广州航海学院

毕业设计（论文）

基于 java 注解与反射的 ioc 对象管理容器框架开发

专业:	计算机科学与技术	班级:	计算机 174 班
姓名:	郑锦龙	学号:	201715210426
导师:	沈明	职称:	副教授

信息与通信工程学院

二〇二一年五月

摘　　要

在较为复杂的项目中，对象之间纵横交错的依赖关系导致初始化对象成本大，对象被传递多次才能被使用，在传递的过程中也造成了不必要的耦合。并且在传统项目中一个行为具有多种实现时，也必须要依赖具体实现才能构建出具体的行为对象。

为了解决在开发中遇到的以上问题，本文基于控制反转（Inversion of Control）的设计原则，以 java 的注解与反射为主，并使用常用的设计模式作为辅助，设计出的一款对象实例管理容器框架。可将对象注册到对象管理容器中，并且对象构建时可以从容器中获取依赖对象进行依赖注入（Dependency Injection），完全削减了对象因依赖关系链路过长的构建成本。并且因为 java 的向上转型的特性，在依赖注入时可以将具体实现注入到抽象声明中，达到了遵循依赖倒置原则（Dependence Inversion Principle）的目的，从而降低耦合等级，使本身对具体实现耦合降级为对抽象行为耦合。

由于框架是否易于扩展是至关重要的，所以本文还展现了框架的扩展性，其中包含面向切面编程 AOP（Aspect Oriented Programming）的实现、使用 Netty 实现的 Web 服务器、基于动态代理实现的面向接口的外部 http 请求器。还有一些比较细化的功能：yaml 配置载入与管理、对实例的注入完成初始化方法调用与销毁方法调用等。

将对象注册到对象管理容器中必然会获得的优化：代码中最多只需进行一次构建对象、对具体实现的依赖大幅度被削减、传递依赖代码完全被削减等。程序将会获得以下能力：AOP 切面增强、Web 服务器、外部服务请求器、根据配置文件初始化项目等。

关键词：java 注解；控制反转；依赖注入：面向切面：依赖倒置原则

ABSTRACT

In some more complex projects, the crisscrossing dependencies between objects lead to the high cost of initializing the object, the object is passed on several times before it can be used, and the unnecessary coupling is also caused in the process of transferring. Additionally, in tradition projects, when a behavior has multiple implements, it's also essential to rely on the concrete implements to build concrete behavior objects.

In order to deal with the above problems encountered in the development process, this essay, based on the Inversion of Control design principle, mainly depends on Java annotations and reflection, and uses common design patterns as auxiliary means, to design an object instance management container framework. This kind of framework allows objects to be registered into the object management container, and dependent objects can be obtained from the container for dependency injection during object construction, which completely reduces the construction cost of the overlong link because of the object dependencies. Besides, due to the upcasting of Java, the concrete implements can be injected into the abstract declarations during dependency injection, which can achieve the purpose of following the Dependence Inversion Principle, thus, reduce the coupling level, and downgrade the coupling resulting the concrete implements coupling being downgraded to the abstract behavior coupling.

Since the issue whether the framework is easy to extend or not is critical, this article also shows extensibility of the framework, including implement of AOP (Aspect Oriented Programming), a Web server implemented by Netty, and an external HTTP requester which is dominated by dynamic proxy. There are also some more detailed functions, like YAML configuration load and management, completing initialization call and destruction of instance injection, etc.

Optimizations that are guaranteed by registering an object in the object management container are as follows: the object will only be built at most once in the code, the dependency on the concrete implement is drastically reduced, the transitive dependency code is completely reduced, and so on. And capabilities that application will acquire include: AOP enhancement, Web server, external service requesters, and initializing projects according to configuration files, etc.

