AOP 也是 OOP 的延续，在 Spring 框架中是一个非常重要的内容，使用 AOP 可以对业务逻辑的各个部分进行隔离，从而使各模块间的业务逻辑耦合度降低，提高代码

的可复用性，同时也能提高开发效率。

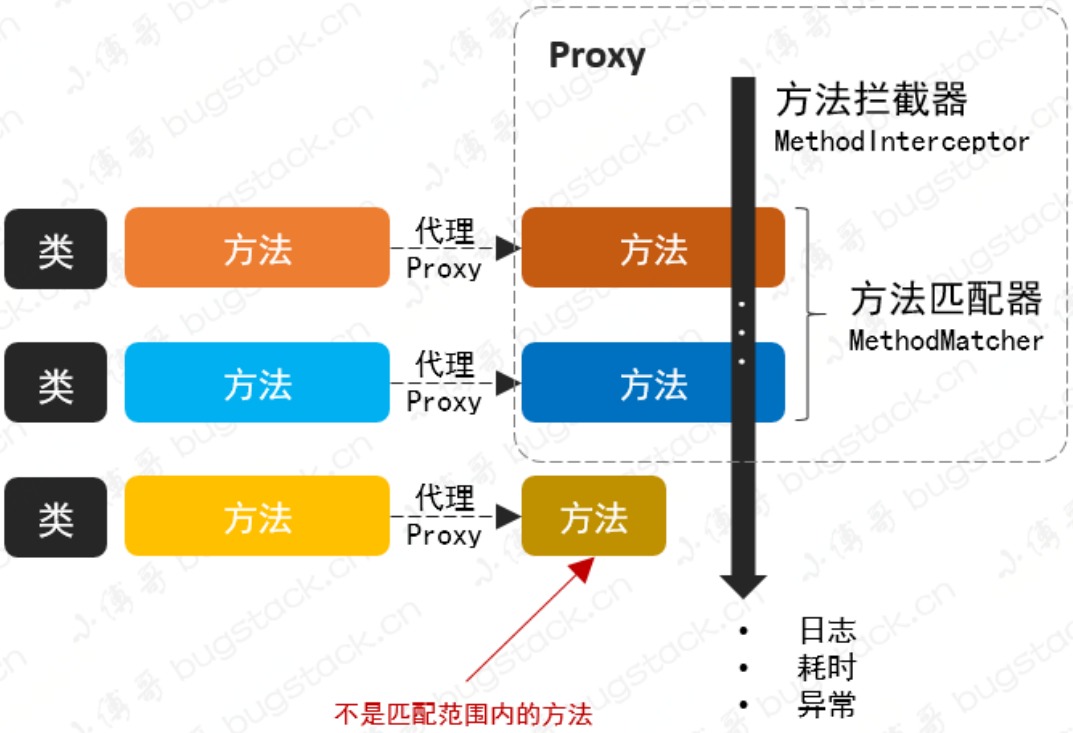
关于 AOP 的核心技术实现主要是动态代理的使用，就像你可以给一个接口的实现类，使用代理的方式替换掉这个实现类，使用代理类来处理你需要的逻辑。比如：

```
@Test
public void test_proxy_class() {
    IUserService userService = (IUserService) Proxy.newProxyInstance(Thread.current
Thread().getContextClassLoader(), new Class[]{IUserService.class}, (proxy, method,
args) -> "你被代理了！");
    String result = userService.queryUserInfo();
    System.out.println("测试结果：" + result);
}
```

代理类的实现基本都大家都见过，那么有了一个基本的思路后，接下来就需要考虑下怎么给方法做代理呢，而不是代理类。另外怎么去代理所有符合某些规则的所有类中方法呢。如果可以代理掉所有类的方法，就可以做一个方法拦截器，给所有被代理的方法添加上一些自定义处理，比如打印日志、记录耗时、监控异常等。

三、方案

在把 AOP 整个切面设计融合到 Spring 前，我们需要解决两个问题，包括：如何给符合规则的方法做代理，以及做完代理方法的案例后，把类的职责拆分出来。而这两个功能点的实现，都是以切面的思想进行设计和开发。如果不是很清楚 AOP 是啥，你可以把切面理解为用刀切韭菜，一根一根切总是有点慢，那么用手(代理)把韭菜捏成一把，用菜刀或者斧头这样不同的拦截操作来处理。而程序中其实也是一样，只不过韭菜变成了方法，菜刀变成了拦截方法。整体设计结构如下图：



- 就像你在使用 Spring 的 AOP 一样，只处理一些需要被拦截的方法。在拦截方法后，执行你对方法的扩展操作。
- 那么我们就需要先来实现一个可以代理方法的 Proxy，其实代理方法主要是使用到方法拦截器类处理方法的调用 `MethodInterceptor#invoke`，而不是直接使用 invoke 方法中的入参 Method method 进行 `method.invoke(targetObj, args)` 这块是整个使用时的差异。
- 除了以上的核心功能实现，还需要使用到 `org.aspectj.weaver.tools.PointcutParser` 处理拦截表达式 `"execution(* cn.bugstack.springframework.test.bean.IUserService.*(..))"`，有了方法代理和处理拦截，我们就可以完成设计出一个 AOP 的雏形了。

四、实现

1. 工程结构

```
small-spring-step-11
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── aop
    │           │   ├── aspectj
```

```
    │           │   │   └── AspectJExpressionPointcut.java
    │           │   ├── framework
    │           │   │   ├── AopProxy.java
    │           │   │   ├── Cglib2AopProxy.java
    │           │   │   ├── JdkDynamicAopProxy.java
    │           │   │   └── ReflectiveMethodInvocation.java
    │           │   ├── AdvisedSupport.java
    │           │   ├── ClassFilter.java
    │           │   ├── MethodMatcher.java
    │           │   ├── Pointcut.java
    │           │   └── TargetSource.java
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   │   ├── BeanPostProcessor.java
    │           │   │   │   ├── BeanReference.java
    │           │   │   │   ├── ConfigurableBeanFactory.java
    │           │   │   │   └── SingletonBeanRegistry.java
    │           │   │   ├── support
    │           │   │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   │   ├── AbstractBeanFactory.java
    │           │   │   │   ├── BeanDefinitionReader.java
    │           │   │   │   ├── BeanDefinitionRegistry.java
    │           │   │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   │   ├── DefaultListableBeanFactory.java
    │           │   │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   │   ├── DisposableBeanAdapter.java
    │           │   │   │   ├── FactoryBeanRegistrySupport.java
    │           │   │   │   ├── InstantiationStrategy.java
    │           │   │   │   └── SimpleInstantiationStrategy.java
    │           │   │   ├── support
    │           │   │   │   └── XmlBeanDefinitionReader.java
    │           │   │   ├── Aware.java
    │           │   │   ├── BeanClassLoaderAware.java
    │           │   │   ├── BeanFactory.java
    │           │   │   ├── BeanFactoryAware.java
    │           │   │   ├── BeanNameAware.java
    │           │   │   ├── ConfigurableListableBeanFactory.java
    │           │   │   ├── DisposableBean.java
    │           │   │   ├── FactoryBean.java
```

```
    │           │   │   ├── HierarchicalBeanFactory.java
    │           │   │   ├── InitializingBean.java
    │           │   │   └── ListableBeanFactory.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
    │           │   └── PropertyValues.java
    │           ├── context
    │           │   ├── event
    │           │   │   ├── AbstractApplicationEventMulticaster.java
    │           │   │   ├── ApplicationContextEvent.java
    │           │   │   ├── ApplicationEventMulticaster.java
    │           │   │   ├── ContextClosedEvent.java
    │           │   │   ├── ContextRefreshedEvent.java
    │           │   │   └── SimpleApplicationEventMulticaster.java
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   ├── ApplicationContextAwareProcessor.java
    │           │   │   └── ClassPathXmlApplicationContext.java
    │           │   ├── ApplicationContext.java
    │           │   ├── ApplicationContextAware.java
    │           │   ├── ApplicationEvent.java
    │           │   ├── ApplicationEventPublisher.java
    │           │   ├── ApplicationListener.java
    │           │   └── ConfigurableApplicationContext.java
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           └── utils
    │               └── ClassUtils.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── IUserService.java
                │   ├── UserService.java
                │   └── UserServiceInterceptor.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

AOP 切点表达式和使用以及基于 JDK 和 CGLIB 的动态代理类关系，如图 12-2

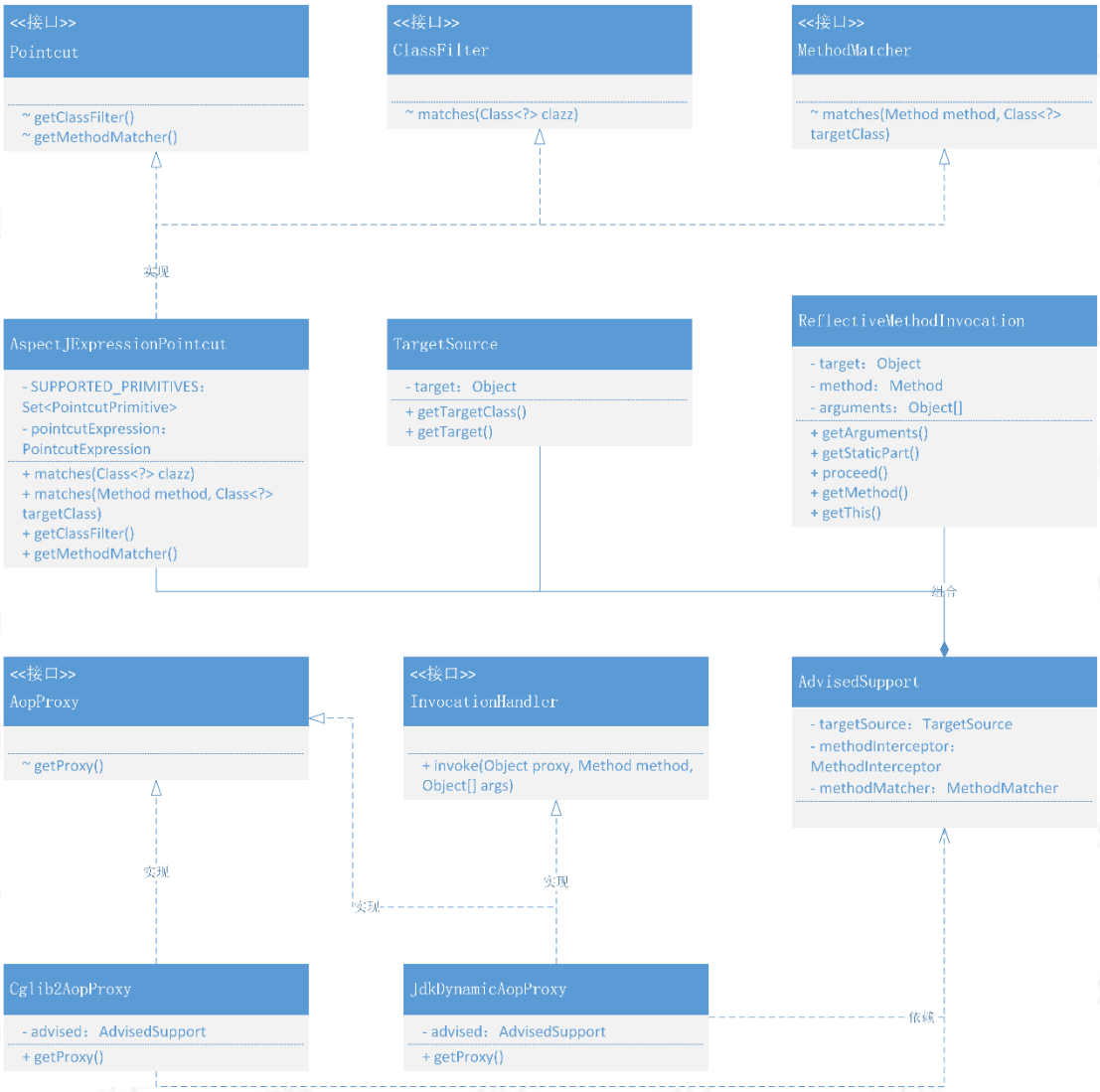


图 12-2

- 整个类关系图就是 AOP 实现核心逻辑的地方，上面部分是关于方法的匹配实现，下面从 AopProxy 开始是关于方法的代理操作。
- AspectJExpressionPointcut 的核心功能主要依赖于 aspectj 组件并处理 Pointcut、ClassFilter,、MethodMatcher 接口实现，专门用于处理类和方法的匹配过滤操作。
- AopProxy 是代理的抽象对象，它的实现主要是基于 JDK 的代理和 Cglib 代理。在前面章节关于对象的实例化 CglibSubclassingInstantiationStrategy，我们也使用过 Cglib 提供的功能。

2. 代理方法案例

在实现 AOP 的核心功能之前，我们先做一个代理方法的案例，通过这样一个可以概括代理方法的核心全貌，可以让大家更好的理解后续拆解各个方法，设计成解耦功能的 AOP 实现过程。

单元测试

```
@Test
public void test_proxy_method() {
    // 目标对象(可以替换成任何的目标对象)
    Object targetObj = new UserService();
    // AOP 代理
    IUserService proxy = (IUserService) Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(), targetObj.getClass().getInterfaces(), new InvocationHandler() {
        // 方法匹配器
        MethodMatcher methodMatcher = new AspectJExpressionPointcut("execution(* cn.bugstack.springframework.test.bean.IUserService.*(..))");
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            if (methodMatcher.matches(method, targetObj.getClass())) {
                // 方法拦截器
                MethodInterceptor methodInterceptor = invocation -> {
                    long start = System.currentTimeMillis();
                    try {
                        return invocation.proceed();
                    } finally {
                        System.out.println("监控 - Begin By AOP");
                        System.out.println("方法名称:" + invocation.getMethod().getName());
                        System.out.println("方法耗时:" + (System.currentTimeMillis() - start) + "ms");
                        System.out.println("监控 - End\r\n");
                    }
                };
                // 反射调用
                return methodInterceptor.invoke(new ReflectiveMethodInvocation(targetObj, method, args));
            }
            return method.invoke(targetObj, args);
        }
```

```
    });
    String result = proxy.queryUserInfo();
    System.out.println("测试结果：" + result);
}
```

- 首先整个案例的目标是给一个 UserService 当成目标对象，对类中的所有方法进行拦截添加监控信息打印处理。
- 从案例中你可以看到有代理的实现 Proxy.newProxyInstance，有方法的匹配 MethodMatcher，有反射的调用 invoke(Object proxy, Method method, Object[] args)，也用用户自己拦截方法后的操作。这样一看其实和我们使用的 AOP 就非常类似了，只不过你在使用 AOP 的时候是框架已经提供更好的功能，这里是把所有的核心过程给你展示出来了。

测试结果

```
监控 - Begin By AOP
方法名称：queryUserInfo
方法耗时：86ms
监控 - End

测试结果：小傅哥，100001，深圳

Process finished with exit code 0
```

- 从测试结果可以看到我们已经对 UserService#queryUserInfo 方法进行了拦截监控操作，其实后面我们实现的 AOP 就是现在体现出的结果，只不过我们需要把这部分测试的案例解耦为更具有扩展性的各个模块实现。

拆解案例

图 12-3

- 拆解过程可以参考截图 12-3，我们需要把代理对象拆解出来，因为它可以是 JDK 的实现也可以是 Cglib 的处理。
- 方法匹配器操作其实已经是一个单独的实现类了，不过我们还需要把传入的目标对象、方法匹配、拦截方法，都进行统一的包装，方便外部调用时进行一个入参透传。
- 最后其实是 `ReflectiveMethodInvocation` 的使用，它目前已经是实现 `MethodInvocation` 接口的一个包装后的类，参数信息包括：调用的对象、调用的方法、调用的入参。

3. 切点表达式

定义接口

cn.bugstack.springframework.aop.Pointcut

```
public interface Pointcut {

    /**
     * Return the ClassFilter for this pointcut.
     * @return the ClassFilter (never <code>null</code>)
     */
    ClassFilter getClassFilter();

    /**
     * Return the MethodMatcher for this pointcut.
     * @return the MethodMatcher (never <code>null</code>)
     */
    MethodMatcher getMethodMatcher();

}
```

- 切入点接口，定义用于获取 ClassFilter、MethodMatcher 的两个类，这两个接口获取都是切点表达式提供的内容。

cn.bugstack.springframework.aop.ClassFilter

```
public interface ClassFilter {

    /**
     * Should the pointcut apply to the given interface or target class?
     * @param clazz the candidate target class
     * @return whether the advice should apply to the given target class
     */
    boolean matches(Class<?> clazz);
```

```
}
```

- 定义类匹配类，用于切点找到给定的接口和目标类。

cn.bugstack.springframework.aop.MethodMatcher

```
public interface MethodMatcher {

    /**
     * Perform static checking whether the given method matches. If this
     * @return whether or not this method matches statically
     */
    boolean matches(Method method, Class<?> targetClass);

}
```

- 方法匹配，找到表达式范围内匹配下的目标类和方法。在上文的案例中有所体现：`methodMatcher.matches(method, targetObj.getClass())`

实现切点表达式类

```
public class AspectJExpressionPointcut implements Pointcut, ClassFilter, MethodMatc
her {

    private static final Set<PointcutPrimitive> SUPPORTED_PRIMITIVES = new HashSet<
PointcutPrimitive>();

    static {
        SUPPORTED_PRIMITIVES.add(PointcutPrimitive.EXECUTION);
    }

    private final PointcutExpression pointcutExpression;

    public AspectJExpressionPointcut(String expression) {
        PointcutParser pointcutParser = PointcutParser.getPointcutParserSupportingS
pecifiedPrimitivesAndUsingSpecifiedClassLoaderForResolution(SUPPORTED_PRIMITIVES, t
his.getClass().getClassLoader());
        pointcutExpression = pointcutParser.parsePointcutExpression(expression);
    }

    @Override
    public boolean matches(Class<?> clazz) {
        return pointcutExpression.couldMatchJoinPointsInType(clazz);
    }
```

```
    @Override
    public boolean matches(Method method, Class<?> targetClass) {
        return pointcutExpression.matchesMethodExecution(method).alwaysMatches();
    }

    @Override
    public ClassFilter getClassFilter() {
        return this;
    }

    @Override
    public MethodMatcher getMethodMatcher() {
        return this;
    }

}
```

- 切点表达式实现了 Pointcut、ClassFilter、MethodMatcher，三个接口定义方法，同时这个类主要是对 aspectj 包提供的表达式校验方法使用。
- 匹配 matches：`pointcutExpression.couldMatchJoinPointsInType(clazz)`、`pointcutExpression.matchesMethodExecution(method).alwaysMatches()`，这部分内容可以单独测试验证。

匹配验证

```
@Test
public void test_aop() throws NoSuchMethodException {
    AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut("execution(*
 cn.bugstack.springframework.test.bean.UserService.*(..))");
    Class<UserService> clazz = UserService.class;
    Method method = clazz.getDeclaredMethod("queryUserInfo");

    System.out.println(pointcut.matches(clazz));
    System.out.println(pointcut.matches(method, clazz));

    // true、true
}
```

- 这里单独提供出来一个匹配方法的验证测试，可以看看你拦截的方法与对应的对象是否匹配。

4. 包装切面通知信息

cn.bugstack.springframework.aop.AdvisedSupport

```
public class AdvisedSupport {

    // 被代理的目标对象
    private TargetSource targetSource;
    // 方法拦截器
    private MethodInterceptor methodInterceptor;
    // 方法匹配器(检查目标方法是否符合通知条件)
    private MethodMatcher methodMatcher;

    // ...get/set
}
```

- AdvisedSupport，主要是用于把代理、拦截、匹配的各项属性包装到一个类中，方便在 Proxy 实现类进行使用。*这和你的业务开发中包装入参是一个道理*
- TargetSource，是一个目标对象，在目标对象类中提供 Object 入参属性，以及获取目标类 TargetClass 信息。
- MethodInterceptor，是一个具体拦截方法实现类，由用户自己实现 MethodInterceptor#invoke 方法，做具体的处理。*像我们本文的案例中是做方法监控处理*
- MethodMatcher，是一个匹配方法的操作，这个对象由 AspectJExpressionPointcut 提供服务。

5. 代理抽象实现(JDK&Cglib)

定义接口

cn.bugstack.springframework.aop.framework

```
public interface AopProxy {

    Object getProxy();

}
```

- 定义一个标准接口，用于获取代理类。因为具体实现代理的方式可以有 JDK 方式，也可以是 Cglib 方式，所以定义接口会更加方便管理实现类。

cn.bugstack.springframework.aop.framework.JdkDynamicAopProxy

```
public class JdkDynamicAopProxy implements AopProxy, InvocationHandler {

    private final AdvisedSupport advised;

    public JdkDynamicAopProxy(AdvisedSupport advised) {
        this.advised = advised;
    }

    @Override
    public Object getProxy() {
        return Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(
), advised.getTargetSource().getTargetClass(), this);
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwab
le {
        if (advised.getMethodMatcher().matches(method, advised.getTargetSource().ge
tTarget().getClass())) {
            MethodInterceptor methodInterceptor = advised.getMethodInterceptor();
            return methodInterceptor.invoke(new ReflectiveMethodInvocation(advised.
getTargetSource().getTarget(), method, args));
        }
        return method.invoke(advised.getTargetSource().getTarget(), args);
    }

}
```

- 基于 JDK 实现的代理类，需要实现接口 AopProxy、InvocationHandler，这样就可以把代理对象 getProxy 和反射调用方法 invoke 分开处理了。
- getProxy 方法中的是代理一个对象的操作，需要提供入参 ClassLoader、AdvisedSupport、和当前这个类 this，因为这个类提供了 invoke 方法。
- invoke 方法中主要处理匹配的方法后，使用用户自己提供的方法拦截实现，做反射调用 methodInterceptor.invoke 。
- 这里还有一个 ReflectiveMethodInvocation，其他它就是一个入参的包装信息，提供了入参对象：目标对象、方法、入参。

cn.bugstack.springframework.aop.framework.Cglib2AopProxy

```
public class Cglib2AopProxy implements AopProxy {

    private final AdvisedSupport advised;

    public Cglib2AopProxy(AdvisedSupport advised) {
        this.advised = advised;
```

```
    }

    @Override
    public Object getProxy() {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(advised.getTargetSource().getTarget().getClass());
        enhancer.setInterfaces(advised.getTargetSource().getTargetClass());
        enhancer.setCallback(new DynamicAdvisedInterceptor(advised));
        return enhancer.create();
    }

    private static class DynamicAdvisedInterceptor implements MethodInterceptor {

        @Override
        public Object intercept(Object o, Method method, Object[] objects, MethodPr
oxy methodProxy) throws Throwable {
            CglibMethodInvocation methodInvocation = new CglibMethodInvocation(advi
sed.getTargetSource().getTarget(), method, objects, methodProxy);
            if (advised.getMethodMatcher().matches(method, advised.getTargetSource(
).getTarget().getClass())) {
                return advised.getMethodInterceptor().invoke(methodInvocation);
            }
            return methodInvocation.proceed();
        }
    }

    private static class CglibMethodInvocation extends ReflectiveMethodInvocation {

        @Override
        public Object proceed() throws Throwable {
            return this.methodProxy.invoke(this.target, this.arguments);
        }

    }

}
```

- 基于 Cglib 使用 Enhancer 代理的类可以在运行期间为接口使用底层 ASM 字节码增强技术处理对象的代理对象生成，因此被代理类不需要实现任何接口。
- 关于扩展进去的用户拦截方法，主要是在 Enhancer#setCallback 中处理，用户自己的新增的拦截处理。这里可以看到 DynamicAdvisedInterceptor#intercept 匹配方法后做了相应的反射操作。

五、测试

1. 事先准备

```
public class UserService implements IUserService {

    public String queryUserInfo() {
        try {
            Thread.sleep(new Random(1).nextInt(100));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "小傅哥，100001，深圳";
    }

    public String register(String userName) {
        try {
            Thread.sleep(new Random(1).nextInt(100));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "注册用户：" + userName + " success！";
    }

}
```

- 在 UserService 中提供了 2 个不同方法，另外你还可以增加新的类来加入测试。后面我们的测试过程，会给这个两个方法添加我们的拦截处理，打印方法执行耗时。

2. 自定义拦截方法

```
public class UserServiceInterceptor implements MethodInterceptor {

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        long start = System.currentTimeMillis();
        try {
            return invocation.proceed();
        } finally {
            System.out.println("监控 - Begin By AOP");
            System.out.println("方法名称：" + invocation.getMethod());
```

```
            System.out.println("方法耗时：
" + (System.currentTimeMillis() - start) + "ms");
            System.out.println("监控 - End\r\n");
        }
    }

}
```

- 用户自定义的拦截方法需要实现 MethodInterceptor 接口的 invoke 方法，使用方式与 Spring AOP 非常相似，也是包装 invocation.proceed() 放行，并在 finally 中添加监控信息。

3. 单元测试

```
@Test
public void test_dynamic() {
    // 目标对象
    IUserService userService = new UserService();

    // 组装代理信息
    AdvisedSupport advisedSupport = new AdvisedSupport();
    advisedSupport.setTargetSource(new TargetSource(userService));
    advisedSupport.setMethodInterceptor(new UserServiceInterceptor());
    advisedSupport.setMethodMatcher(new AspectJExpressionPointcut("execution(* cn.b
ugstack.springframework.test.bean.IUserService.*(..))"));

    // 代理对象(JdkDynamicAopProxy)
    IUserService proxy_jdk = (IUserService) new JdkDynamicAopProxy(advisedSupport).
getProxy();
    // 测试调用
    System.out.println("测试结果：" + proxy_jdk.queryUserInfo());

    // 代理对象(Cglib2AopProxy)
    IUserService proxy_cglib = (IUserService) new Cglib2AopProxy(advisedSupport).ge
tProxy();
    // 测试调用
    System.out.println("测试结果：" + proxy_cglib.register("花花"));
}
```

- 整个案例测试了 AOP 在于 Spring 结合前的核心代码，包括什么是目标对象、怎么组装代理信息、如何调用代理对象。
- AdvisedSupport，包装了目标对象、用户自己实现的拦截方法以及方法匹配表达式。

- 之后就是分别调用 JdkDynamicAopProxy、Cglib2AopProxy，两个不同方式实现的代理类，看看是否可以成功拦截方法

测试结果

```
监控 - Begin By AOP
方法名称：
public abstract java.lang.String cn.bugstack.springframework.test.bean.IUserService
.queryUserInfo()
方法耗时：86ms
监控 - End

测试结果：小傅哥，100001，深圳
监控 - Begin By AOP
方法名称：
public java.lang.String cn.bugstack.springframework.test.bean.UserService.register(
java.lang.String)
方法耗时：97ms
监控 - End

测试结果：注册用户：花花 success!

Process finished with exit code 0
```

- 如 AOP 功能定义一样，我们可以通过这样的代理方式、方法匹配和拦截后，在对应的目标方法下，做了拦截操作进行监控信息打印。

六、总结

- 从本文对 Proxy#newProxyInstance、MethodInterceptor#invoke，的使用验证切面核心原理以及再把功能拆解到 Spring 框架实现中，可以看到一个貌似复杂的技术其实核心内容往往没有太多，但因为需要为了满足后续更多的扩展就需要进行职责解耦和包装，通过这样设计模式的使用，以此让调用方能更加简化，自身也可以不断按需扩展。
- AOP 的功能实现目前还没有与 Spring 结合，只是对切面技术的一个具体实现，你可以先学习到如何处理代理对象、过滤方法、拦截方法，以及使用 Cglib 和 JDK 代理的区别，其实这与的技术不只是在 Spring 框架中有所体现，在其他各类需要减少人工硬编码的场景下，都会用到。*比如 RPC、Mybatis、MQ、分布式任务*
- 一些核心技术的使用上，都是具有很强的关联性的，它们也不是孤立存在的。而这个能把整个技术栈串联起来的过程，需要你来大量的学习、积累、由点到面的铺设，才能在一个知识点的学习拓展到一个知识面和知识体系的建设。

第 13 章：把 AOP 扩展到 Bean 的生命周期

一、功能整合

嘎小子，这片代码水太深你把握不住！

在电视剧《楚汉传奇》中有这么一段刘邦与韩信的饮酒对话，刘邦问韩信我那个曹参读过书见过世面能带多少兵，韩信说能带一万五，又补充说一万五都吃力。刘邦又一一说出樊哙、卢绾、周勃，韩信笑着说不足 2 万，脑子不行。这时候刘邦有点挂不住脸了，问：那我呢，我能带多少兵。韩信说，你能带十万。刘邦一看比他们都多，啊，还行。转头一想就问韩信那你呢，你能带多少兵。韩信喝多了，说啊，我，我多多益善。这时候刘邦恼了领导劲上来了，问：那我为什么能管着你，你给我说，说呀！

这像不像你领导问你，你能写多少代码、搭多少框架、接多少项目。可能很大一部分没经历太多的新人码农，仅仅是能完成一些简单的功能模块开发，而没有办法驾驭整个项目的涉及到的所有工程，也不能为项目提炼出一些可复用的通用性组件模块。在初级码农的心里，接一点需求还好，但没有人带的时候完全接一个较大型项目就会比较慌了，不知道这里有没有坑，自己也把握住不。

这些代码一块块的带着能写，但是都弄到一块，就太难了！

在代码开发成长的这条路上，要经历 CRUD、ERP 查数据、接口包装、功能开发、服务整合、系统建设等，一直到独立带人承担较大型项目的搭建。这一过程需要你能有大量的编写代码经验积累和复杂问题的处理手段，之后才能一段段的把看似独立的模块后者代码片段组装成一个较大型能跑起来的项目。就像 Spring 的开发过程一样，我们总是不断在添加新的功能片段，最后又把技术实现与 Spring 容器整合，让使用方可以更简单的运用 Spring 提供的能力。

二、目标

在上一章节我们通过基于 Proxy.newProxyInstance 代理操作中处理方法匹配和方法拦截，对匹配的对象进行自定义的处理操作。并把这样的技术核心内容

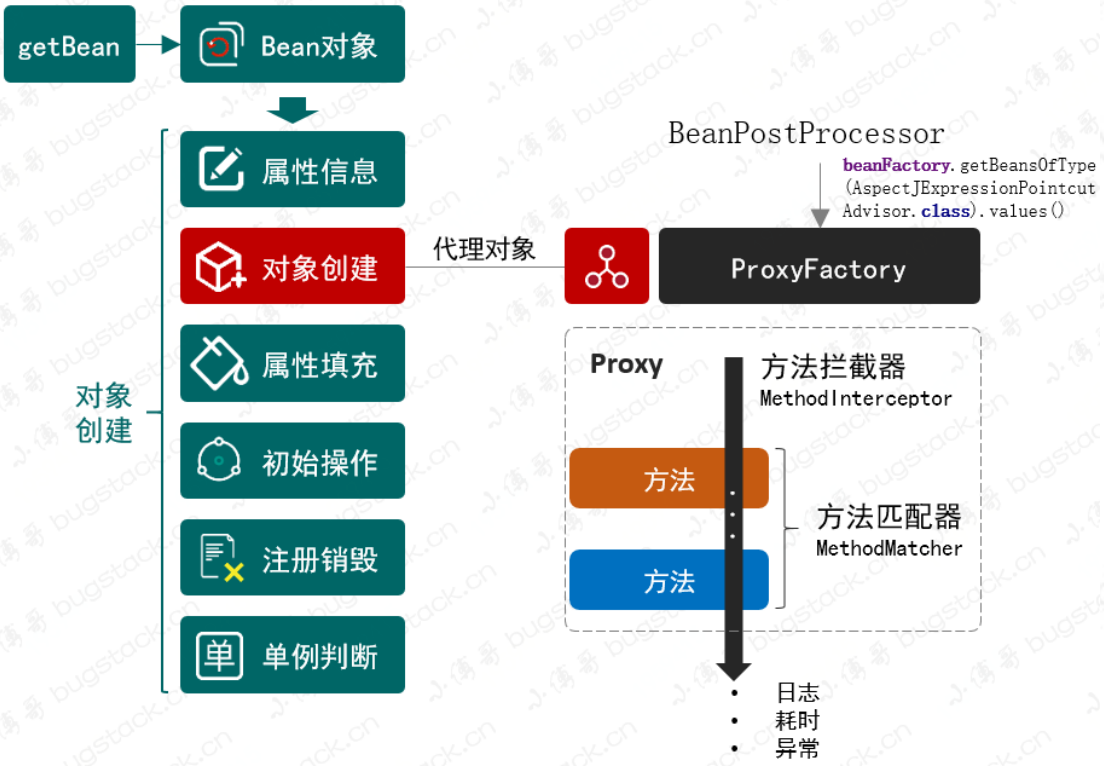
拆解到 Spring 中，用于实现 AOP 部分，通过拆分后基本可以明确各个类的职责，包括你的代理目标对象属性、拦截器属性、方法匹配属性，以及两种不同的代理操作 JDK 和 CGlib 的方式。

再有了一个 AOP 核心功能的实现后，我们可以通过单元测试的方式进行验证切面功能对方法进行拦截，但如果这是一个面向用户使用的功能，就不太可能让用户这么复杂且没有与 Spring 结合的方式单独使用 AOP，虽然可以满足需求，但使用上还是过去分散。

因此我们需要在本章节完成 AOP 核心功能与 Spring 框架的整合，最终能通过在 Spring 配置的方式完成切面的操作。

三、方案

其实在有了 AOP 的核心功能实现后，把这部分功能服务融入到 Spring 其实也不难，只不过要解决几个问题，包括：怎么借着 BeanPostProcessor 把动态代理融入到 Bean 的生命周期中，以及如何组装各项切点、拦截、前置的功能和适配对应的代理器。整体设计结构如下图：



- 为了可以让对象创建过程中，能把 xml 中配置的代理对象也就是切面的一些类对象实例化，就需要用到 BeanPostProcessor 提供的方法，因为这个类的中的方法可以分别作用与 Bean 对象执行初始化前后修改 Bean 的对象的扩展信息。但这里需要集合于 BeanPostProcessor 实现新的接口和实现类，这样才能定向获取对应的类信息。
- 但因为创建的是代理对象不是之前流程里的普通对象，所以我们需要前置于其他对象的创建，所以在实际开发的过程中，需要在 AbstractAutowireCapableBeanFactory#createBean 优先完成 Bean 对象的判断，是否需要代理，有则直接返回代理对象。*在 Spring 的源码中会有 createBean 和 doCreateBean 的方法拆分*
- 这里还包括要解决方法拦截器的具体功能，提供一些 BeforeAdvice、AfterAdvice 的实现，让用户可以更简化的使用切面功能。除此之外还包括需要包装切面表达式以及拦截方法的整合，以及提供不同类型的代理方式的代理工厂，来包装我们的切面服务。

四、实现

1. 工程结构

```
small-spring-step-12
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── aop
    │           │   ├── aspectj
    │           │   │   └── AspectJExpressionPointcut.java
    │           │   │   └── AspectJExpressionPointcutAdvisor.java
    │           │   ├── framework
    │           │   │   ├── adapter
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── autoproxy
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── AopProxy.java
    │           │   │   ├── Cglib2AopProxy.java
    │           │   │   ├── JdkDynamicAopProxy.java
    │           │   │   ├── ProxyFactory.java
    │           │   │   └── ReflectiveMethodInvocation.java
    │           │   ├── AdvisedSupport.java
    │           │   ├── Advisor.java
    │           │   ├── BeforeAdvice.java
    │           │   ├── ClassFilter.java
```

```
    │           │   ├── MethodBeforeAdvice.java
    │           │   ├── MethodMatcher.java
    │           │   ├── Pointcut.java
    │           │   ├── PointcutAdvisor.java
    │           │   └── TargetSource.java
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   │   ├── BeanPostProcessor.java
    │           │   │   │   ├── BeanReference.java
    │           │   │   │   ├── ConfigurableBeanFactory.java
    │           │   │   │   ├── InstantiationAwareBeanPostProcessor.java
    │           │   │   │   └── SingletonBeanRegistry.java
    │           │   │   ├── support
    │           │   │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   │   ├── AbstractBeanFactory.java
    │           │   │   │   ├── BeanDefinitionReader.java
    │           │   │   │   ├── BeanDefinitionRegistry.java
    │           │   │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   │   ├── DefaultListableBeanFactory.java
    │           │   │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   │   ├── DisposableBeanAdapter.java
    │           │   │   │   ├── FactoryBeanRegistrySupport.java
    │           │   │   │   ├── InstantiationStrategy.java
    │           │   │   │   └── SimpleInstantiationStrategy.java
    │           │   │   ├── support
    │           │   │   │   └── XmlBeanDefinitionReader.java
    │           │   │   ├── Aware.java
    │           │   │   ├── BeanClassLoaderAware.java
    │           │   │   ├── BeanFactory.java
    │           │   │   ├── BeanFactoryAware.java
    │           │   │   ├── BeanNameAware.java
    │           │   │   ├── ConfigurableListableBeanFactory.java
    │           │   │   ├── DisposableBean.java
    │           │   │   ├── FactoryBean.java
    │           │   │   ├── HierarchicalBeanFactory.java
    │           │   │   ├── InitializingBean.java
    │           │   │   └── ListableBeanFactory.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
```

```
    │           │   └── PropertyValues.java
    │           ├── context
    │           │   ├── event
    │           │   │   ├── AbstractApplicationEventMulticaster.java
    │           │   │   ├── ApplicationContextEvent.java
    │           │   │   ├── ApplicationEventMulticaster.java
    │           │   │   ├── ContextClosedEvent.java
    │           │   │   ├── ContextRefreshedEvent.java
    │           │   │   └── SimpleApplicationEventMulticaster.java
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   ├── ApplicationContextAwareProcessor.java
    │           │   │   └── ClassPathXmlApplicationContext.java
    │           │   ├── ApplicationContext.java
    │           │   ├── ApplicationContextAware.java
    │           │   ├── ApplicationEvent.java
    │           │   ├── ApplicationEventPublisher.java
    │           │   ├── ApplicationListener.java
    │           │   └── ConfigurableApplicationContext.java
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           └── utils
    │               └── ClassUtils.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── IUserService.java
                │   ├── UserService.java
                │   └── UserServiceInterceptor.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

AOP 动态代理融入到 Bean 的生命周期中类关系，如图 13-2

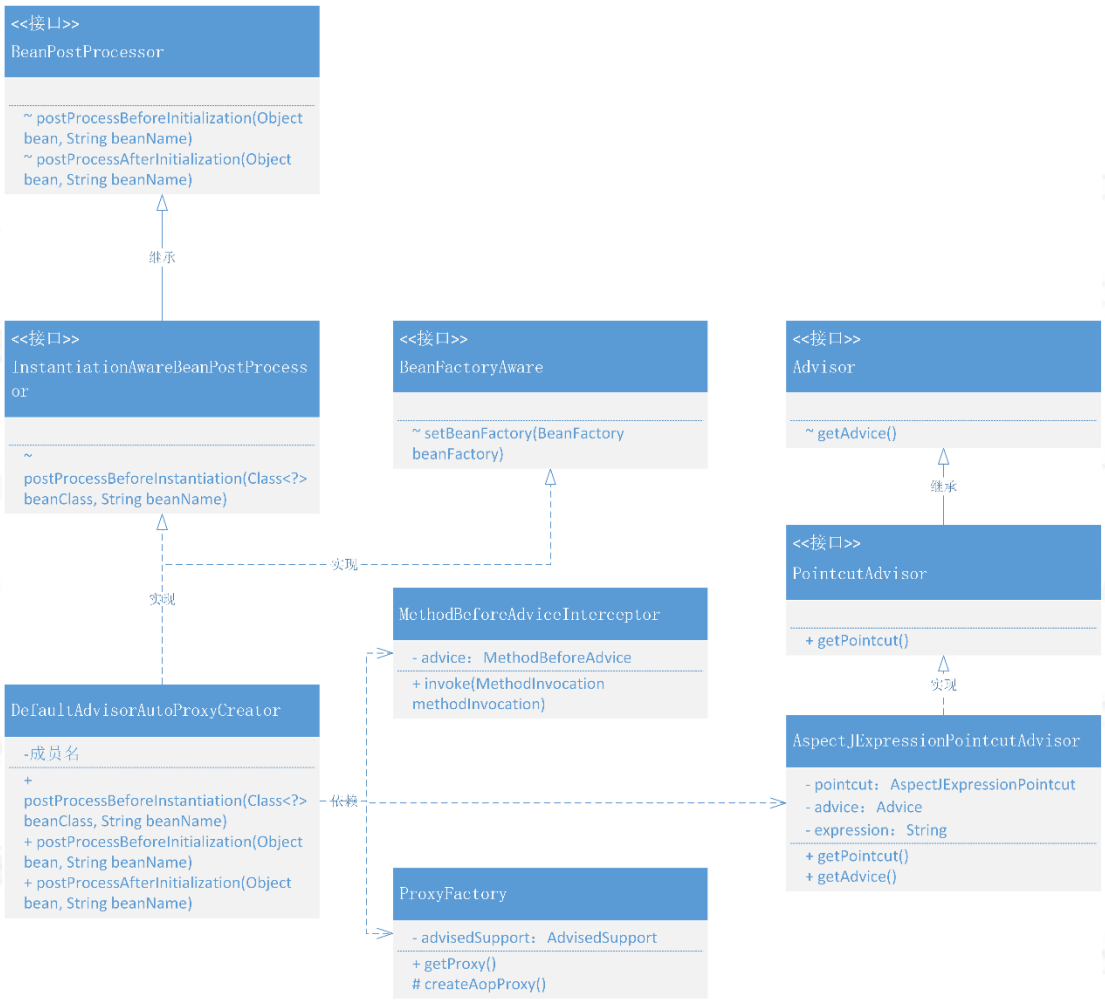


图 13-2

- 整个类关系图中可以看到，在以 BeanPostProcessor 接口实现继承的 InstantiationAwareBeanPostProcessor 接口后，做了一个自动代理创建的类 DefaultAdvisorAutoProxyCreator，这个类的就是用于处理整个 AOP 代理融入到 Bean 生命周期中的核心类。
- DefaultAdvisorAutoProxyCreator 会依赖于拦截器、代理工厂和 Pointcut 与 Advisor 的包装服务 AspectJExpressionPointcutAdvisor，由它提供切面、拦截方法和表达式。
- Spring 的 AOP 把 Advice 细化了 BeforeAdvice、AfterAdvice、AfterReturningAdvice、ThrowsAdvice，目前我们做的测试案例中只用到了 BeforeAdvice，这部分可以对照 Spring 的源码进行补充测试。

2. 定义 Advice 拦截器链

cn.bugstack.springframework.aop.BeforeAdvice

```
public interface BeforeAdvice extends Advice {

}
```

cn.bugstack.springframework.aop.MethodBeforeAdvice

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    /**
     * Callback before a given method is invoked.
     *
     * @param method method being invoked
     * @param args   arguments to the method
     * @param target target of the method invocation. May be <code>null</code>.
     * @throws Throwable if this object wishes to abort the call.
     *                   Any exception thrown will be returned to the caller if it'
s
     *                   allowed by the method signature. Otherwise the exception
     *                   will be wrapped as a runtime exception.
     */
    void before(Method method, Object[] args, Object target) throws Throwable;

}
```

- 在 Spring 框架中，Advice 都是通过方法拦截器 MethodInterceptor 实现的。环绕 Advice 类似一个拦截器的链路，Before Advice、After advice 等，不过暂时我们需要那么多就只定义了一个 MethodBeforeAdvice 的接口定义。

3. 定义 Advisor 访问者

cn.bugstack.springframework.aop.Advisor

```
public interface Advisor {

    /**
     * Return the advice part of this aspect. An advice may be an
     * interceptor, a before advice, a throws advice, etc.
     * @return the advice that should apply if the pointcut matches
     * @see org.aopalliance.intercept.MethodInterceptor
     * @see BeforeAdvice
     */
    Advice getAdvice();

}
```

cn.bugstack.springframework.aop.PointcutAdvisor

```
public interface PointcutAdvisor extends Advisor {

    /**
     * Get the Pointcut that drives this advisor.
     */
    Pointcut getPointcut();

}
```

- Advisor 承担了 Pointcut 和 Advice 的组合，Pointcut 用于获取 JoinPoint，而 Advice 决定于 JoinPoint 执行什么操作。

cn.bugstack.springframework.aop.aspectj.AspectJExpressionPointcutAdvisor

```
public class AspectJExpressionPointcutAdvisor implements PointcutAdvisor {

    // 切面
    private AspectJExpressionPointcut pointcut;
    // 具体的拦截方法
    private Advice advice;
    // 表达式
    private String expression;

    public void setExpression(String expression){
        this.expression = expression;
    }

    @Override
    public Pointcut getPointcut() {
        if (null == pointcut) {
            pointcut = new AspectJExpressionPointcut(expression);
        }
        return pointcut;
    }

    @Override
    public Advice getAdvice() {
        return advice;
    }

    public void setAdvice(Advice advice){
        this.advice = advice;
```

```
    }

}
```

- AspectJExpressionPointcutAdvisor 实现了 PointcutAdvisor 接口，把切面 pointcut、拦截方法 advice 和具体的拦截表达式包装在一起。这样就可以在 xml 的配置中定义一个 pointcutAdvisor 切面拦截器了。

4. 方法拦截器

cn.bugstack.springframework.aop.framework.adapter.MethodBeforeAdviceInterceptor

```
public class MethodBeforeAdviceInterceptor implements MethodInterceptor {

    private MethodBeforeAdvice advice;

    public MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {
        this.advice = advice;
    }

    @Override
    public Object invoke(MethodInvocation methodInvocation) throws Throwable {
        this.advice.before(methodInvocation.getMethod(), methodInvocation.getArgume
nts(), methodInvocation.getThis());
        return methodInvocation.proceed();
    }

}
```

- MethodBeforeAdviceInterceptor 实现了 MethodInterceptor 接口，在 invoke 方法中调用 advice 中的 before 方法，传入对应的参数信息。
- 而这个 advice.before 则是用于自己实现 MethodBeforeAdvice 接口后做的相应处理。*其实可以看到具体的 MethodInterceptor 实现类，其实和我们之前做的测试是一样的，只不过现在交给了 Spring 来处理*

5. 代理工厂

cn.bugstack.springframework.aop.framework.ProxyFactory

```
public class ProxyFactory {
```

```
    private AdvisedSupport advisedSupport;

    public ProxyFactory(AdvisedSupport advisedSupport) {
        this.advisedSupport = advisedSupport;
    }

    public Object getProxy() {
        return createAopProxy().getProxy();
    }

    private AopProxy createAopProxy() {
        if (advisedSupport.isProxyTargetClass()) {
            return new Cglib2AopProxy(advisedSupport);
        }

        return new JdkDynamicAopProxy(advisedSupport);
    }

}
```

- 其实这个代理工厂主要解决的是关于 JDK 和 Cglib 两种代理的选择问题，有了代理工厂就可以按照不同的创建需求进行控制。

6. 融入 Bean 生命周期的自动代理创建者

cn.bugstack.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator

```
public class DefaultAdvisorAutoProxyCreator implements InstantiationAwareBeanPostProcessor, BeanFactoryAware {

    private DefaultListableBeanFactory beanFactory;

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        this.beanFactory = (DefaultListableBeanFactory) beanFactory;
    }

    @Override
    public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException {

        if (isInfrastructureClass(beanClass)) return null;
```

```
        Collection<AspectJExpressionPointcutAdvisor> advisors = beanFactory.getBean
sOfType(AspectJExpressionPointcutAdvisor.class).values();

        for (AspectJExpressionPointcutAdvisor advisor : advisors) {
            ClassFilter classFilter = advisor.getPointcut().getClassFilter();
            if (!classFilter.matches(beanClass)) continue;

            AdvisedSupport advisedSupport = new AdvisedSupport();

            TargetSource targetSource = null;
            try {
                targetSource = new TargetSource(beanClass.getDeclaredConstructor().
newInstance());
            } catch (Exception e) {
                e.printStackTrace();
            }
            advisedSupport.setTargetSource(targetSource);
            advisedSupport.setMethodInterceptor((MethodInterceptor) advisor.getAdvi
ce());
            advisedSupport.setMethodMatcher(advisor.getPointcut().getMethodMatcher(
));
            advisedSupport.setProxyTargetClass(false);

            return new ProxyFactory(advisedSupport).getProxy();

        }

        return null;
    }

}
```

- 这个 DefaultAdvisorAutoProxyCreator 类的主要核心实现在于 postProcessBeforeInstantiation 方法中，从通过 beanFactory.getBeansOfType 获取 AspectJExpressionPointcutAdvisor 开始。
- 获取了 advisors 以后就可以遍历相应的 AspectJExpressionPointcutAdvisor 填充对应的属性信息，包括：目标对象、拦截方法、匹配器，之后返回代理对象即可。
- 那么现在调用方获取到的这个 Bean 对象就是一个已经被切面注入的对象了，当调用方法的时候，则会被按需拦截，处理用户需要的信息。

五、测试

1. 事先准备

```
public class UserService implements IUserService {

    public String queryUserInfo() {
        try {
            Thread.sleep(new Random(1).nextInt(100));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "小傅哥，100001，深圳";
    }

    public String register(String userName) {
        try {
            Thread.sleep(new Random(1).nextInt(100));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "注册用户：" + userName + " success！";
    }

}
```

- 在 UserService 中提供了 2 个不同方法，另外你还可以增加新的类来加入测试。后面我们的测试过程，会给这个两个方法添加我们的拦截处理，打印方法执行耗时。

2. 自定义拦截方法

```
public class UserServiceBeforeAdvice implements MethodBeforeAdvice {

    @Override
    public void before(Method method, Object[] args, Object target) throws Throwabl
e {
        System.out.println("拦截方法：" + method.getName());
    }

}
```

- 与上一章节的拦截方法相比，我们不在是实现 MethodInterceptor 接口，而是实现 MethodBeforeAdvice 环绕拦截。在这个方法中我们可以获取到方法的一些信息，如果还开发了它的 MethodAfterAdvice 则可以两个接口一起实现。

3. spring.xml 配置 AOP

```
<beans>

    <bean id="userService" class="cn.bugstack.springframework.test.bean.UserService
"/>

    <bean class="cn.bugstack.springframework.aop.framework.autoproxy.DefaultAdvisor
AutoProxyCreator"/>

    <bean id="beforeAdvice" class="cn.bugstack.springframework.test.bean.UserServic
eBeforeAdvice"/>

    <bean id="methodInterceptor" class="cn.bugstack.springframework.aop.framework.a
dapter.MethodBeforeAdviceInterceptor">
        <property name="advice" ref="beforeAdvice"/>
    </bean>

    <bean id="pointcutAdvisor" class="cn.bugstack.springframework.aop.aspectj.Aspec
tJExpressionPointcutAdvisor">
        <property name="expression" value="execution(* cn.bugstack.springframework.
test.bean.IUserService.*(..))"/>
        <property name="advice" ref="methodInterceptor"/>
    </bean>

</beans>
```

- 这回再使用 AOP 就可以像 Spring 中一样，通过在 xml 中配置即可。因为我们已经把 AOP 的功能融合到 Bean 的生命周期里去了，你的新增拦截方法都会被自动处理。

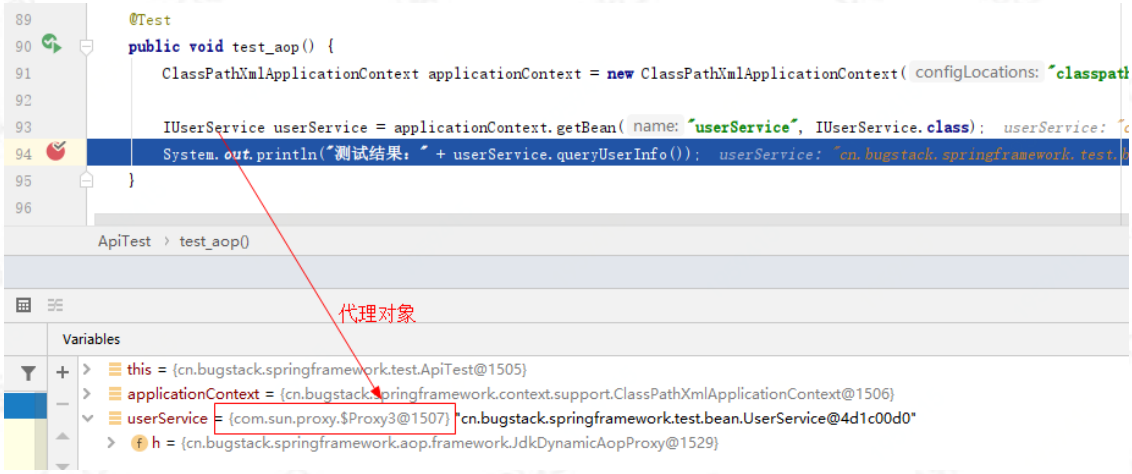
4. 单元测试

```
@Test
public void test_aop() {
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplication
Context("classpath:spring.xml");
    IUserService userService = applicationContext.getBean("userService", IUserServi
```

```
ce.class);
    System.out.println("测试结果：" + userService.queryUserInfo());
}
```

- 在单元测试中你只需要按照正常获取和使用 Bean 对象即可，不过这个时候如果被切面拦截了，那么其实你获取到的就是对应的代理对象里面的处理操作了。

测试结果




```
拦截方法：queryUserInfo
测试结果：小傅哥，100001，深圳

Process finished with exit code 0
```

- 通过测试结果可以看到，我们已经让拦截方法生效了，也不需要自己手动处理切面、拦截方法等内容。*截图上可以看到，这个时候的 IUserService 就是一个代理对象*

六、总结

- 本章节实现 AOP 功能的外在体现主要是把以前自己在单元测试中的切面拦截，交给 Spring 的 xml 配置了，也就不需要自己手动处理了。那么这里有一个非常重要的知识点，就是把相应的功能如何与 Spring 的 Bean 生命周期结合起来，本章节用到的 BeanPostProcessor，因为它可以解决在 Bean 对象执行初始化方法之前，用于修改新实例化 Bean 对象的扩展点，所以我们也就可以处理自己的 AOP 代理对象逻辑了。
- 一个功能的实现往往包括核心部分、组装部分、链接部分，为了这些各自职责的分工，则需要创建接口和类，由不同关系的继承、实现进行组装。只有明确了各个职责分工，才好灵活的扩展相应的功能逻辑，否则很难驾驭大型系统的开发和建设，也就是那种不好把握的感觉。

- 目前我们实现的 AOP 与 Spring 源码中的核心逻辑是类似的，但更会偏简单一些，也不会考虑更多的复杂场景遇到的问题，包括是否有构造函数、是否为代理中的切面等。其实也可以看出只要是 Java 中的一些特性，都需要在真实使用的 Spring 中进行完整的实现，否则在使用这些功能的时候就会遇到各种问题。

第 14 章：自动扫描 Bean 对象注册

一、简化使用

式复杂，没等搞明白大促都过去了！

你经历过 618 和双 11 吗？你加入过大促时候那么多复杂的营销活动赚几毛钱吗？你开发过连读明白玩法都需要一周但只使用 3 天的大促需求吗？有时候对于有些产品的需求真的是太复杂了，复杂到开发、测试都需要在整个过程中不断的学习最后才可能读懂产品为啥这样的玩，要是一个长期的活动可能也就算了，培养用户心智吗！但这一整套拉新、助力、激活、下单、投保、领券、消费、开红包等等一连串的骚操作下来，如果在线上只用 3 天呢，或者是只用 1 天，那 TM 连参与的用户都没弄明白呢，活动就结束了，最后能打来什么样好的数据呢？*对于这样流程复杂，估计连羊毛当都看不上！！！*

以上只是举个例子，大部分时候并不会搞的这么恶心，评审也是过不去的！而同样的道理用在程序设计开发和使用中也是一样的，如果你把你的代码逻辑实现的过于分散，让外部调用方在使用的时候，需要调用你的接口多个和多次，还没有消息触达，只能定时自己轮训你的接口查看订单状态，每次还只能查 10 条，查多了你说不行，等等反人类的设计，都会给调用方带来要干你的体会。

所以，如果我们能在完成目的的情况下，都是希望尽可能流程简单、模式清晰、自动服务。那这在 Spring 的框架中也是有所体现的，这个框架的普及使用程度和它所能带来的方便性是分不开的，而我们如果能做到如此的方便，那肯定是一种好的设计和实现。

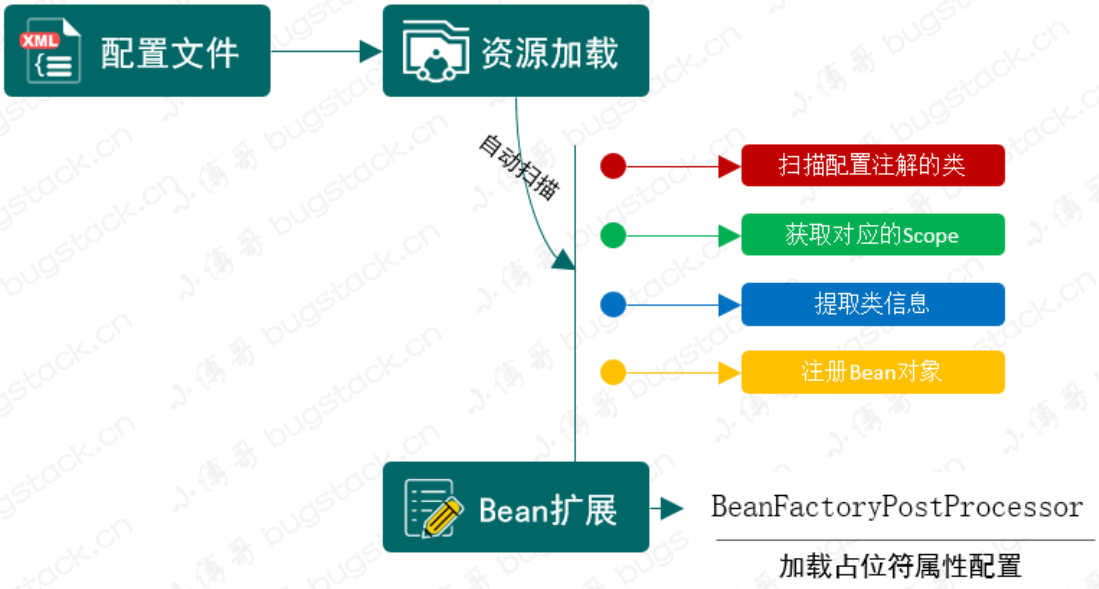
二、目标

其实到本章节我们已经把关于 IOC 和 AOP 全部核心内容都已经实现完成了，只不过在使用上还有点像早期的 Spring 版本，需要一个一个在 spring.xml 中进行配置。这与实际的目前使用的 Spring 框架还是有蛮大的差别，而这种差别其实都是在核心功能逻辑之上建设的在更少的配置下，做到更简化的使用。

这其中就包括：包的扫描注册、注解配置的使用、占位符属性的填充等等，而我们的目标就是在目前的核心逻辑上填充一些自动化的功能，让大家可以学习到这部分的设计和实现，从中体会到一些关于代码逻辑的实现过程，总结一些编码经验。

三、方案

首先我们要考虑，为了可以简化 Bean 对象的配置，让整个 Bean 对象的注册都是自动扫描的，那么基本需要的元素包括：扫描路径入口、XML 解析扫描信息、给需要扫描的 Bean 对象做注解标记、扫描 Class 对象摘取 Bean 注册的基本信息，组装注册信息、注册成 Bean 对象。那么在这些条件元素的支撑下，就可以实现出通过自定义注解和配置扫描路径的情况下，完成 Bean 对象的注册。除此之外再顺带解决一个配置中占位符属性的知识点，比如可以通过 `${token}` 给 Bean 对象注入进去属性信息，那么这个操作需要用到 BeanFactoryPostProcessor，因为它可以处理 **在所有的 BeanDefinition 加载完成后，实例化 Bean 对象之前，提供修改 BeanDefinition 属性的机制** 而实现这部分内容是为了后续把此类内容结合到自动化配置处理中。整体设计结构如下图：



结合 bean 的生命周期，包扫描只不过是扫描特定注解的类，提取类的相关信息组装成 BeanDefinition 注册到容器中。

在 XmlBeanDefinitionReader 中解析`<context:component-scan />`标签，扫描类组装 BeanDefinition 然后注册到容器中的操作在 ClassPathBeanDefinitionScanner#doScan 中实现。

- 自动扫描注册主要是扫描添加了自定义注解的类，在 xml 加载过程中提取类的信息，组装 BeanDefinition 注册到 Spring 容器中。
- 所以我们会用到 `<context:component-scan />` 配置包路径并在 XmlBeanDefinitionReader 解析并做相应的处理。*这里的处理会包括对类的扫描、获取注解信息等*
- 最后还包括了一部分关于 `BeanFactoryPostProcessor` 的使用，因为我们需要完成对占位符配置信息的加载，所以需要使用到 BeanFactoryPostProcessor 在所有的 BeanDefinition 加载完成后，实例化 Bean 对象之前，修改 BeanDefinition 的属性信息。*这一部分的实现也为后续处理关于占位符配置到注解上做准备*

四、实现

1. 工程结构

```
small-spring-step-13
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── aop
    │           │   ├── aspectj
    │           │   │   └── AspectJExpressionPointcut.java
    │           │   │   └── AspectJExpressionPointcutAdvisor.java
    │           │   ├── framework
    │           │   │   ├── adapter
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── autoproxy
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── AopProxy.java
    │           │   │   ├── Cglib2AopProxy.java
    │           │   │   ├── JdkDynamicAopProxy.java
    │           │   │   ├── ProxyFactory.java
    │           │   │   └── ReflectiveMethodInvocation.java
    │           │   ├── AdvisedSupport.java
    │           │   ├── Advisor.java
    │           │   ├── BeforeAdvice.java
    │           │   ├── ClassFilter.java
    │           │   ├── MethodBeforeAdvice.java
    │           │   ├── MethodMatcher.java
    │           │   ├── Pointcut.java
    │           │   ├── PointcutAdvisor.java
```

```
    │           │   └── TargetSource.java
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   │   ├── BeanPostProcessor.java
    │           │   │   │   ├── BeanReference.java
    │           │   │   │   ├── ConfigurableBeanFactory.java
    │           │   │   │   ├── InstantiationAwareBeanPostProcessor.java
    │           │   │   │   └── SingletonBeanRegistry.java
    │           │   │   ├── support
    │           │   │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   │   ├── AbstractBeanFactory.java
    │           │   │   │   ├── BeanDefinitionReader.java
    │           │   │   │   ├── BeanDefinitionRegistry.java
    │           │   │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   │   ├── DefaultListableBeanFactory.java
    │           │   │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   │   ├── DisposableBeanAdapter.java
    │           │   │   │   ├── FactoryBeanRegistrySupport.java
    │           │   │   │   ├── InstantiationStrategy.java
    │           │   │   │   └── SimpleInstantiationStrategy.java
    │           │   │   ├── support
    │           │   │   │   └── XmlBeanDefinitionReader.java
    │           │   │   ├── Aware.java
    │           │   │   ├── BeanClassLoaderAware.java
    │           │   │   ├── BeanFactory.java
    │           │   │   ├── BeanFactoryAware.java
    │           │   │   ├── BeanNameAware.java
    │           │   │   ├── ConfigurableListableBeanFactory.java
    │           │   │   ├── DisposableBean.java
    │           │   │   ├── FactoryBean.java
    │           │   │   ├── HierarchicalBeanFactory.java
    │           │   │   ├── InitializingBean.java
    │           │   │   ├── ListableBeanFactory.java
    │           │   │   └── PropertyPlaceholderConfigurer.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
    │           │   └── PropertyValues.java
    │           ├── context
    │           │   ├── annotation
```

```
    │           │   │   ├── ClassPathBeanDefinitionScanner.java
    │           │   │   ├── ClassPathScanningCandidateComponentProvider.java
    │           │   │   └── Scope.java
    │           │   ├── event
    │           │   │   ├── AbstractApplicationEventMulticaster.java
    │           │   │   ├── ApplicationContextEvent.java
    │           │   │   ├── ApplicationEventMulticaster.java
    │           │   │   ├── ContextClosedEvent.java
    │           │   │   ├── ContextRefreshedEvent.java
    │           │   │   └── SimpleApplicationEventMulticaster.java
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   ├── ApplicationContextAwareProcessor.java
    │           │   │   └── ClassPathXmlApplicationContext.java
    │           │   ├── ApplicationContext.java
    │           │   ├── ApplicationContextAware.java
    │           │   ├── ApplicationEvent.java
    │           │   ├── ApplicationEventPublisher.java
    │           │   ├── ApplicationListener.java
    │           │   └── ConfigurableApplicationContext.java
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           ├── stereotype
    │           │   └── Component.java
    │           └── utils
    │               └── ClassUtils.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── IUserService.java
                │   └── UserService.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

在 Bean 的生命周期中自动加载包扫描注册 Bean 对象和设置占位符属性的类关系，如图 14-2

图 14-2

- 整个类的关系结构来看，其实涉及的内容并不多，主要包括的就是 xml 解析类 XmlBeanDefinitionReader 对 ClassPathBeanDefinitionScanner#doScan 的使用。
- 在 doScan 方法中处理所有指定路径下添加了注解的类，拆解出类的信息：名称、作用范围等，进行创建 BeanDefinition 好用于 Bean 对象的注册操作。
- PropertyPlaceholderConfigurer 目前看上去像一块单独的内容，后续会把这块的内容与自动加载 Bean 对象进行整合，也就是可以在注解上使用占位符配置一些在配置文件里的属性信息。

2. 处理占位符配置

cn.bugstack.springframework.beans.factory.PropertyPlaceholderConfigurer

```
public class PropertyPlaceholderConfigurer implements BeanFactoryPostProcessor {

    /**
     * Default placeholder prefix: {@value}
     */
    public static final String DEFAULT_PLACEHOLDER_PREFIX = "${";

    /**
     * Default placeholder suffix: {@value}
     */
    public static final String DEFAULT_PLACEHOLDER_SUFFIX = "}";

    private String location;

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
 throws BeansException {
        // 加载属性文件
        try {
            DefaultResourceLoader resourceLoader = new DefaultResourceLoader();
            Resource resource = resourceLoader.getResource(location);
            Properties properties = new Properties();
            properties.load(resource.getInputStream());

            String[] beanDefinitionNames = beanFactory.getBeanDefinitionNames();
            for (String beanName : beanDefinitionNames) {
                BeanDefinition beanDefinition = beanFactory.getBeanDefinition(beanN
ame);

                PropertyValues propertyValues = beanDefinition.getPropertyValues();
                for (PropertyValue propertyValue : propertyValues.getPropertyValues
()) {
                    Object value = propertyValue.getValue();
                    if (!(value instanceof String)) continue;
                    String strVal = (String) value;
                    StringBuilder buffer = new StringBuilder(strVal);
                    int startIdx = strVal.indexOf(DEFAULT_PLACEHOLDER_PREFIX);
                    int stopIdx = strVal.indexOf(DEFAULT_PLACEHOLDER_SUFFIX);
                    if (startIdx != -1 && stopIdx != -1 && startIdx < stopIdx) {
                        String propKey = strVal.substring(startIdx + 2, stopIdx);
                        String propVal = properties.getProperty(propKey);
                        buffer.replace(startIdx, stopIdx + 1, propVal);
                        propertyValues.addPropertyValue(new PropertyValue(propertyV
alue.getName(), buffer.toString()));
```

```
                    }
                }
            }
        } catch (IOException e) {
            throw new BeansException("Could not load properties", e);
        }
    }

    public void setLocation(String location) {
        this.location = location;
    }

}
```

- 依赖于 BeanFactoryPostProcessor 在 Bean 生命周期的属性，可以在 Bean 对象实例化之前，改变属性信息。所以这里通过实现 BeanFactoryPostProcessor 接口，完成对配置文件的加载以及摘取占位符中的在属性文件里的配置。
- 这样就可以把提取到的配置信息放置到属性配置中了，`buffer.replace(startIdx, stopIdx + 1, propVal);` `propertyValues.addPropertyValue`

3. 定义拦截注解

cn.bugstack.springframework.context.annotation.Scope

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Scope {

    String value() default "singleton";

}
```

- 用于配置作用域的自定义注解，方便通过配置 Bean 对象注解的时候，拿到 Bean 对象的作用域。*不过一般都使用默认的 singleton*

cn.bugstack.springframework.stereotype.Component

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Component {
```

```
    String value() default "";

}
```

- Component 自定义注解大家都非常熟悉了，用于配置到 Class 类上的。除此之外还有 Service、Controller，不过所有的处理方式基本一致，这里就只展示一个 Component 即可。

4. 处理对象扫描装配

cn.bugstack.springframework.context.annotation.ClassPathScanningCandidateComponentProvider

```
public class ClassPathScanningCandidateComponentProvider {

    public Set<BeanDefinition> findCandidateComponents(String basePackage) {
        Set<BeanDefinition> candidates = new LinkedHashSet<>();
        Set<Class<?>> classes = ClassUtil.scanPackageByAnnotation(basePackage, Comp
onent.class);
        for (Class<?> clazz : classes) {
            candidates.add(new BeanDefinition(clazz));
        }
        return candidates;
    }

}
```

- 这里先要提供一个可以通过配置路径 basePackage=cn.bugstack.springframework.test.bean，解析出 classes 信息的工具方法 findCandidateComponents，通过这个方法就可以扫描到所有 @Component 注解的 Bean 对象了。

cn.bugstack.springframework.context.annotation.ClassPathBeanDefinitionScanner

```
public class ClassPathBeanDefinitionScanner extends ClassPathScanningCandidateCompo
nentProvider {

    private BeanDefinitionRegistry registry;

    public ClassPathBeanDefinitionScanner(BeanDefinitionRegistry registry) {
        this.registry = registry;
    }
```

```
    public void doScan(String... basePackages) {
        for (String basePackage : basePackages) {
            Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
            for (BeanDefinition beanDefinition : candidates) {
                // 解析 Bean 的作用域 singleton、prototype
                String beanScope = resolveBeanScope(beanDefinition);
                if (StrUtil.isNotEmpty(beanScope)) {
                    beanDefinition.setScope(beanScope);
                }
                registry.registerBeanDefinition(determineBeanName(beanDefinition),
beanDefinition);
            }
        }
    }

    private String resolveBeanScope(BeanDefinition beanDefinition) {
        Class<?> beanClass = beanDefinition.getBeanClass();
        Scope scope = beanClass.getAnnotation(Scope.class);
        if (null != scope) return scope.value();
        return StrUtil.EMPTY;
    }

    private String determineBeanName(BeanDefinition beanDefinition) {
        Class<?> beanClass = beanDefinition.getBeanClass();
        Component component = beanClass.getAnnotation(Component.class);
        String value = component.value();
        if (StrUtil.isEmpty(value)) {
            value = StrUtil.lowerFirst(beanClass.getSimpleName());
        }
        return value;
    }

}
```

- ClassPathBeanDefinitionScanner 是继承自 ClassPathScanningCandidateComponentProvider 的具体扫描包处理的类，在 doScan 中除了获取到扫描的类信息以后，还需要获取 Bean 的作用域和类名，如果不配置类名基本都是把首字母缩写。

5. 解析 xml 中调用扫描

cn.bugstack.springframework.beans.factory.xml.XmlBeanDefinitionReader

```
public class XmlBeanDefinitionReader extends AbstractBeanDefinitionReader {

    protected void doLoadBeanDefinitions(InputStream inputStream) throws ClassNotFoundException, DocumentException {
        SAXReader reader = new SAXReader();
        Document document = reader.read(inputStream);
        Element root = document.getRootElement();

        // 解析 context:component-scan 标签，扫描包中的类并提取相关信息，用于组装 BeanDefinition
        Element componentScan = root.element("component-scan");
        if (null != componentScan) {
            String scanPath = componentScan.attributeValue("base-package");
            if (StrUtil.isEmpty(scanPath)) {
                throw new BeansException("The value of base-package attribute can not be empty or null");
            }
            scanPackage(scanPath);
        }

        // ... 省略其他

        // 注册 BeanDefinition
        getRegistry().registerBeanDefinition(beanName, beanDefinition);
    }

    private void scanPackage(String scanPath) {
        String[] basePackages = StrUtil.splitToArray(scanPath, ',');
        ClassPathBeanDefinitionScanner scanner = new ClassPathBeanDefinitionScanner(getRegistry());
        scanner.doScan(basePackages);
    }

}
```

- 关于 XmlBeanDefinitionReader 中主要是在加载配置文件后，处理新增的自定义配置属性 component-scan，解析后调用 scanPackage 方法，其实也就是我们在 ClassPathBeanDefinitionScanner#doScan 功能。

- 另外这里需要注意，为了可以方便的加载和解析 xml，XmlBeanDefinitionReader 已经全部替换为 dom4j 的方式进行解析处理。

五、测试

1. 事先准备

```
@Component("userService")
public class UserService implements IUserService {

    private String token;

    public String queryUserInfo() {
        try {
            Thread.sleep(new Random(1).nextInt(100));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "小傅哥，100001，深圳";
    }

    public String register(String userName) {
        try {
            Thread.sleep(new Random(1).nextInt(100));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "注册用户：" + userName + " success! ";
    }

    @Override
    public String toString() {
        return "UserService#token = { " + token + " }";
    }

    public String getToken() {
        return token;
    }

    public void setToken(String token) {
        this.token = token;
```

```
    }
}
```

- 给 UserService 类添加一个自定义注解 `@Component("userService")` 和一个属性信息 `String token`。这是为了分别测试包扫描和占位符属性。

2. 属性配置文件

```
token=RejDlI78hu223Opo983Ds
```

- 这里配置一个 token 的属性信息，用于通过占位符的方式进行获取

3. spring.xml 配置对象

spring-property.xml

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

    <bean class="cn.bugstack.springframework.beans.factory.PropertyPlaceholderConfi
gurer">
        <property name="location" value="classpath:token.properties"/>
    </bean>

    <bean id="userService" class="cn.bugstack.springframework.test.bean.UserService
">
        <property name="token" value="${token}"/>
    </bean>

</beans>
```

- 加载 `classpath:token.properties` 设置占位符属性值 `${token}`

spring-scan.xml

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

```
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/context">

    <context:component-scan base-package="cn.bugstack.springframework.test.bean"/>

</beans>
```

- 添加 component-scan 属性，设置包扫描根路径

4. 单元测试(占位符)

```
@Test
public void test_property() {
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplication
Context("classpath:spring-property.xml");
    IUserService userService = applicationContext.getBean("userService", IUserServi
ce.class);
    System.out.println("测试结果：" + userService);
}
```

测试结果

```
测试结果：UserService#token = { RejDlI78hu223Opo983Ds }

Process finished with exit code 0
```

- 通过测试结果可以看到 UserService 中的 token 属性已经通过占位符的方式设置进去配置文件里的 token.properties 的属性值了。

5. 单元测试(包扫描)

```
@Test
public void test_scan() {
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplication
Context("classpath:spring-scan.xml");
    IUserService userService = applicationContext.getBean("userService", IUserServi
ce.class);
    System.out.println("测试结果：" + userService.queryUserInfo());
}
```

测试结果

```
测试结果：小傅哥，100001，深圳

Process finished with exit code 0
```

- 通过这个测试结果可以看出来，现在使用注解的方式就可以让 Class 注册完成 Bean 对象了。

六、总结

- 通过整篇的内容实现可以看出来，目前的功能添加其实已经不复杂了，都是在 IOC 和 AOP 核心的基础上来补全功能。这些补全的功能也是在完善 Bean 的生命周期，让整个功能使用也越来越容易。
- 在你不断的实现着 Spring 的各项功能时，也可以把自己在平常使用 Spring 的一些功能想法融入进来，比如像 Spring 是如何动态切换数据源的，线程池是怎么提供配置的，这些内容虽然不是最基础的核心范围，但也非常重要。
- 可能有些时候这些类实现的内容对新人来说比较多，可以一点点动手实现逐步理解，在把一些稍微较有难度的内容实现后，其实后面也就没有那么难理解了。

第 15 章：通过注解注入属性信息

一、逻辑易用

写代码，就是从能用到好用的不断折腾！

你听过扰动函数吗？你写过斐波那契（Fibonacci）散列吗？你实现过梅森旋转算法吗？**怎么** 没听过这些写不了代码吗！不会的，即使没听过你一样可以写的了代码，比如你实现的数据库路由数据总是落在 1 库 1 表它不散列分布、你实现的抽奖系统总是把运营配置的最大红包发出去提高了运营成本、你开发的秒杀系统总是在开始后的 1 秒就挂了货品根本给不出去。

除了一部分仅把编码当成搬砖应付工作外的程序员，还有一部分总是在追求极致的码农。*写代码还能赚钱，真开心！* 这样的码农总是会考虑　还有没有更好的实现逻辑能让代码不仅是能用，还要好用呢？其实这一点的追求到完成，需要大量扩展性学习和深度挖掘，这样你设计出来的系统才更你考虑的更加全面，也能应对各种复杂的场景。

二、目标

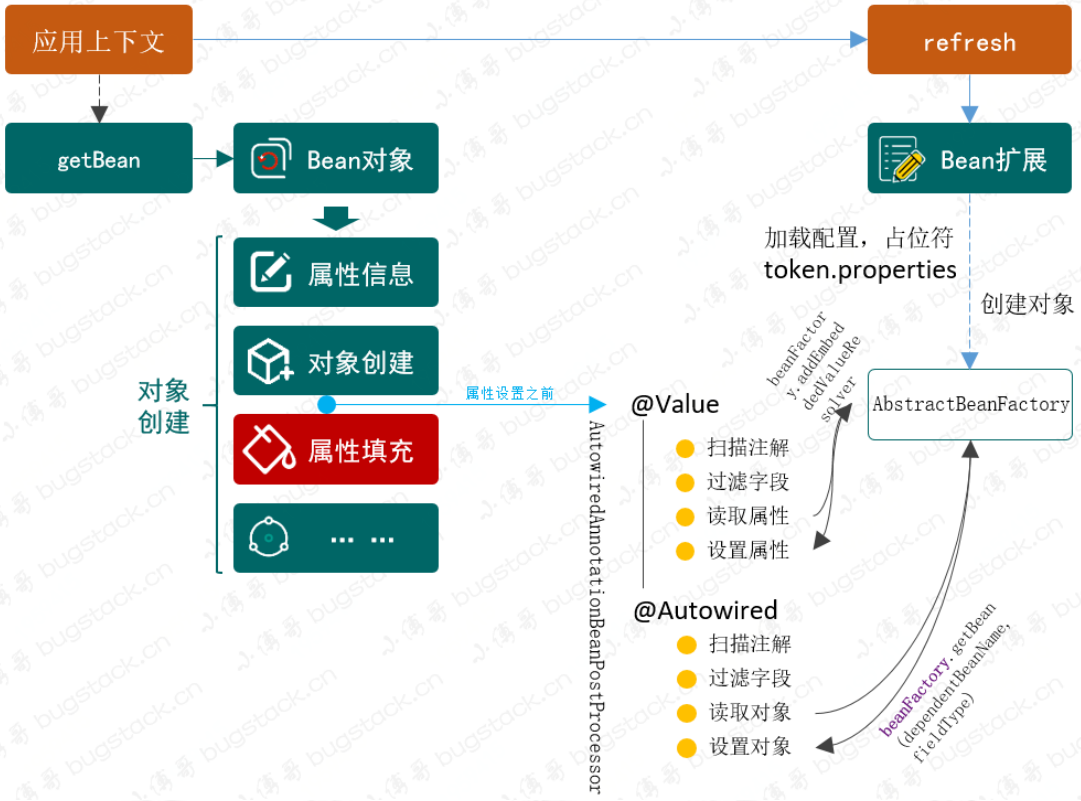
在目前 IOC、AOP 两大核心功能模块的支撑下，完全可以管理 Bean 对象的注册和获取，不过这样的使用方式总感觉像是刀耕火种有点难用。因此在上一章节我们解决需要手动配置 `Bean` 对象到 `spring.xml` 文件中，改为可以自动扫描带有注解 `@Component` 的对象完成自动装配和注册到 `Spring` 容器的操作。

那么在自动扫描包注册 Bean 对象之后，就需要把原来在配置文件中通过 `property name="token"` 配置属性和 Bean 的操作，也改为可以自动注入。这就像我们使用 Spring 框架中 `@Autowired`、`@Value` 注解一样，完成我们对属性和对象的注入操作。

三、方案

其实从我们在完成 Bean 对象的基础功能后，后续陆续添加的功能都是围绕着 Bean 的生命周期进行的，比如修改 Bean 的定义

BeanFactoryPostProcessor，处理 Bean 的属性要用到 BeanPostProcessor，完成个性的属性操作则专门继承 BeanPostProcessor 提供新的接口，因为这样才能通过 instanceof 判断出具有标记性的接口。所以关于 Bean 等等的操作，以及监听 Aware、获取 BeanFactory，都需要在 Bean 的生命周期中完成。那么我们在设计属性和 Bean 对象的注入时候，也会用到 BeanPostProcessor 来完成在设置 Bean 属性之前，允许 BeanPostProcessor 修改属性值。整体设计结构如下图：



- 要处理自动扫描注入，包括属性注入、对象注入，则需要在对象属性 `applyPropertyValues` 填充之前，把属性信息写入到 PropertyValues 的集合中去。这一步的操作相当于是解决了以前在 spring.xml 配置属性的过程。
- 而在属性的读取中，需要依赖于对 Bean 对象的类中属性的配置了注解的扫描，`field.getAnnotation(Value.class);` 依次拿出符合的属性并填充上相应的配置信息。*这里有一点，属性的配置信息需要依赖于 BeanFactoryPostProcessor 的实现类 PropertyPlaceholderConfigurer，把值写入到 AbstractBeanFactory 的 embeddedValueResolvers 集合中，这样才能在属性填充中利用 beanFactory 获取相应的属性值*
- 还有一个是关于 @Autowired 对于对象的注入，其实这一个和属性注入的唯一区别是对于对象的获取 `beanFactory.getBean(fieldType)`，其他就没有什么差一点了。
- 当所有的属性被设置到 PropertyValues 完成以后，接下来就到了创建对象的下一步，属性填充，而此时就会把我们一一获取到的配置和对象填充到属性上，也就实现了自动注入的功能。

四、实现

1. 工程结构

```
small-spring-step-14
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── aop
    │           │   ├── aspectj
    │           │   │   └── AspectJExpressionPointcut.java
    │           │   │   └── AspectJExpressionPointcutAdvisor.java
    │           │   ├── framework
    │           │   │   ├── adapter
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── autoproxy
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── AopProxy.java
    │           │   │   ├── Cglib2AopProxy.java
    │           │   │   ├── JdkDynamicAopProxy.java
    │           │   │   ├── ProxyFactory.java
    │           │   │   └── ReflectiveMethodInvocation.java
    │           │   ├── AdvisedSupport.java
    │           │   ├── Advisor.java
    │           │   ├── BeforeAdvice.java
    │           │   ├── ClassFilter.java
    │           │   ├── MethodBeforeAdvice.java
    │           │   ├── MethodMatcher.java
    │           │   ├── Pointcut.java
    │           │   ├── PointcutAdvisor.java
    │           │   └── TargetSource.java
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── annotation
    │           │   │   │   ├── Autowired.java
    │           │   │   │   ├── AutowiredAnnotationBeanPostProcessor.java
    │           │   │   │   ├── Qualifier.java
    │           │   │   │   └── Value.java
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
```

```
    │           │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   ├── BeanPostProcessor.java
    │           │   │   ├── BeanReference.java
    │           │   │   ├── ConfigurableBeanFactory.java
    │           │   │   ├── InstantiationAwareBeanPostProcessor.java
    │           │   │   └── SingletonBeanRegistry.java
    │           │   ├── support
    │           │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   ├── AbstractBeanFactory.java
    │           │   │   ├── BeanDefinitionReader.java
    │           │   │   ├── BeanDefinitionRegistry.java
    │           │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   ├── DefaultListableBeanFactory.java
    │           │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   ├── DisposableBeanAdapter.java
    │           │   │   ├── FactoryBeanRegistrySupport.java
    │           │   │   ├── InstantiationStrategy.java
    │           │   │   └── SimpleInstantiationStrategy.java
    │           │   ├── support
    │           │   │   └── XmlBeanDefinitionReader.java
    │           │   ├── Aware.java
    │           │   ├── BeanClassLoaderAware.java
    │           │   ├── BeanFactory.java
    │           │   ├── BeanFactoryAware.java
    │           │   ├── BeanNameAware.java
    │           │   ├── ConfigurableListableBeanFactory.java
    │           │   ├── DisposableBean.java
    │           │   ├── FactoryBean.java
    │           │   ├── HierarchicalBeanFactory.java
    │           │   ├── InitializingBean.java
    │           │   ├── ListableBeanFactory.java
    │           │   └── PropertyPlaceholderConfigurer.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
    │           │   └── PropertyValues.java
    │           ├── context
    │           │   ├── annotation
    │           │   │   ├── ClassPathBeanDefinitionScanner.java
    │           │   │   ├── ClassPathScanningCandidateComponentProvider.java
    │           │   │   └── Scope.java
    │           │   ├── event
    │           │   │   ├── AbstractApplicationEventMulticaster.java
    │           │   │   ├── ApplicationContextEvent.java
```

```
    │           │   │   ├── ApplicationEventMulticaster.java
    │           │   │   ├── ContextClosedEvent.java
    │           │   │   ├── ContextRefreshedEvent.java
    │           │   │   └── SimpleApplicationEventMulticaster.java
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   ├── ApplicationContextAwareProcessor.java
    │           │   │   └── ClassPathXmlApplicationContext.java
    │           │   ├── ApplicationContext.java
    │           │   ├── ApplicationContextAware.java
    │           │   ├── ApplicationEvent.java
    │           │   ├── ApplicationEventPublisher.java
    │           │   ├── ApplicationListener.java
    │           │   └── ConfigurableApplicationContext.java
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           ├── stereotype
    │           │   └── Component.java
    │           └── utils
    │               ├── ClassUtils.java
    │               └── StringValueResolver.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── IUserService.java
                │   └── UserService.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

自动扫描注入占位符配置和对象的类关系，如图 15-2

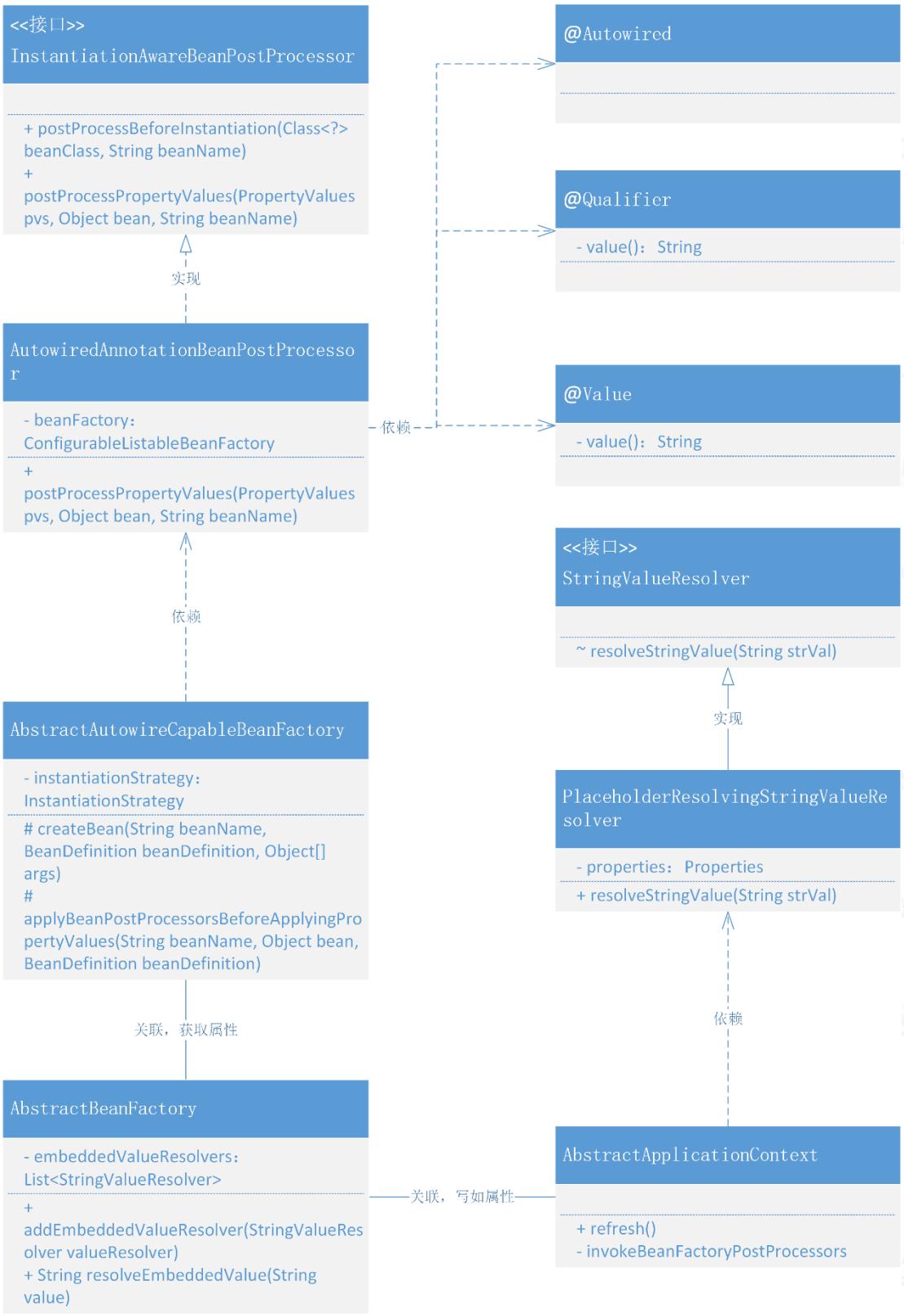


图 15-2

- 在整个类图中以围绕实现接口 InstantiationAwareBeanPostProcessor 的类 AutowiredAnnotationBeanPostProcessor 作为入口点，被

AbstractAutowireCapableBeanFactory 创建 Bean 对象过程中调用扫描整个类的属性配置中含有自定义注解 Value、Autowired、Qualifier，的属性值。

- 这里稍有变动的是关于属性值信息的获取，在注解配置的属性字段扫描到信息注入时，包括了占位符从配置文件获取信息也包括 Bean 对象，Bean 对象可以直接获取，但配置信息需要在 AbstractBeanFactory 中添加新的属性集合 embeddedValueResolvers，由 PropertyPlaceholderConfigurer#postProcessBeanFactory 进行操作填充到属性集合中。

2. 把读取到属性填充到容器

定义解析字符串接口

cn.bugstack.springframework.util.StringValueResolver

```
public interface StringValueResolver {

    String resolveStringValue(String strVal);

}
```

- 接口 StringValueResolver 是一个解析字符串操作的接口

填充字符串

```
public class PropertyPlaceholderConfigurer implements BeanFactoryPostProcessor {

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
 throws BeansException {
        try {
            // 加载属性文件
            DefaultResourceLoader resourceLoader = new DefaultResourceLoader();
            Resource resource = resourceLoader.getResource(location);

            // ... 占位符替换属性值、设置属性值

            // 向容器中添加字符串解析器，供解析@Value 注解使用
            StringValueResolver valueResolver = new PlaceholderResolvingStringValue
Resolver(properties);
            beanFactory.addEmbeddedValueResolver(valueResolver);

        } catch (IOException e) {
            throw new BeansException("Could not load properties", e);
        }
```

```
    }

    private class PlaceholderResolvingStringValueResolver implements StringValueRes
olver {

        private final Properties properties;

        public PlaceholderResolvingStringValueResolver(Properties properties) {
            this.properties = properties;
        }

        @Override
        public String resolveStringValue(String strVal) {
            return PropertyPlaceholderConfigurer.this.resolvePlaceholder(strVal, pr
operties);
        }

    }

}
```

- 在解析属性配置的类 PropertyPlaceholderConfigurer 中，最主要的其实就是这行代码的操作 `beanFactory.addEmbeddedValueResolver(valueResolver)` 这是把属性值写入到了 AbstractBeanFactory 的 embeddedValueResolvers 中。
- 这里说明下，embeddedValueResolvers 是 AbstractBeanFactory 类新增加的集合 `List<StringValueResolver> embeddedValueResolvers` String resolvers to apply e.g. to annotation attribute values

3. 自定义属性注入注解

自定义注解，Autowired、Qualifier、Value

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD, ElementType.METHOD})
public @interface Autowired {
}

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.
TYPE, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
```

```
public @interface Qualifier {

    String value() default "";

}

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Value {

    /**
     * The actual value expression: e.g. "#{systemProperties.myProp}".
     */
    String value();

}
```

- 3 个注解在我们日常使用 Spring 也是非常常见的，注入对象、注入属性，而 Qualifier 一般与 Autowired 配合使用。

4. 扫描自定义注解

cn.bugstack.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor

```
public class AutowiredAnnotationBeanPostProcessor implements InstantiationAwareBeanPostProcessor, BeanFactoryAware {

    private ConfigurableListableBeanFactory beanFactory;

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        this.beanFactory = (ConfigurableListableBeanFactory) beanFactory;
    }

    @Override
    public PropertyValues postProcessPropertyValues(PropertyValues pvs, Object bean, String beanName) throws BeansException {
        // 1. 处理注解 @Value
        Class<?> clazz = bean.getClass();
        clazz = ClassUtils.isCglibProxyClass(clazz) ? clazz.getSuperclass() : clazz;
```

```
        Field[] declaredFields = clazz.getDeclaredFields();

        for (Field field : declaredFields) {
            Value valueAnnotation = field.getAnnotation(Value.class);
            if (null != valueAnnotation) {
                String value = valueAnnotation.value();
                value = beanFactory.resolveEmbeddedValue(value);
                BeanUtil.setFieldValue(bean, field.getName(), value);
            }
        }

        // 2. 处理注解 @Autowired
        for (Field field : declaredFields) {
            Autowired autowiredAnnotation = field.getAnnotation(Autowired.class);
            if (null != autowiredAnnotation) {
                Class<?> fieldType = field.getType();
                String dependentBeanName = null;
                Qualifier qualifierAnnotation = field.getAnnotation(Qualifier.class
);
                Object dependentBean = null;
                if (null != qualifierAnnotation) {
                    dependentBeanName = qualifierAnnotation.value();
                    dependentBean = beanFactory.getBean(dependentBeanName, fieldTyp
e);
                } else {
                    dependentBean = beanFactory.getBean(fieldType);
                }
                BeanUtil.setFieldValue(bean, field.getName(), dependentBean);
            }
        }

        return pvs;
    }

}
```

- AutowiredAnnotationBeanPostProcessor 是实现接口 InstantiationAwareBeanPostProcessor 的一个用于在 Bean 对象实例化完成后，设置属性操作前的处理属性信息的类和操作方法。*只有实现了 BeanPostProcessor 接口才有机会在 Bean 的生命周期中处理初始化信息*
- 核心方法 postProcessPropertyValues，主要用于处理类含有 @Value、@Autowired 注解的属性，进行属性信息的提取和设置。

- 这里需要注意一点因为我们在 AbstractAutowireCapableBeanFactory 类中使用的是 CglibSubclassingInstantiationStrategy 进行类的创建，所以在 AutowiredAnnotationBeanPostProcessor#postProcessPropertyValues 中需要判断是否为 CGlib 创建对象，否则是不能正确拿到类信息的。

```
ClassUtils.isCglibProxyClass(clazz) ? clazz.getSuperclass() : clazz;
```

5. 在 Bean 的生命周期中调用属性注入

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactory implements AutowireCapableBeanFactory {

    private InstantiationStrategy instantiationStrategy = new CglibSubclassingInstantiationStrategy();

    @Override
    protected Object createBean(String beanName, BeanDefinition beanDefinition, Object[] args) throws BeansException {
        Object bean = null;
        try {
            // 判断是否返回代理 Bean 对象
            bean = resolveBeforeInstantiation(beanName, beanDefinition);
            if (null != bean) {
                return bean;
            }
            // 实例化 Bean
            bean = createBeanInstance(beanDefinition, beanName, args);
            // 在设置 Bean 属性之前，允许 BeanPostProcessor 修改属性值
            applyBeanPostProcessorsBeforeApplyingPropertyValues(beanName, bean, beanDefinition);
            // 给 Bean 填充属性
            applyPropertyValues(beanName, bean, beanDefinition);
            // 执行 Bean 的初始化方法和 BeanPostProcessor 的前置和后置处理方法
            bean = initializeBean(beanName, bean, beanDefinition);
        } catch (Exception e) {
            throw new BeansException("Instantiation of bean failed", e);
        }

        // 注册实现了 DisposableBean 接口的 Bean 对象
        registerDisposableBeanIfNecessary(beanName, bean, beanDefinition);
```

```
        // 判断 SCOPE_SINGLETON、SCOPE_PROTOTYPE
        if (beanDefinition.isSingleton()) {
            registerSingleton(beanName, bean);
        }
        return bean;
    }

    /**
     * 在设置 Bean 属性之前，允许 BeanPostProcessor 修改属性值
     *
     * @param beanName
     * @param bean
     * @param beanDefinition
     */
    protected void applyBeanPostProcessorsBeforeApplyingPropertyValues(String beanN
ame, Object bean, BeanDefinition beanDefinition) {
        for (BeanPostProcessor beanPostProcessor : getBeanPostProcessors()) {
            if (beanPostProcessor instanceof InstantiationAwareBeanPostProcessor){
                PropertyValues pvs = ((InstantiationAwareBeanPostProcessor) beanPos
tProcessor).postProcessPropertyValues(beanDefinition.getPropertyValues(), bean, bea
nName);
                if (null != pvs) {
                    for (PropertyValue propertyValue : pvs.getPropertyValues()) {
                        beanDefinition.getPropertyValues().addPropertyValue(propert
yValue);
                    }
                }
            }
        }
    }

    // ...
}
```

- AbstractAutowireCapableBeanFactory#createBean 方法中有这一条新增加的方法调用，就是在设置 Bean 属性之前，允许 BeanPostProcessor 修改属性值 的操作 applyBeanPostProcessorsBeforeApplyingPropertyValues
- 那么这个 applyBeanPostProcessorsBeforeApplyingPropertyValues 方法中，首先就是获取已经注入的 BeanPostProcessor 集合并从中筛选出继承接口 InstantiationAwareBeanPostProcessor 的实现类。

- 最后就是调用相应的 postProcessPropertyValues 方法以及循环设置属性值信息，`beanDefinition.getPropertyValues().addPropertyValue(propertyValue);`

五、测试

1. 事先准备

配置 Dao

```
@Component
public class UserDao {

    private static Map<String, String> hashMap = new HashMap<>();

    static {
        hashMap.put("10001", "小傅哥，北京，亦庄");
        hashMap.put("10002", "八杯水，上海，尖沙咀");
        hashMap.put("10003", "阿毛，香港，铜锣湾");
    }

    public String queryUserName(String uId) {
        return hashMap.get(uId);
    }

}
```

- 给类配置上一个自动扫描注册 Bean 对象的注解 `@Component`，接下来会把这个类注入到 UserService 中。

注解注入到 UserService

```
@Component("userService")
public class UserService implements IUserService {

    @Value("${token}")
    private String token;

    @Autowired
    private UserDao userDao;

    public String queryUserInfo() {
```

```
        try {
            Thread.sleep(new Random(1).nextInt(100));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return userDao.queryUserName("10001") + ", " + token;
    }

    // ...
}
```

- 这里包括了两种类型的注入，一个是占位符注入属性信息 `@Value("${token}")`，另外一个是注入对象信息 `@Autowired`

2. 属性配置文件

token.properties

```
token=RejDlI78hu223Opo983Ds
```

spring.xml

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

    <bean class="cn.bugstack.springframework.beans.factory.PropertyPlaceholderConfi
gurer">
        <property name="location" value="classpath:token.properties"/>
    </bean>

    <context:component-scan base-package="cn.bugstack.springframework.test.bean"/>

</beans>
```

- 在 spring.xml 中配置了扫描属性信息和自动扫描包路径范围。

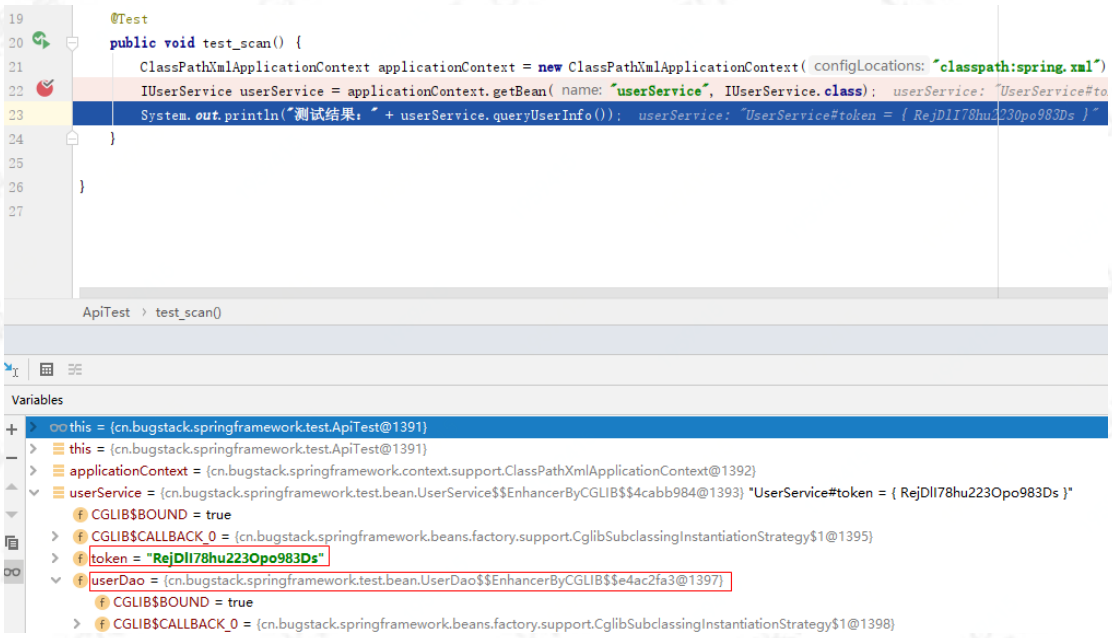
3. 单元测试

```
@Test
public void test_scan() {
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplication
Context("classpath:spring.xml");
    IUserService userService = applicationContext.getBean("userService", IUserServi
ce.class);
    System.out.println("测试结果：" + userService.queryUserInfo());
}
```

- 单元测试时候就可以完整的测试一个类注入到 Spring 容器，同时这个属性信息也可以被自动扫描填充上。

测试结果

```
测试结果：小傅哥，北京，亦庄，RejDlI78hu223Opo983Ds

Process finished with exit code 0
```



- 从测试结果可以看到现在我们的使用方式已经通过了，有自动扫描类，有注解注入属性。这与使用 Spring 框架越来越像了。

六、总结

- 从整个注解信息扫描注入的实现内容来看，我们一直是围绕着在 Bean 的生命周期中进行处理，就像 BeanPostProcessor 用于修改新实例化 Bean 对象的扩展点，提供的接口方法可以用于处理 Bean 对象实例化前后进行处理操作。而有时候需要做一些差异化的控制，所以还需要继承 BeanPostProcessor 接口，定义新的接口 InstantiationAwareBeanPostProcessor 这样就可以区分出不同扩展点的操作了。
- 像是接口用 instanceof 判断，注解用 Field.getAnnotation(Value.class); 获取，都是相当于在类上做的一些标识性信息，便于可以用一些方法找到这些功能点，以便进行处理。所以在我们日常开发设计的组件中，也可以运用上这些特点。
- 当你思考把你的实现融入到一个已经细分好的 Bean 生命周期中，你会发现它的设计是如此的好，可以让你在任何初始化的时间点上，任何面上，都能做你需要的扩展或者改变，这也是我们做程序设计时追求的灵活性。

第 16 章：给代理对象设置属性注入

一、功能强化

怎么了，运行的好好的放在别人电脑上就出错？

是不是有时候你觉得提交的代码，功能完善、逻辑正确、格式漂亮，但不管是小哥哥还是小姐姐，只要测试人员一上手，就会发现 **这有 Bug、那有 Bug、你回去改改别耽误我时间！** 这是为什么呢？

因为测试人员的输入的数据可不是你已经跑了几十遍能通过运行的简单数据，他们的数据更偏向于用户真实使用时候的输入效果。就像我们在使用 Spring 的时候，谁规定用户一定会使用普通的类对象呢，只要是 Java 的 JDK 中能提供的骚操作就都有可能在 Spring 框架下使用，比如：MyBatis 用了代理类、RPC 链接了注册中心、分库分表切换了数据源，那这些就都需要 Spring 来支持。而如果你在开发的过程中没有考虑到这些，可能也就忽略了此类功能的实现，**这好了**，测试那上手肯定就出 Bug 了！

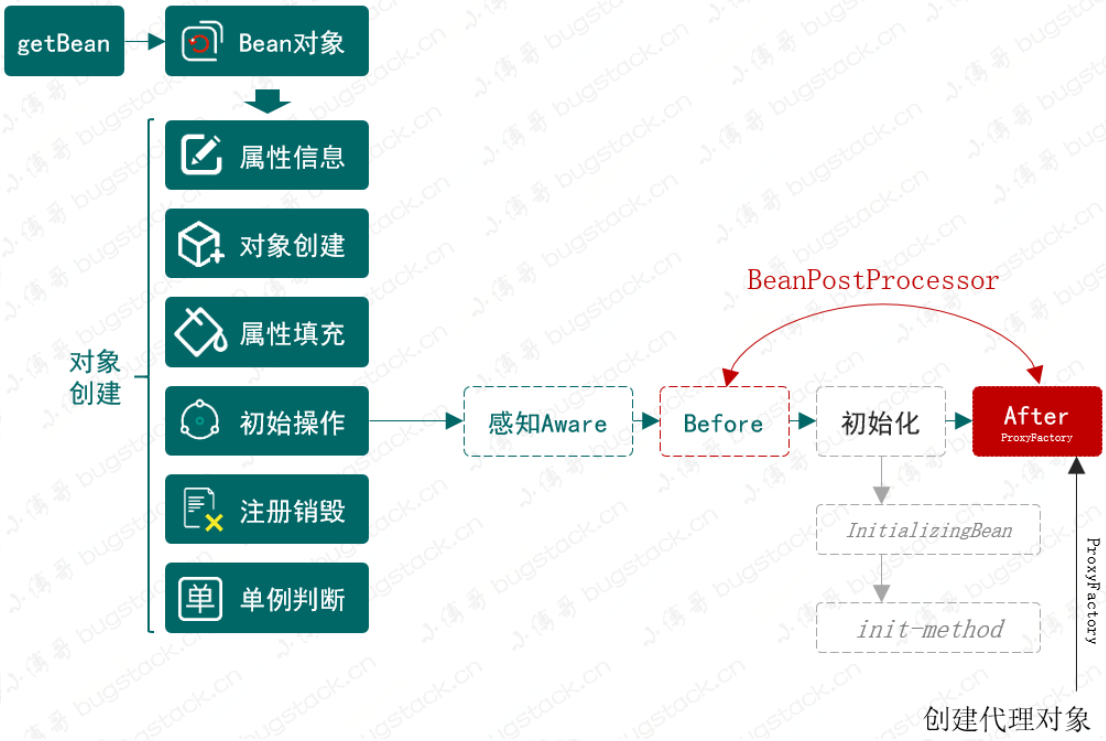
二、目标

其实本章节要解决的问题就是关于如何给代理对象中的属性填充相应的值，因为在之前把 AOP 动态代理，融入到 Bean 的生命周期时，创建代理对象是在整个创建 Bean 对象之前，也就是说这个代理对象的创建并不是在 Bean 生命周期中。
所以本章节中我们要把代理对象的创建融入到 Bean 的生命周期中，也就是需要把创建代理对象的逻辑迁移到 Bean 对象执行初始化方法之后，在执行代理对象的创建。

三、方案

按照创建代理对象的操作 `DefaultAdvisorAutoProxyCreator` 实现的 `InstantiationAwareBeanPostProcessor` 接口，那么原本在 Before 中的操作，

则需要放到 After 中处理。整体设计如下：



- 在创建 Bean 对象 `createBean` 的生命周期中，有一个阶段是在 Bean 对象属性填充完成以后，执行 Bean 的初始化方法和 BeanPostProcessor 的前置和后置处理，例如：感知 Aware 对象、处理 init-method 方法等。那么在这个阶段的 `BeanPostProcessor After` 就可以用于创建代理对象操作。
- 在 DefaultAdvisorAutoProxyCreator 用于创建代理对象的操作中，需要把创建操作从 postProcessBeforeInstantiation 方法中迁移到 postProcessAfterInitialization，这样才能满足 Bean 属性填充后的创建操作。

四、实现

1. 工程结构

```
small-spring-step-15
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── aop
    │           │   ├── aspectj
    │           │   │   └── AspectJExpressionPointcut.java
    │           │   │   └── AspectJExpressionPointcutAdvisor.java
    │           │   ├── framework
```

```
    │           │   │   ├── adapter
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── autoproxy
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── AopProxy.java
    │           │   │   ├── Cglib2AopProxy.java
    │           │   │   ├── JdkDynamicAopProxy.java
    │           │   │   ├── ProxyFactory.java
    │           │   │   └── ReflectiveMethodInvocation.java
    │           │   ├── AdvisedSupport.java
    │           │   ├── Advisor.java
    │           │   ├── BeforeAdvice.java
    │           │   ├── ClassFilter.java
    │           │   ├── MethodBeforeAdvice.java
    │           │   ├── MethodMatcher.java
    │           │   ├── Pointcut.java
    │           │   ├── PointcutAdvisor.java
    │           │   └── TargetSource.java
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── annotation
    │           │   │   │   ├── Autowired.java
    │           │   │   │   ├── AutowiredAnnotationBeanPostProcessor.java
    │           │   │   │   ├── Qualifier.java
    │           │   │   │   └── Value.java
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   │   ├── BeanPostProcessor.java
    │           │   │   │   ├── BeanReference.java
    │           │   │   │   ├── ConfigurableBeanFactory.java
    │           │   │   │   ├── InstantiationAwareBeanPostProcessor.java
    │           │   │   │   └── SingletonBeanRegistry.java
    │           │   │   ├── support
    │           │   │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   │   ├── AbstractBeanFactory.java
    │           │   │   │   ├── BeanDefinitionReader.java
    │           │   │   │   ├── BeanDefinitionRegistry.java
    │           │   │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   │   ├── DefaultListableBeanFactory.java
    │           │   │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   │   ├── DisposableBeanAdapter.java
```

```
    │           │   │   ├── FactoryBeanRegistrySupport.java
    │           │   │   ├── InstantiationStrategy.java
    │           │   │   └── SimpleInstantiationStrategy.java
    │           │   ├── support
    │           │   │   └── XmlBeanDefinitionReader.java
    │           │   ├── Aware.java
    │           │   ├── BeanClassLoaderAware.java
    │           │   ├── BeanFactory.java
    │           │   ├── BeanFactoryAware.java
    │           │   ├── BeanNameAware.java
    │           │   ├── ConfigurableListableBeanFactory.java
    │           │   ├── DisposableBean.java
    │           │   ├── FactoryBean.java
    │           │   ├── HierarchicalBeanFactory.java
    │           │   ├── InitializingBean.java
    │           │   ├── ListableBeanFactory.java
    │           │   └── PropertyPlaceholderConfigurer.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
    │           │   └── PropertyValues.java
    │           ├── context
    │           │   ├── annotation
    │           │   │   ├── ClassPathBeanDefinitionScanner.java
    │           │   │   ├── ClassPathScanningCandidateComponentProvider.java
    │           │   │   └── Scope.java
    │           │   ├── event
    │           │   │   ├── AbstractApplicationEventMulticaster.java
    │           │   │   ├── ApplicationContextEvent.java
    │           │   │   ├── ApplicationEventMulticaster.java
    │           │   │   ├── ContextClosedEvent.java
    │           │   │   ├── ContextRefreshedEvent.java
    │           │   │   └── SimpleApplicationEventMulticaster.java
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   ├── ApplicationContextAwareProcessor.java
    │           │   │   └── ClassPathXmlApplicationContext.java
    │           │   ├── ApplicationContext.java
    │           │   ├── ApplicationContextAware.java
    │           │   ├── ApplicationEvent.java
    │           │   ├── ApplicationEventPublisher.java
    │           │   ├── ApplicationListener.java
    │           │   └── ConfigurableApplicationContext.java
```

```
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           ├── stereotype
    │           │   └── Component.java
    │           └── utils
    │               ├── ClassUtils.java
    │               └── StringValueResolver.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── IUserService.java
                │   └── UserService.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

在 Bean 的生命周期中创建代理对象的类关系，如图 16-2

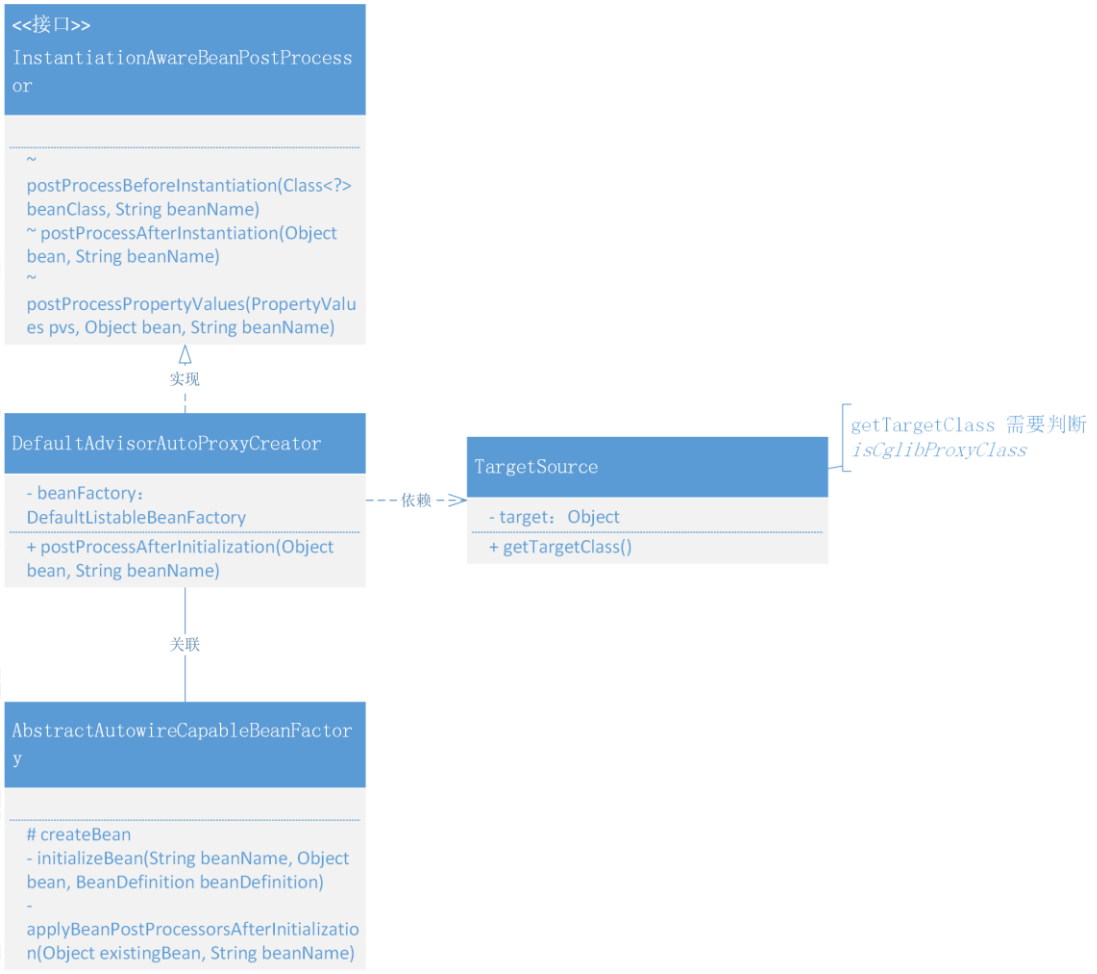


图 16-2

- 虽然本章节要完成的是关于代理对象中属性的填充问题，但实际解决的思路是处理在 Bean 的生命周期中合适的位置（初始化 `initializeBean`）中处理代理类的创建。
- 所以以上的改动并不会涉及太多内容，主要包括：DefaultAdvisorAutoProxyCreator 类创建代理对象的操作放置在 postProcessAfterInitialization 方法中以及对应在 AbstractAutowireCapableBeanFactory 完成初始化方法的调用操作。
- 另外还有一点要注意，就是目前我们在 Spring 框架中，AbstractAutowireCapableBeanFactory 类里使用的是 CglibSubclassingInstantiationStrategy 创建对象，所以有需要判断对象获取接口的方法中，也都需要判断是否为 CGlib 创建，否则是不能正确获取到接口的。如：

```
ClassUtils.isCglibProxyClass(clazz) ? clazz.getSuperclass() : clazz;
```

2. 判断 CGlib 对象

cn.bugstack.springframework.aop.TargetSource

```
public class TargetSource {

    private final Object target;

    /**
     * Return the type of targets returned by this {@link TargetSource}.
     * <p>Can return <code>null</code>, although certain usages of a
     * <code>TargetSource</code> might just work with a predetermined
     * target class.
     *
     * @return the type of targets returned by this {@link TargetSource}
     */
    public Class<?>[] getTargetClass() {
        Class<?> clazz = this.target.getClass();
        clazz = ClassUtils.isCglibProxyClass(clazz) ? clazz.getSuperclass() : clazz
;
        return clazz.getInterfaces();
    }

}
```

- 在 TargetSource#getTargetClass 是用于获取 target 对象的接口信息的，那么这个 target 可能是 JDK 代理 创建也可能是 CGlib 创建，为了保证都能正确的获取到结果，这里需要增加判读 `ClassUtils.isCglibProxyClass(clazz)`

3. 迁移创建 AOP 代理方法

cn.bugstack.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator

```
public class DefaultAdvisorAutoProxyCreator implements InstantiationAwareBeanPostPr
ocessor, BeanFactoryAware {

    private DefaultListableBeanFactory beanFactory;

    @Override
    public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanNam
```

```
e) throws BeansException {
        return null;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) thro
ws BeansException {

        if (isInfrastructureClass(bean.getClass())) return bean;

        Collection<AspectJExpressionPointcutAdvisor> advisors = beanFactory.getBean
sOfType(AspectJExpressionPointcutAdvisor.class).values();

        for (AspectJExpressionPointcutAdvisor advisor : advisors) {
            ClassFilter classFilter = advisor.getPointcut().getClassFilter();
            // 过滤匹配类
            if (!classFilter.matches(bean.getClass())) continue;

            AdvisedSupport advisedSupport = new AdvisedSupport();

            TargetSource targetSource = new TargetSource(bean);
            advisedSupport.setTargetSource(targetSource);
            advisedSupport.setMethodInterceptor((MethodInterceptor) advisor.getAdvi
ce());
            advisedSupport.setMethodMatcher(advisor.getPointcut().getMethodMatcher(
));
            advisedSupport.setProxyTargetClass(false);

            // 返回代理对象
            return new ProxyFactory(advisedSupport).getProxy();
        }

        return bean;
    }

}
```

- 关于 DefaultAdvisorAutoProxyCreator 类的操作主要就是把创建 AOP 代理的操作从 postProcessBeforeInstantiation 移动到 postProcessAfterInitialization 中去。
- 通过设置一些 AOP 的必备参数后，返回代理对象 `new ProxyFactory(advisedSupport).getProxy()` 这个代理对象中就包括间接调用了 TargetSource 中对 getTargetClass() 的获取。

4. 在 Bean 的生命周期中初始化执行

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactory implements AutowireCapableBeanFactory {

    private InstantiationStrategy instantiationStrategy = new CglibSubclassingInstantiationStrategy();

    @Override
    protected Object createBean(String beanName, BeanDefinition beanDefinition, Object[] args) throws BeansException {
        Object bean = null;
        try {
            // ...

            // 执行 Bean 的初始化方法和 BeanPostProcessor 的前置和后置处理方法
            bean = initializeBean(beanName, bean, beanDefinition);
        } catch (Exception e) {
            throw new BeansException("Instantiation of bean failed", e);
        }
        // ...
        return bean;
    }

    private Object initializeBean(String beanName, Object bean, BeanDefinition beanDefinition) {

        // ...

        wrappedBean = applyBeanPostProcessorsAfterInitialization(bean, beanName);
        return wrappedBean;
    }

    @Override
    public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName) throws BeansException {
        Object result = existingBean;
        for (BeanPostProcessor processor : getBeanPostProcessors()) {
            Object current = processor.postProcessAfterInitialization(result, beanName);
            if (null == current) return result;
```

```
            result = current;
        }
        return result;
    }

}
```

- 在 AbstractAutowireCapableBeanFactory#createBean 方法中，其实关注点就在于 initializeBean -> applyBeanPostProcessorsAfterInitialization 这一块逻辑的调用，最终完成 AOP 代理对象的创建操作。

五、测试

1. 事先准备

UserService 添加属性字段

```
public class UserService implements IUserService {

    private String token;

    public String queryUserInfo() {
        try {
            Thread.sleep(new Random(1).nextInt(100));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "小傅哥，100001，深圳，" + token;
    }

}
```

- token 是在 UserService 中新增的属性信息，用于测试代理对象的属性填充操作。

2. 属性配置文件

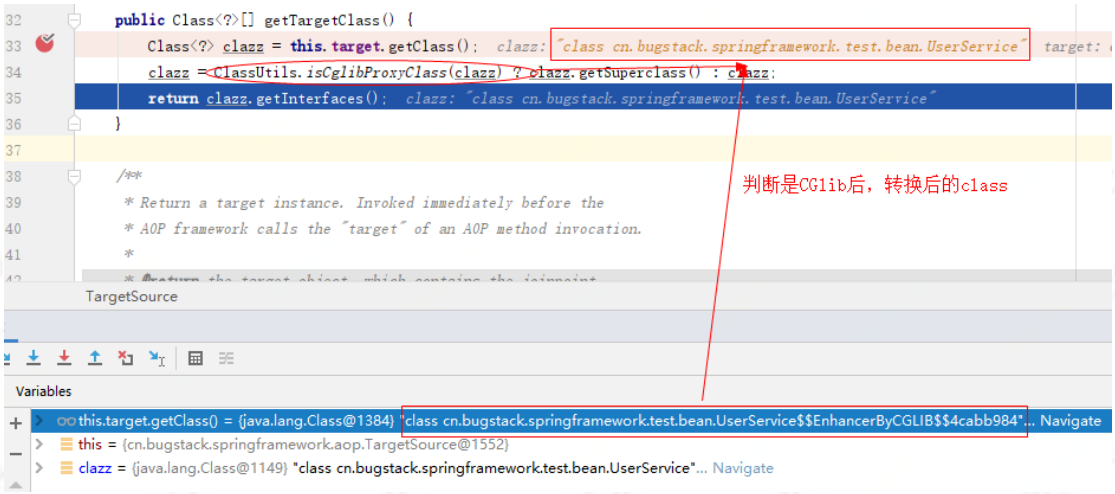
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd">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

```
    <bean id="userService" class="cn.bugstack.springframework.test.bean.UserService
">
        <property name="token" value="RejDlI78hu223Opo983Ds"/>
    </bean>

    <bean class="cn.bugstack.springframework.aop.framework.autoproxy.DefaultAdvisor
AutoProxyCreator"/>

    <bean id="beforeAdvice" class="cn.bugstack.springframework.test.bean.UserServic
eBeforeAdvice"/>

    <bean id="methodInterceptor" class="cn.bugstack.springframework.aop.framework.a
dapter.MethodBeforeAdviceInterceptor">
        <property name="advice" ref="beforeAdvice"/>
    </bean>

    <bean id="pointcutAdvisor" class="cn.bugstack.springframework.aop.aspectj.Aspec
tJExpressionPointcutAdvisor">
        <property name="expression" value="execution(* cn.bugstack.springframework.
test.bean.IUserService.*(..))"/>
        <property name="advice" ref="methodInterceptor"/>
    </bean>

</beans>
```

- 与我们对 AOP 的测试来说，唯一新增加的就是 property 的配置：`<property name="token" value="RejDlI78hu223Opo983Ds"/>`

3. 单元测试

```
@Test
public void test_autoProxy() {
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplication
Context("classpath:spring.xml");
    IUserService userService = applicationContext.getBean("userService", IUserServi
ce.class);
    System.out.println("测试结果：" + userService.queryUserInfo());
}
```



测试结果

拦截方法：queryUserInfo

测试结果：小傅哥，100001，深圳，RejDlI78hu223Opo983Ds

```
Process finished with exit code 0
```

- 从测试结果可以看到，通过对 Bean 生命周期的调整，在创建 AOP 代理对象就可以把代理对象的属性信息填充进去了。
- 另外这里还有一块是关于在 TargetSource#getTargetClass 中关于是否为 CGlib 的方法判断，只有这样操作才可以获取到争取的类信息。

六、总结

- 本章节的核心知识内容主要是完善了 Bean 的生命周期，在创建类的操作中完成代理对象的创建，通过这样的方式就可以让代理对象中的属性也可以随着创建过程被填充进去。
- 除了核心功能的实现外也要关注到对象的初始化操作是 CglibSubclassingInstantiationStrategy、SimpleInstantiationStrategy，这两种方式中的 CGlib 创建对象，会影响到很多地方用于接口获取的操作，因为 CGlib 创建对象走的是 ASM 字节码生成的操作，所以和普通的 JDK 代理生成对象是不一样，需要注意。
- 程序的 Bug 往往是对需求的使用场景理解不足，功能的完善是对一个细化场景的程序精雕，开发程序的过程远远不只是写代码那么回事，更重要的是思考这是什么场景、遇到了哪些问题、要怎么解决、可以学到什么中不断的锤炼自己的程序逻辑。

高级篇： Design

第 17 章：三级缓存处理循环依赖

一、不止能用

嘎哈呀，又不是不能用！

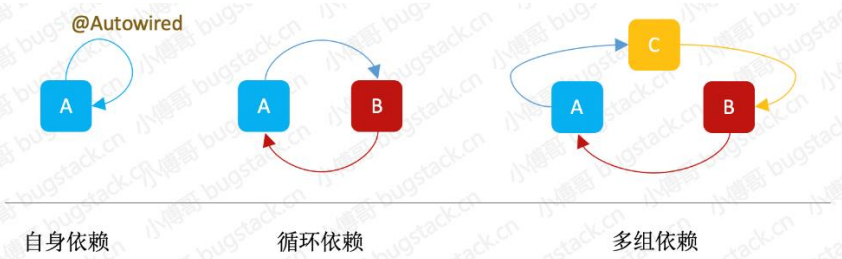
我经常说业务逻辑的代码实现，就像擦屁屁的纸，80%的面积都是保护手的。而那 20%的核心流程也就仅仅是你说的能用就行，反正每次都洗手呗。

其实想把程序从**能用**实现到**好用**并不容易，这包括你对业务的理解、你对架构的把控、你对细节的实现等等，也包括你是否能做一些列的抽象实现，不至于整个程序随着开发的越多就变的越臃肿不堪。

那么对于编程上的写好程序的理解，我通常喜欢用生活中实际的例子来表达，因为有不少前辈的研发大牛都说：“你要面对对象编程”。所以嘞，我可能会用生活中的超市、展台、货架、官渡等来对我的程序开发中的类或者领域服务进行命名和实现，这样抽象化出来的代码逻辑更具有扩展性，也能让新接手的人快速理解并且不至于慌乱的开发。

二、目标

按照目前我们实现的 Spring 框架，是可以满足一个基本需求的，但如果你配置了 A、B 两个 Bean 对象互相依赖，那么立马会抛出 java.lang.StackOverflowError，为什么呢？因为 A 创建时需要依赖 B 创建，而 B 的创建又依赖于 A 创建，就这样死循环了。
而这个循环依赖基本也可以说是 Spring 中非常经典的实现了，所要解决的场景主要有以下三种情况：



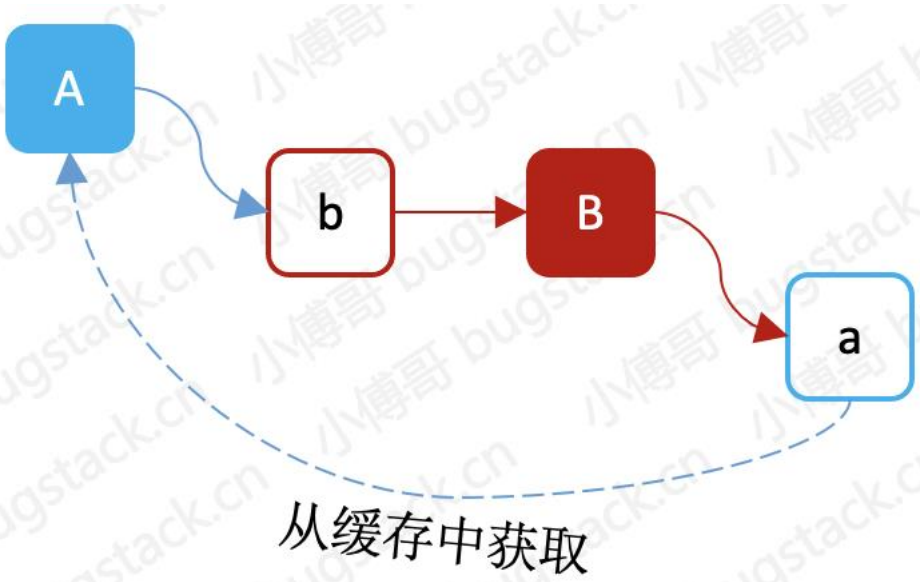
- 循环依赖主要分为这三种，自身依赖于自身、互相循环依赖、多组循环依赖。
- 但无论循环依赖的数量有多少，循环依赖的本质是一样的。就是你的完整创建依赖于我，而我的完整创建也依赖于你，但我们互相没法解耦，最终导致依赖创建失败。
- 所以需要 Spring 提供了除了构造函数注入和原型注入外的，setter 循环依赖注入解决方案。

三、设计

按照 Spring 框架的设计，用于解决循环依赖需要用到三个缓存，这三个缓存分别存放了成品对象、半成品对象(未填充属性值)、代理对象，分阶段存放对象内容，来解决循环依赖问题。

那么，这里我们需要知道一个核心的原理，就是用于解决循环依赖就必须是三级缓存呢，二级行吗？一级可以不？其实都能解决，只不过 Spring 框架的实现要保证几个事情，如只有一级缓存处理流程没法拆分，复杂度也会增加，同时半成品对象可能会有空指针异常。而将半成品与成品对象分开，处理起来也更加优雅、简单、易扩展。另外 Spring 的两大特性中不仅有 IOC 还有 AOP，也就是基于字节码增强后的方法，该存放到哪，而三级缓存最主要，要解决的循环依赖就是对 AOP 的处理，但如果把 AOP 代理对象的创建提前，那么二级缓存也一样可以解决。但是，这就违背了 Spring 创建对象的原则，Spring 更喜欢把所有的普通 Bean 都初始化完成，在处理代理对象的初始化。

不过，没关系我们可以先尝试仅适用一级缓存来解决循环依赖，通过这样的方式从中学习到处理循环依赖的最核心原来，也就是那 20%的地方。



- 如果仅以一级缓存解决循环依赖，那么在实现上可以通过在 A 对象 newInstance 创建且未填充属性后，直接放入缓存中。
- 在 A 对象的属性填充 B 对象时，如果缓存中不能获取到 B 对象，则开始创建 B 对象，同样创建完成后，把 B 对象填充到缓存中去。
- 接下来就开始对 B 对象的属性进行填充，恰好这会可以从缓存中拿到半成品的 A 对象，那么这个时候 B 对象的属性就填充完了。
- 最后返回来继续完成 A 对象的属性填充，把实例化后并填充了属性的 B 对象赋值给 A 对象的 b 属性，这样就完成了一个循环依赖操作。

代码实现

```
private final static Map<String, Object> singletonObjects = new ConcurrentHashMap<>
(256);

private static <T> T getBean(Class<T> beanClass) throws Exception {
    String beanName = beanClass.getSimpleName().toLowerCase();
    if (singletonObjects.containsKey(beanName)) {
        return (T) singletonObjects.get(beanName);
    }
    // 实例化对象入缓存
    Object obj = beanClass.newInstance();
    singletonObjects.put(beanName, obj);
    // 属性填充补全对象
```

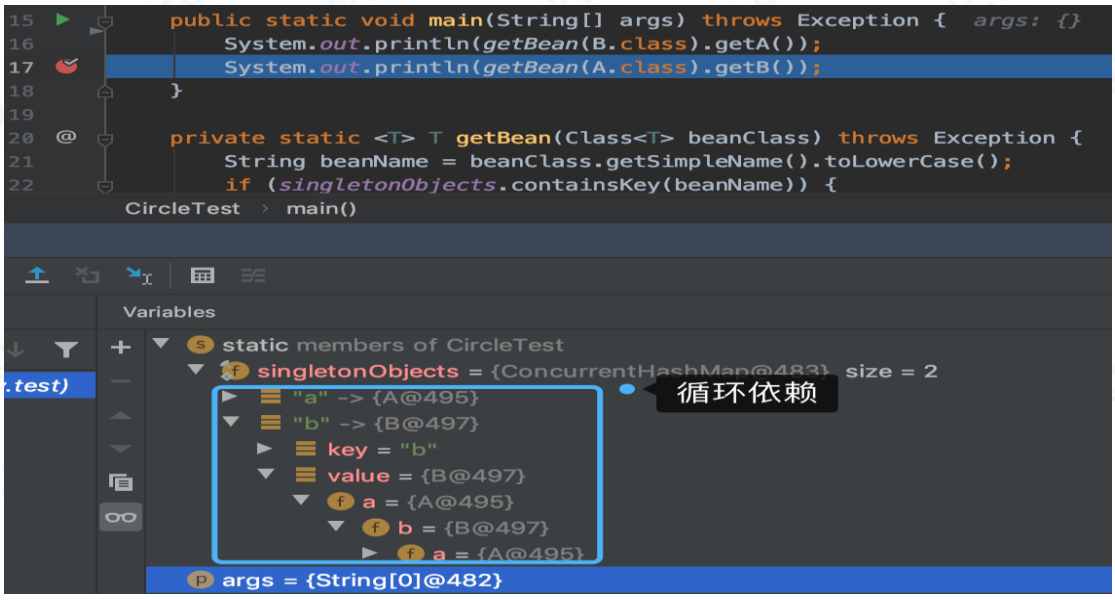
```
    Field[] fields = obj.getClass().getDeclaredFields();
    for (Field field : fields) {
        field.setAccessible(true);
        Class<?> fieldClass = field.getType();
        String fieldBeanName = fieldClass.getSimpleName().toLowerCase();
        field.set(obj, singletonObjects.containsKey(fieldBeanName) ? singletonObjec
ts.get(fieldBeanName) : getBean(fieldClass));
        field.setAccessible(false);
    }
    return (T) obj;
}
```

- 使用一级缓存存放对象的方式，就是这样简单的实现过程，只要是创建完对象，立马塞到缓存里去。这样就可以在其他对象创建时候获取到属性需要填充的对象了。

测试结果

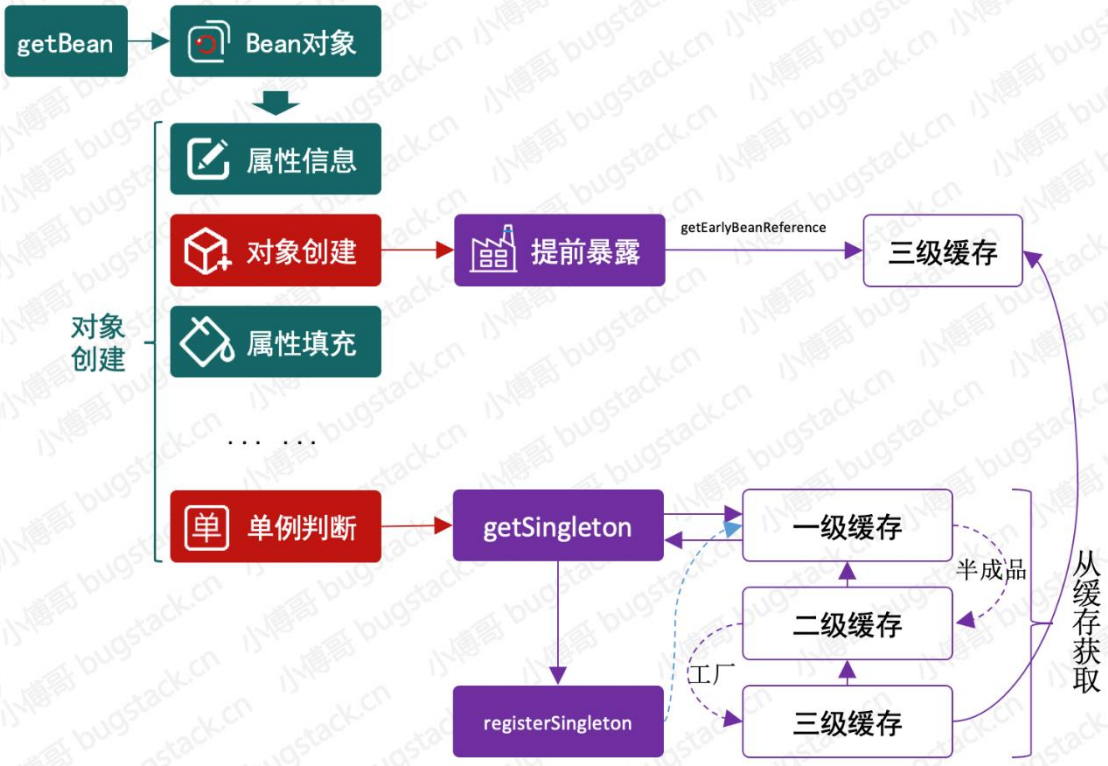
```
public static void main(String[] args) throws Exception {
    System.out.println(getBean(B.class).getA());
    System.out.println(getBean(A.class).getB());
}
```

```
cn.bugstack.springframework.test.A@49476842
cn.bugstack.springframework.test.B@78308db1

Process finished with exit code 0
```



- 从测试效果和截图依赖过程中可以看到，一级缓存也可以解决简单场景的循环依赖问题。
- 其实 `getBean`，是整个解决循环依赖的核心内容，A 创建后填充属性时依赖 B，那么就去创建 B，在创建 B 开始填充时发现依赖于 A，但此时 A 这个半成品对象已经存放在缓存到 `singletonObjects` 中了，所以 B 可以正常创建，在通过递归把 A 也创建完整了。

有了以上这部分关于循环依赖的处理内容，在理解循环依赖就没那么复杂了。接下来我们带着这个感觉去思考如果有对象不只是简单的对象，还有代理对象，还有 AOP 应用，要怎么处理这样的依赖问题。整体设计结构如下图：



- 关于循环依赖在我们目前的 Spring 框架中扩展起来也并不会太复杂，主要就是对于创建对象的提前暴露，如果是工厂对象则会使用 getEarlyBeanReference 逻辑提前将工厂对象存放到三级缓存中。等到后续获取对象的时候实际拿到的是工厂对象中 getObject，这个才是最终的实际对象。
- 在创建对象的 `AbstractAutowireCapableBeanFactory#doCreateBean` 方法中，提前暴露对象以后，就可以通过接下来的流程，getSingleton 从三个缓存中以此寻找对象，一级、二级如果有则直接取走，如果对象是三级缓存中则会从三级缓存中获取后并删掉工厂对象，把实际对象放到二级缓存中。

- 最后是关于单例的对象的注册操作，这个注册操作就是把真实的实际对象放到一级缓存中，因为此时它已经是一个成品对象了。

四、实现

1. 工程结构

```
small-spring-step-16
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── aop
    │           │   ├── aspectj
    │           │   │   └── AspectJExpressionPointcut.java
    │           │   │   └── AspectJExpressionPointcutAdvisor.java
    │           │   ├── framework
    │           │   │   ├── adapter
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── autoproxy
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── AopProxy.java
    │           │   │   ├── Cglib2AopProxy.java
    │           │   │   ├── JdkDynamicAopProxy.java
    │           │   │   ├── ProxyFactory.java
    │           │   │   └── ReflectiveMethodInvocation.java
    │           │   ├── AdvisedSupport.java
    │           │   ├── Advisor.java
    │           │   ├── BeforeAdvice.java
    │           │   ├── ClassFilter.java
    │           │   ├── MethodBeforeAdvice.java
    │           │   ├── MethodMatcher.java
    │           │   ├── Pointcut.java
    │           │   ├── PointcutAdvisor.java
    │           │   └── TargetSource.java
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── annotation
    │           │   │   │   ├── Autowired.java
    │           │   │   │   ├── AutowiredAnnotationBeanPostProcessor.java
    │           │   │   │   ├── Qualifier.java
    │           │   │   │   └── Value.java
```

```
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   │   ├── BeanPostProcessor.java
    │           │   │   │   ├── BeanReference.java
    │           │   │   │   ├── ConfigurableBeanFactory.java
    │           │   │   │   ├── InstantiationAwareBeanPostProcessor.java
    │           │   │   │   └── SingletonBeanRegistry.java
    │           │   │   ├── support
    │           │   │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   │   ├── AbstractBeanFactory.java
    │           │   │   │   ├── BeanDefinitionReader.java
    │           │   │   │   ├── BeanDefinitionRegistry.java
    │           │   │   │   ├── CglibSubclassingInstantiationStrategy.java
    │           │   │   │   ├── DefaultListableBeanFactory.java
    │           │   │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   │   ├── DisposableBeanAdapter.java
    │           │   │   │   ├── FactoryBeanRegistrySupport.java
    │           │   │   │   ├── InstantiationStrategy.java
    │           │   │   │   └── SimpleInstantiationStrategy.java
    │           │   │   ├── support
    │           │   │   │   └── XmlBeanDefinitionReader.java
    │           │   │   ├── Aware.java
    │           │   │   ├── BeanClassLoaderAware.java
    │           │   │   ├── BeanFactory.java
    │           │   │   ├── BeanFactoryAware.java
    │           │   │   ├── BeanNameAware.java
    │           │   │   ├── ConfigurableListableBeanFactory.java
    │           │   │   ├── DisposableBean.java
    │           │   │   ├── FactoryBean.java
    │           │   │   ├── HierarchicalBeanFactory.java
    │           │   │   ├── InitializingBean.java
    │           │   │   ├── ListableBeanFactory.java
    │           │   │   ├── ObjectFactory.java
    │           │   │   └── PropertyPlaceholderConfigurer.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
    │           │   └── PropertyValues.java
    │           ├── context
    │           │   ├── annotation
    │           │   │   ├── ClassPathBeanDefinitionScanner.java
    │           │   │   ├── ClassPathScanningCandidateComponentProvider.java
```

```
    │           │   │   └── Scope.java
    │           │   ├── event
    │           │   │   ├── AbstractApplicationEventMulticaster.java
    │           │   │   ├── ApplicationContextEvent.java
    │           │   │   ├── ApplicationEventMulticaster.java
    │           │   │   ├── ContextClosedEvent.java
    │           │   │   ├── ContextRefreshedEvent.java
    │           │   │   └── SimpleApplicationEventMulticaster.java
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   ├── ApplicationContextAwareProcessor.java
    │           │   │   └── ClassPathXmlApplicationContext.java
    │           │   ├── ApplicationContext.java
    │           │   ├── ApplicationContextAware.java
    │           │   ├── ApplicationEvent.java
    │           │   ├── ApplicationEventPublisher.java
    │           │   ├── ApplicationListener.java
    │           │   └── ConfigurableApplicationContext.java
    │           ├── core.io
    │           │   ├── ClassPathResource.java
    │           │   ├── DefaultResourceLoader.java
    │           │   ├── FileSystemResource.java
    │           │   ├── Resource.java
    │           │   ├── ResourceLoader.java
    │           │   └── UrlResource.java
    │           ├── stereotype
    │           │   └── Component.java
    │           └── utils
    │               ├── ClassUtils.java
    │               └── StringValueResolver.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   ├── Husband.java
                │   ├── HusbandMother.java
                │   ├── IMother.java
                │   ├── SpouseAdvice.java
                │   └── Wife.java
                ├── ApiTest.java
                └── CircleTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

处理循环依赖核心流程的类关系的操作过程包括：

- 循环依赖的核心功能实现主要包括 DefaultSingletonBeanRegistry 提供三级缓存：`singletonObjects`、`earlySingletonObjects`、`singletonFactories`，分别存放成品对象、半成品对象和工厂对象。同时包装三个缓存提供方法：getSingleton、registerSingleton、addSingletonFactory，这样使用方就可以分别在不同时间段存放和获取对应的对象了。
- 在 AbstractAutowireCapableBeanFactory 的 doCreateBean 方法中，提供了关于提前暴露对象的操作，`addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, beanDefinition, finalBean));`，以及后续获取对象和注册对象的操作，`exposedObject = getSingleton(beanName);`、`registerSingleton(beanName, exposedObject);`，经过这样的处理就可以完成对复杂场景循环依赖的操作。
- 另外在 DefaultAdvisorAutoProxyCreator 提供的切面服务中，也需要实现接口 InstantiationAwareBeanPostProcessor 新增的 getEarlyBeanReference 方法，便于把依赖的切面对象也能存放到三级缓存中，处理对应的循环依赖。

2. 设置三级缓存

cn.bugstack.springframework.beans.factory.support.DefaultSingletonBeanRegistry

```
public class DefaultSingletonBeanRegistry implements SingletonBeanRegistry {

    // 一级缓存，普通对象
    private Map<String, Object> singletonObjects = new ConcurrentHashMap<>();

    // 二级缓存，提前暴漏对象，没有完全实例化的对象
    protected final Map<String, Object> earlySingletonObjects = new HashMap<String,
 Object>();

    // 三级缓存，存放代理对象
    private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<St
ring, ObjectFactory<?>>();

    private final Map<String, DisposableBean> disposableBeans = new LinkedHashMap<>
();

    @Override
    public Object getSingleton(String beanName) {
        Object singletonObject = singletonObjects.get(beanName);
        if (null == singletonObject) {
```

```
            singletonObject = earlySingletonObjects.get(beanName);
            // 判断二级缓存中是否有对象，这个对象就是代理对象，因为只有代理对象才会放到三级缓存中
            if (null == singletonObject) {
                ObjectFactory<?> singletonFactory = singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    singletonObject = singletonFactory.getObject();
                    // 把三级缓存中的代理对象中的真实对象获取出来，放入二级缓存中
                    earlySingletonObjects.put(beanName, singletonObject);
                    singletonFactories.remove(beanName);
                }
            }
        }
        return singletonObject;
    }

    public void registerSingleton(String beanName, Object singletonObject) {
        singletonObjects.put(beanName, singletonObject);
        earlySingletonObjects.remove(beanName);
        singletonFactories.remove(beanName);
    }

    protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory){
        if (!this.singletonObjects.containsKey(beanName)) {
            this.singletonFactories.put(beanName, singletonFactory);
            this.earlySingletonObjects.remove(beanName);
        }
    }

    public void registerDisposableBean(String beanName, DisposableBean bean) {
        disposableBeans.put(beanName, bean);
    }

}
```

- 在用于提供单例对象注册的操作的 DefaultSingletonBeanRegistry 类中，共有三个缓存对象的属性；singletonObjects、earlySingletonObjects、singletonFactories，如它们的名字一样，用于存放不同类型的对象（单例对象、早期的半成品单例对象、单例工厂对象）。
- 紧接着在这三个缓存对象下提供了获取、添加和注册不同对象的方法，包括：getSingleton、registerSingleton、addSingletonFactory，其实后面这两个方法都比

较简单，主要是 getSingleton 的操作，它是在一层层处理不同时期的单例对象，直至拿到有效的对象。

3. 提前暴露对象

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
public abstract class AbstractAutowireCapableBeanFactory extends AbstractBeanFactor
y implements AutowireCapableBeanFactory {

    protected Object doCreateBean(String beanName, BeanDefinition beanDefinition, O
bject[] args) {
        Object bean = null;
        try {
            // 实例化 Bean
            bean = createBeanInstance(beanDefinition, beanName, args);

            // 处理循环依赖，将实例化后的Bean 对象提前放入缓存中暴露出来
            if (beanDefinition.isSingleton()) {
                Object finalBean = bean;
                addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName,
 beanDefinition, finalBean));
            }

            // 实例化后判断
            boolean continueWithPropertyPopulation = applyBeanPostProcessorsAfterIn
stantiation(beanName, bean);
            if (!continueWithPropertyPopulation) {
                return bean;
            }
            // 在设置 Bean 属性之前，允许 BeanPostProcessor 修改属性值
            applyBeanPostProcessorsBeforeApplyingPropertyValues(beanName, bean, bea
nDefinition);
            // 给 Bean 填充属性
            applyPropertyValues(beanName, bean, beanDefinition);
            // 执行 Bean 的初始化方法和 BeanPostProcessor 的前置和后置处理方法
            bean = initializeBean(beanName, bean, beanDefinition);
        } catch (Exception e) {
            throw new BeansException("Instantiation of bean failed", e);
        }

        // 注册实现了 DisposableBean 接口的 Bean 对象
```

```
        registerDisposableBeanIfNecessary(beanName, bean, beanDefinition);

        // 判断 SCOPE_SINGLETON、SCOPE_PROTOTYPE
        Object exposedObject = bean;
        if (beanDefinition.isSingleton()) {
            // 获取代理对象
            exposedObject = getSingleton(beanName);
            registerSingleton(beanName, exposedObject);
        }
        return exposedObject;

    }

    protected Object getEarlyBeanReference(String beanName, BeanDefinition beanDefi
nition, Object bean) {
        Object exposedObject = bean;
        for (BeanPostProcessor beanPostProcessor : getBeanPostProcessors()) {
            if (beanPostProcessor instanceof InstantiationAwareBeanPostProcessor) {
                exposedObject = ((InstantiationAwareBeanPostProcessor) beanPostProc
essor).getEarlyBeanReference(exposedObject, beanName);
                if (null == exposedObject) return exposedObject;
            }
        }

        return exposedObject;
    }

    // ...
}
```

- 在 AbstractAutowireCapableBeanFactory#doCreateBean 的方法中主要是扩展了对象的提前暴露 addSingletonFactory 了，和单例对象的获取 getSingleton 以及注册操作 registerSingleton。
- 这里提到一点 getEarlyBeanReference 就是定义在如 AOP 切面中这样的代理对象，可以参考源码中接口 InstantiationAwareBeanPostProcessor#getEarlyBeanReference 方法的实现。

五、测试

因为是要测试循环依赖，我们找一个比较贴近的场景来做测试，*我说过我是一个喜欢从生活中发现面向对象编程的人* 我们的案例场景人物包括：老公和媳妇

互相依赖、婆婆是一个模拟成代理妈妈职责、在加上一个切面来关心家庭生活

1. 事先准备

老公，类

```
public class Husband {

    private Wife wife;

    public String queryWife(){
        return "Husband.wife";
    }

}
```

媳妇，类

```
public class Wife {

    private Husband husband;
    private IMother mother; // 婆婆

    public String queryHusband() {
        return "Wife.husband、Mother.callMother：" + mother.callMother();
    }

}
```

婆婆，代理了媳妇原来妈妈的职责的类

```
public class HusbandMother implements FactoryBean<IMother> {

    @Override
    public IMother getObject() throws Exception {
        return (IMother) Proxy.newProxyInstance(Thread.currentThread().getContextCl
assLoader(), new Class[]{IMother.class}, (proxy, method, args) -> "婚后媳妇妈妈的职责
被婆婆代理了！" + method.getName());
    }

}
```

切面，类

```
public class SpouseAdvice implements MethodBeforeAdvice {

    @Override
    public void before(Method method, Object[] args, Object target) throws Throwabl
e {
        System.out.println("关怀小两口(切面)：" + method);
    }

}
```

2. 属性配置文件

spring.xml

```
<bean id="husband" class="cn.bugstack.springframework.test.bean.Husband">
    <property name="wife" ref="wife"/>
</bean>

<bean id="wife" class="cn.bugstack.springframework.test.bean.Wife">
    <property name="husband" ref="husband"/>
    <property name="mother" ref="husbandMother"/>
</bean>

<bean id="husbandMother" class="cn.bugstack.springframework.test.bean.HusbandMother
"/>

<!-- AOP 配置，验证三级缓存 -->
<bean class="cn.bugstack.springframework.aop.framework.autoproxy.DefaultAdvisorAuto
ProxyCreator"/>

<bean id="beforeAdvice" class="cn.bugstack.springframework.test.bean.SpouseAdvice"/
>

<bean id="methodInterceptor" class="cn.bugstack.springframework.aop.framework.adapt
er.MethodBeforeAdviceInterceptor">
    <property name="advice" ref="beforeAdvice"/>
</bean>

<bean id="pointcutAdvisor" class="cn.bugstack.springframework.aop.aspectj.AspectJEx
pressionPointcutAdvisor">
    <property name="expression" value="execution(* cn.bugstack.springframework.test
.bean.Wife.*(..))"/>
```

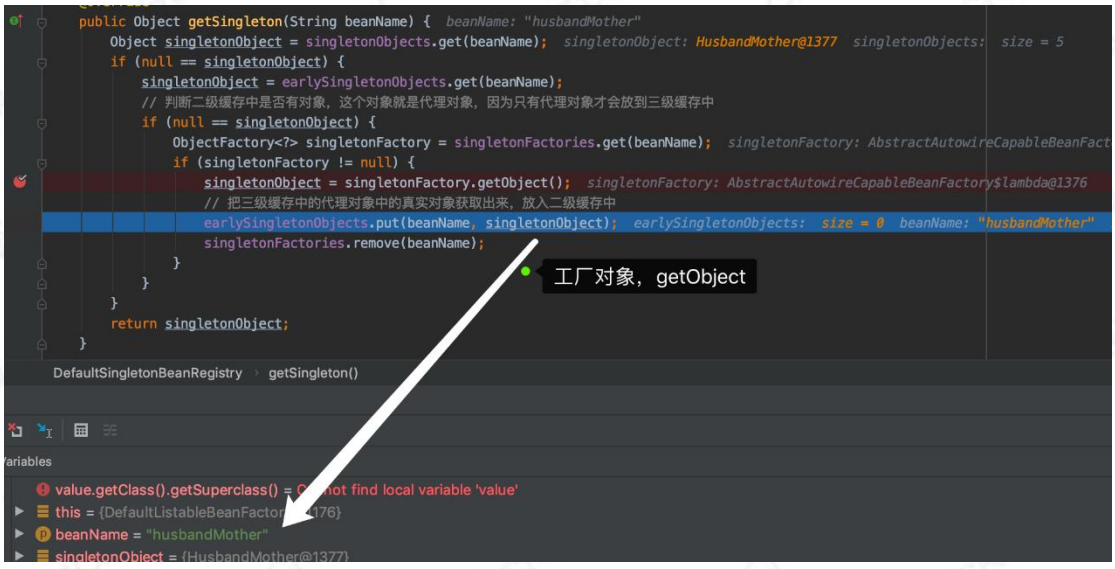
```
    <property name="advice" ref="methodInterceptor"/>
</bean>
```

- 这个里的配置就很简单了，配置 husband 依赖 wife，配置 wife 依赖 husband 和 mother，最后是关于 AOP 切面的依赖使用。

3. 单元测试

```
@Test
public void test_circular() {
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplication
Context("classpath:spring.xml");
    Husband husband = applicationContext.getBean("husband", Husband.class);
    Wife wife = applicationContext.getBean("wife", Wife.class);
    System.out.println("老公的媳妇：" + husband.queryWife());
    System.out.println("媳妇的老公：" + wife.queryHusband());
}
```

测试结果



老公的媳妇：Husband.wife
关怀小两口(切面)：
public java.lang.String cn.bugstack.springframework.test.bean.Wife.queryHusband()
媳妇的老公：Wife.husband、Mother.callMother：婚后媳妇妈妈的职责被婆婆代理了！callMother

```
Process finished with exit code 0
```

- 从测试结果可以看到，无论是简单对象依赖 *老公依赖媳妇、媳妇依赖老公*，还是代理工程对象或者 AOP 切面对象都可以在三级缓存下解决循环依赖的问题了。
- 此外从运行截图 `DefaultSingletonBeanRegistry#getSingleton` 中也可以看到凡事需要三级缓存存放工厂对象的类，都会使用到 getObject 获取真实对象，并随后存入半成品对象 earlySingletonObjects 中以及移除工厂对象。

六、总结

- Spring 中所有的功能都是以解决 Java 编程中的特性而存在的，就像我们本章节处理的循环依赖，如果没有 Spring 框架的情况下，可能我们也会尽可能避免写出循环依赖的操作，因为在没有经过加工处理后，这样的依赖关系肯定会报错的。*那么这也就是程序从能用到好用的升级*
- 在解决循环依赖的核心流程中，主要是提前暴露对象的设计，以及建立三级缓存的数据结构来存放不同时期的对象，如果说没有如切面和工厂中的代理对象，那么二级缓存也就可以解决了，哪怕是只有一级缓存。但为了设计上的合理和可扩展性，所以创建了三级缓存来放置不同时期的对象。
- 通过这样的学习也可以思考 我们在做程序设计时，将要上线的功能是否能全面支撑起业务的拓展和频繁变化的特性，有时候这些设计思路是可以帮我们拓宽更多的技术设计视野。*记得要多加练习!*

第 18 章：数据类型转换

一、锦上添花

值得的，总是在精雕细琢！

在你写的程序开发中，你有为一个类名、方法名、属性名，反复斟酌吗？代码格式间隔大小、编写方式、注释描述不断的提升吗？你有为一个功能逻辑的实现不断的重构吗？**我有，我一直都有**，为了能写好一块代码，甚至会忘记时间从上午到下午，当能实现完成后，会欣赏似的看待自己的代码，*也根本不舍得把他交给别人*！

如果你也是这样的工程师，其实在你不去刻意追求大厂、高薪、好职位的时候，也会把你送到那个位置上去。想不被这个已经有些内卷的行业打下去，那么基本就需要选择一条能沉淀下来核心知识的路径来提升自己，做好长期规划，让以后你的 30 岁有 30 岁的能力，35 岁有 35 岁的经历！

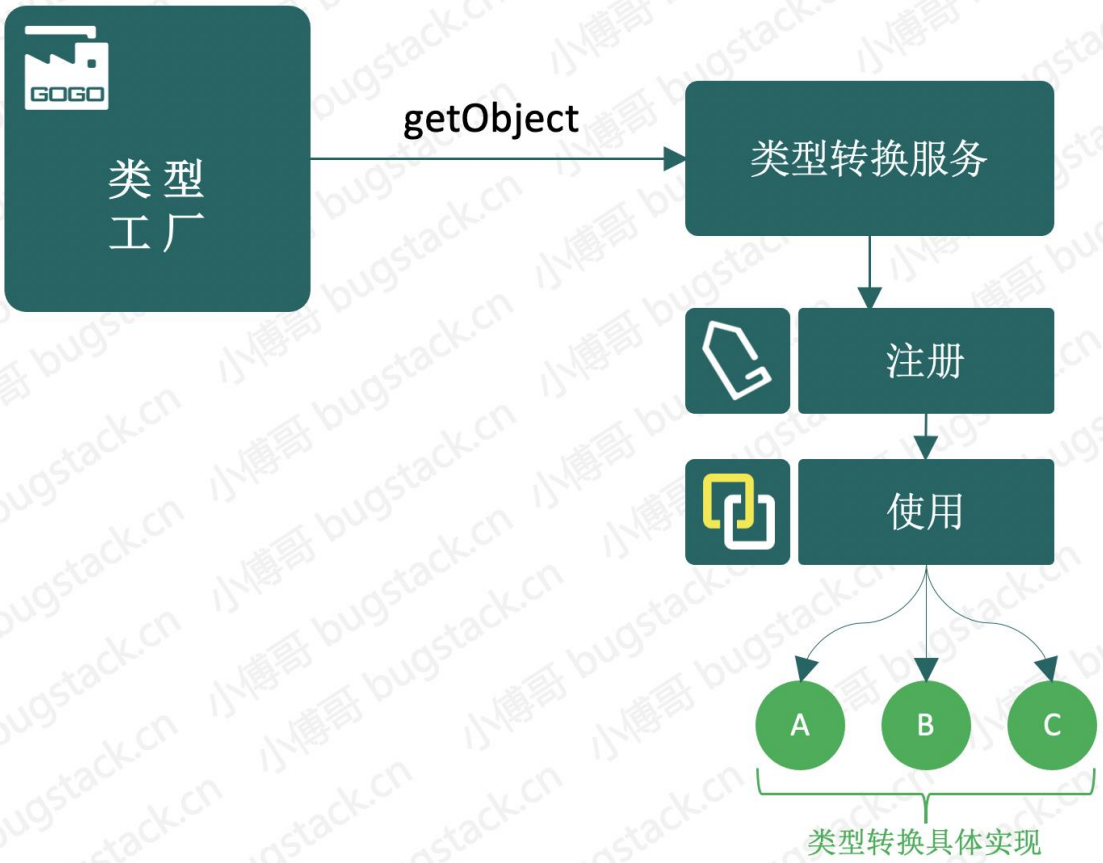
二、目标

其实实现到本章节，关于 IOC、AOP 在日常使用和面试中高频出现的技术点都该涵盖了。那么为了补全整个框架内容的结构，方便读者后续在阅读 Spring 时不至于对类型转换的知识体系陌生，这里再添加一些关于此类知识的实现。

类型转换也可以叫做数据转换，比如从 String 到 Integer、从 String 到 Date、从 Dubbo 到 Long 等等，但这些操作不能在已经使用框架的情况下还需要手动处理，所以我们要把这样的功能扩展到 Spring 框架中。

三、设计

如果我们来把只是看上去一个简单的类型转换操作抽象成框架，那么它就需要一个标准的接口，谁实现这个接口就具备类型转换的具体实现，提供类型转换的能力。那么在有了这样接口后，还需要类型转换服务的注册、工厂等内容，才可以把类型转换抽象成一个组件服务。整体设计结构如下图：



- 首先从工厂出发我们需要实现一个 `ConversionServiceFactoryBean` 来对类型转换服务进行操作。
- 而实现类型转换的服务，需要定义 `Converter` 转换类型、`ConverterRegistry` 注册类型转换功能，另外转换类型的操作较多，所以这里也会需要定义一个类型转换工厂 `ConverterFactory` 各个具体的转换操作来实现这个工厂接口。

四、实现

1. 工程结构

```
small-spring-step-17
└── src
    ├── main
    │   └── java
    │       └── cn.bugstack.springframework
    │           ├── aop
    │           │   ├── aspectj
```

```
    │           │   │   └── AspectJExpressionPointcut.java
    │           │   │   └── AspectJExpressionPointcutAdvisor.java
    │           │   ├── framework
    │           │   │   ├── adapter
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── autoproxy
    │           │   │   │   └── MethodBeforeAdviceInterceptor.java
    │           │   │   ├── AopProxy.java
    │           │   │   ├── Cglib2AopProxy.java
    │           │   │   ├── JdkDynamicAopProxy.java
    │           │   │   ├── ProxyFactory.java
    │           │   │   └── ReflectiveMethodInvocation.java
    │           │   ├── AdvisedSupport.java
    │           │   ├── Advisor.java
    │           │   ├── BeforeAdvice.java
    │           │   ├── ClassFilter.java
    │           │   ├── MethodBeforeAdvice.java
    │           │   ├── MethodMatcher.java
    │           │   ├── Pointcut.java
    │           │   ├── PointcutAdvisor.java
    │           │   └── TargetSource.java
    │           ├── beans
    │           │   ├── factory
    │           │   │   ├── annotation
    │           │   │   │   ├── Autowired.java
    │           │   │   │   ├── AutowiredAnnotationBeanPostProcessor.java
    │           │   │   │   ├── Qualifier.java
    │           │   │   │   └── Value.java
    │           │   │   ├── config
    │           │   │   │   ├── AutowireCapableBeanFactory.java
    │           │   │   │   ├── BeanDefinition.java
    │           │   │   │   ├── BeanFactoryPostProcessor.java
    │           │   │   │   ├── BeanPostProcessor.java
    │           │   │   │   ├── BeanReference.java
    │           │   │   │   ├── ConfigurableBeanFactory.java
    │           │   │   │   ├── InstantiationAwareBeanPostProcessor.java
    │           │   │   │   └── SingletonBeanRegistry.java
    │           │   │   ├── support
    │           │   │   │   ├── AbstractAutowireCapableBeanFactory.java
    │           │   │   │   ├── AbstractBeanDefinitionReader.java
    │           │   │   │   ├── AbstractBeanFactory.java
    │           │   │   │   ├── BeanDefinitionReader.java
    │           │   │   │   ├── BeanDefinitionRegistry.java
    │           │   │   │   ├── CglibSubclassingInstantiationStrategy.java
```

```
    │           │   │   ├── DefaultListableBeanFactory.java
    │           │   │   ├── DefaultSingletonBeanRegistry.java
    │           │   │   ├── DisposableBeanAdapter.java
    │           │   │   ├── FactoryBeanRegistrySupport.java
    │           │   │   ├── InstantiationStrategy.java
    │           │   │   └── SimpleInstantiationStrategy.java
    │           │   ├── support
    │           │   │   └── XmlBeanDefinitionReader.java
    │           │   ├── Aware.java
    │           │   ├── BeanClassLoaderAware.java
    │           │   ├── BeanFactory.java
    │           │   ├── BeanFactoryAware.java
    │           │   ├── BeanNameAware.java
    │           │   ├── ConfigurableListableBeanFactory.java
    │           │   ├── DisposableBean.java
    │           │   ├── FactoryBean.java
    │           │   ├── HierarchicalBeanFactory.java
    │           │   ├── InitializingBean.java
    │           │   ├── ListableBeanFactory.java
    │           │   ├── ObjectFactory.java
    │           │   └── PropertyPlaceholderConfigurer.java
    │           │   ├── BeansException.java
    │           │   ├── PropertyValue.java
    │           │   └── PropertyValues.java
    │           ├── context
    │           │   ├── annotation
    │           │   │   ├── ClassPathBeanDefinitionScanner.java
    │           │   │   ├── ClassPathScanningCandidateComponentProvider.java
    │           │   │   └── Scope.java
    │           │   ├── event
    │           │   │   ├── AbstractApplicationEventMulticaster.java
    │           │   │   ├── ApplicationContextEvent.java
    │           │   │   ├── ApplicationEventMulticaster.java
    │           │   │   ├── ContextClosedEvent.java
    │           │   │   ├── ContextRefreshedEvent.java
    │           │   │   └── SimpleApplicationEventMulticaster.java
    │           │   ├── support
    │           │   │   ├── AbstractApplicationContext.java
    │           │   │   ├── AbstractRefreshableApplicationContext.java
    │           │   │   ├── AbstractXmlApplicationContext.java
    │           │   │   ├── ApplicationContextAwareProcessor.java
    │           │   │   ├── ClassPathXmlApplicationContext.java
    │           │   │   └── ConversionServiceFactoryBean.java
    │           │   ├── ApplicationContext.java
```

```
    │           │   ├── ApplicationContextAware.java
    │           │   ├── ApplicationEvent.java
    │           │   ├── ApplicationEventPublisher.java
    │           │   ├── ApplicationListener.java
    │           │   └── ConfigurableApplicationContext.java
    │           ├── core
    │           │   ├── convert
    │           │   │   ├── converter
    │           │   │   │   ├── Converter.java
    │           │   │   │   ├── ConverterFactory.java
    │           │   │   │   ├── ConverterRegistry.java
    │           │   │   │   └── GenericConverter.java
    │           │   │   ├── support
    │           │   │   │   ├── DefaultConversionService.java
    │           │   │   │   ├── GenericConversionService.java
    │           │   │   │   └── StringToNumberConverterFactory.java
    │           │   │   └── ConversionService.java
    │           │   └── io
    │           │       ├── ClassPathResource.java
    │           │       ├── DefaultResourceLoader.java
    │           │       ├── FileSystemResource.java
    │           │       ├── Resource.java
    │           │       ├── ResourceLoader.java
    │           │       └── UrlResource.java
    │           ├── stereotype
    │           │   └── Component.java
    │           └── utils
    │               ├── ClassUtils.java
    │               └── StringValueResolver.java
    └── test
        └── java
            └── cn.bugstack.springframework.test
                ├── bean
                │   └── Husband.java
                ├── bean
                │   ├── ConvertersFactoryBean.java
                │   ├── StringToIntegerConverter.java
                │   └── StringToLocalDateConverter.java
                └── ApiTest.java
```

工程源码：公众号「bugstack 虫洞栈」，回复：Spring 专栏，获取完整源码

2. 定义类型转换接口

包：cn.bugstack.springframework.core.convert.converter

类型转换处理接口

```
public interface Converter<S, T>  {

    /** Convert the source object of type {@code S} to target type {@code T}. */
    T convert(S source);

}
```

类型转换工厂

```
public interface ConverterFactory<S, R>{

    /**
     * Get the converter to convert from S to target type T, where T is also an ins
tance of R.
     * @param <T> the target type
     * @param targetType the target type to convert to
     * @return a converter from S to T
     */
    <T extends R> Converter<S, T> getConverter(Class<T> targetType);

}
```

类型转换注册接口

```
public interface ConverterRegistry {

    /**
     * Add a plain converter to this registry.
     * The convertible source/target type pair is derived from the Converter's para
meterized types.
     * @throws IllegalArgumentException if the parameterized types could not be res
olved
     */
    void addConverter(Converter<?, ?> converter);

    /**
     * Add a generic converter to this registry.
     */
    void addConverter(GenericConverter converter);
```

```
    /**
     * Add a ranged converter factory to this registry.
     * The convertible source/target type pair is derived from the ConverterFactory
's parameterized types.
     * @throws IllegalArgumentException if the parameterized types could not be res
olved
     */
    void addConverterFactory(ConverterFactory<?, ?> converterFactory);

}
```

- Converter、ConverterFactory、ConverterRegistry，都是用于定义类型转换操作的相关接口，后续所有的实现都需要围绕这些接口来实现，具体的代码功能可以进行调试验证。

3. 实现类型转换服务

cn.bugstack.springframework.core.convert.support.DefaultConversionService

```
public class DefaultConversionService extends GenericConversionService{

    public DefaultConversionService() {
        addDefaultConverters(this);
    }

    public static void addDefaultConverters(ConverterRegistry converterRegistry) {
        // 添加各类类型转换工厂
        converterRegistry.addConverterFactory(new StringToNumberConverterFactory())
;
    }

}
```

- DefaultConversionService 是继承 GenericConversionService 的实现类，而 GenericConversionService 实现了 ConversionService, ConverterRegistry 两个接口，用于 canConvert 判断和转换接口 convert 操作。

4. 创建类型转换工厂

cn.bugstack.springframework.context.support.ConversionServiceFactoryBean

```
public class ConversionServiceFactoryBean implements FactoryBean<ConversionService>
, InitializingBean {

    @Nullable
    private Set<?> converters;

    @Nullable
    private GenericConversionService conversionService;

    @Override
    public ConversionService getObject() throws Exception {
        return conversionService;
    }

    @Override
    public Class<?> getObjectType() {
        return conversionService.getClass();
    }

    @Override
    public boolean isSingleton() {
        return true;
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        this.conversionService = new DefaultConversionService();
        registerConverters(converters, conversionService);
    }

    private void registerConverters(Set<?> converters, ConverterRegistry registry)
{
        if (converters != null) {
            for (Object converter : converters) {
                if (converter instanceof GenericConverter) {
                    registry.addConverter((GenericConverter) converter);
                } else if (converter instanceof Converter<?, ?>) {
                    registry.addConverter((Converter<?, ?>) converter);
                } else if (converter instanceof ConverterFactory<?, ?>) {
                    registry.addConverterFactory((ConverterFactory<?, ?>) converter
);
                } else {
                    throw new IllegalArgumentException("Each converter object must
implement one of the " +
```

```
                            "Converter, ConverterFactory, or GenericConverter inter
faces");
                }
            }
        }
    }

    public void setConverters(Set<?> converters) {
        this.converters = converters;
    }

}
```

- 有了 FactoryBean 的实现就可以完成工程对象的操作，可以提供出转换对象的服务 GenericConversionService，另外在 afterPropertiesSet 中调用了注册转换操作的类。最终这个类会被配置到 spring.xml 中在启动的过程加载。

5. 类型转换服务使用

cn.bugstack.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory

```
protected void applyPropertyValues(String beanName, Object bean, BeanDefinition bea
nDefinition) {
    try {
        PropertyValues propertyValues = beanDefinition.getPropertyValues();
        for (PropertyValue propertyValue : propertyValues.getPropertyValues()) {
            String name = propertyValue.getName();
            Object value = propertyValue.getValue();
            if (value instanceof BeanReference) {
                // A 依赖 B，获取 B 的实例化
                BeanReference beanReference = (BeanReference) value;
                value = getBean(beanReference.getBeanName());
            }
            // 类型转换
            else {
                Class<?> sourceType = value.getClass();
                Class<?> targetType = (Class<?>) TypeUtil.getFieldType(bean.getClas
s(), name);
                ConversionService conversionService = getConversionService();
                if (conversionService != null) {
                    if (conversionService.canConvert(sourceType, targetType)) {
                        value = conversionService.convert(value, targetType);
```

```
                    }
                }
            }
            // 反射设置属性填充
             BeanUtil.setFieldValue(bean, name, value);
        }
    } catch (Exception e) {
        throw new BeansException("Error setting property values:
" + beanName + " message: " + e);
    }
}
```

- 在 AbstractAutowireCapableBeanFactory#applyPropertyValues 填充属性的操作中，具体使用了类型转换的功能。
- 在 AutowiredAnnotationBeanPostProcessor#postProcessPropertyValues 也有同样的属性类型转换操作。

五、测试

1. 事先准备

```
public class Husband {

    private String wifiName;

    private Date marriageDate; // 添加一个日期类的转换操作

    // ... get/set
}
```

转换时间的操作类

```
public class StringToLocalDateConverter implements Converter<String, LocalDate> {

 private final DateTimeFormatter DATE_TIME_FORMATTER;

 public StringToLocalDateConverter(String pattern) {
  DATE_TIME_FORMATTER = DateTimeFormatter.ofPattern(pattern);
 }

 @Override
 public LocalDate convert(String source) {
  return LocalDate.parse(source, DATE_TIME_FORMATTER);
```

```
    }

}
```

- Husband 是一个基础对象类设置了时间属性，之后再添加一个类型转换的操作用于转换时间信息。

2. 属性配置文件

spring.xml

```
<bean id="husband" class="cn.bugstack.springframework.test.bean.Husband">
    <property name="wifiName" value="你猜"/>
    <property name="marriageDate" value="2021-08-08"/>
</bean>

<bean id="conversionService" class="cn.bugstack.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters" ref="converters"/>
</bean>

<bean id="converters" class="cn.bugstack.springframework.test.converter.ConvertersFactoryBean"/>
```

- 配置基础 Bean 对象，设置属性的日期，同时再添加类型转换的服务和自己实现的 ConvertersFactoryBean

3. 单元测试

```
@Test
public void test_convert() {
    ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext("classpath:spring.xml");
    Husband husband = applicationContext.getBean("husband", Husband.class);
    System.out.println("测试结果：" + husband);
}

@Test
public void test_StringToIntegerConverter() {
    StringToIntegerConverter converter = new StringToIntegerConverter();
    Integer num = converter.convert("1234");
    System.out.println("测试结果：" + num);
```

```
}

@Test
public void test_StringToNumberConverterFactory() {
    StringToNumberConverterFactory converterFactory = new StringToNumberConverterFa
ctory();
    Converter<String, Integer> stringToIntegerConverter = converterFactory.getConve
rter(Integer.class);
    System.out.println("测试结果：" + stringToIntegerConverter.convert("1234"));
    Converter<String, Long> stringToLongConverter = converterFactory.getConverter(L
ong.class);
    System.out.println("测试结果：" + stringToLongConverter.convert("1234"));
}
```

测试结果

```
测试结果：Husband{wifiName='你猜', marriageDate=Sun Aug 08 00:00:00 CST 2021}

Process finished with exit code 0
```

- 这个测试内容还是比较简单的，可以自行验证结果，虽然最终的结果看上去比较简单，但整个框架结构实现设计还是蛮复杂的，把这么一个转换操作抽象为接口适配、工厂模型等方式，还是很值得借鉴的。

六、总结

- 本章节实现的类型转换操作如果只是功能性的开发，就像你自己承接的需求那样，可能只是简单的 if 判断就搞定了，但放在一个成熟的框架要中要考虑的是可复用性、可扩展性，所以会看到接口的定义、工厂的使用等等设计模式在这里体现。
- 最后非常感谢你能坚持学习到这个章节，如果你在学习的过程也是每一个章节都是对着文章、写着代码代码、调试着 bug，感悟着设计，那么你一定会在这个过程中得到很多很多，以后再阅读 Spring 的源码也就不会感觉那么难了。

结尾

感谢，你对本书的支持，可能书中会因作者水水平有限，有一些描述不准确或者错字内容。欢迎提交给我，也欢迎和我讨论相关的技术内容，作者小傅哥(fustack)，非常愿意与同好进行行流，一起提升技术。

本书到这里还不是结束，接下来还会继续编写更多的技术内容。所有的内容的输出都是一个目的，让更多的人对知识能做到，让懂了就是真的懂！