Keywords: Java annotation, control inversion, dependency injection, Aspect

Oriented, dynamic proxy, Dependence Inversion Principle

目　　录

基于 java 注解与反射的 ioc 对象管理容器框架开发

第 1 章 绪论

1.1 课题研究的背景及意义

在开发的过程中，类与类之间的依赖会随着项目的持续扩大而变得复杂起来，当多个类依赖同一个类时，我们就需要将这个类在多个类中实例化，若类在构造实例过程中对其他的类又进行依赖，使得构造时仅仅为了满足依赖类的构造参数需求而又引入新的依赖类。这会导致类之间的依赖关系紊乱，且因为实例化代码的重复而使代码看起来不够整洁。

并且，在开发中也要遵循 DIP（依赖倒置原则），DIP 一直以来都是面向对象设计中所要遵循的原则，上层模块不应依赖底层模块，应依赖于抽象[1]。而在实际开发中，对象不可避免的依赖于某个抽象的具体实现来完成相应的动作，在创建实例时就需要引入实现类，实际上也违背了 DIP。其中也有使用抽象工厂和代理模式来实现层间解耦。

在巧妙的使用某些设计模式组合的方式去遵循这些规则时，其实这不仅使开发人员无法全身心投入到业务的设计上，并且在代码上这些优化本身也是一种“高级”的冗余。为了开发时不必花费大量精力投入到对这些依赖结构解的考虑。IOC（控制反转）容器概念与 DI（依赖注入）方法就出现了，IOC 容器的实现与 DI 的搭配能将与业务功能本身无紧密联系且重复的代码解耦或去除，并且将类与类的依赖关系转变为类与容器的依赖。这有利于我们提升开发效率和降低维护成本。

1.2 IOC 容器的研究现状及存在问题

在 2001 年时，Rod Johnson 在《Expert One-on-One J2EE》提出 EJB 入侵式的一些缺陷，并提了一个更简单的解决方案，即 spring 的前身 interface21[2]。

而 IOC 这个概念 2004 年才被 Martin Fowler 正式命名，并提出规范组件、服务、注入，服务定位器等名词的使用[3]。

在应用上，围绕着 IOC 应用的各种框架已经存在很多年，例如 spring 在实际应用与开发者社区生态方面已经较为成熟，在 2004 年发布并迭代至今的第五个大版本，深受大家喜爱，现在已加入了各大高校教材，成为家喻户晓的必学框架。

在理论上早在 2005 年王咏武在期刊上发表的《向依赖关系宣战——依赖倒置、控制反转和依赖注入辩析》就已经对控制反转的概念，何为依赖倒置，依赖注入讲述的非常清楚。其中提到：分离接口和实现是人们有效地控制依赖关系的最初尝试，而纯粹的抽象接口更好地隔离了相互依赖的两个模块，依赖倒置和控制反转原则从不同的角度描述了利用抽象接口消解耦合的动机，Gof 设计模式正是这一动机的完美体现[4]。其中围绕着 spring 框架举例各种设计模式在 spring 中的应用。

现有的框架系统普遍存在情况，比较完善的框架环境搭建过于复杂繁琐，比如 spring 的 xml 配置文件，随着 json，yaml 等各种更简洁的文件格式，xml 啰嗦的语法也不应该成为主流配置文件格式。

而一些轻量级的，例如 IOCFacotry，Guice 等框架，使用文档与可扩展性等说明书也还未完善。

再者，各种研究中注重概念研究而并没有真正讲述如何在具体语言中实现。

1.3 论文研究内容

本文目的为利用 java 注解与反射与众多 GoF 设计模式来研究在 java 中的 IOC 容器与 DI 的纯注解形式实现。研发出一个纯注解驱动开发 IOC 容器框架。除了指定 YML 配置文件外，无需再引入其他配置文件。着重研究方向在以下几点：

1. 如何进行编译后文件与 jar 包扫描，对各种注解行为打下基础。在扫描中如何能使框架能具有良好的拓展性，可能会有复用的代码来使用户编写代码继承或实现某些接口去拓展他们的注解处理方案。
2. 如何定义容器，使用何种数据结构。
3. 如何注册组件，研究并设计扫描后的包怎么注册到容器中，并由容器管理。
4. 如何注入依赖对象，经过包扫描识别特定注解，从容器中寻找对应已注册组件，进行实例化注入。研究并解决对象的注入中可能存在循环依赖，构造循环依赖，原型模式下注入延迟性等问题。
5. 如何在实例创建前后是否可对组件实例创建前和创建后的增强动作。
6. 如何使框架拥有面向切面编程的能力，使注入的实例中执行方法时可以依照业务进行前置，后置，环绕的增强处理[5]。
7. 如何实现一个 web 服务端，即类似 spring mvc 的功能，研究如何返回视图，视图的参数填充，数据序列化，文件流传输等功能如何实现[6-7]。
8. 参照 SpringCloud 中的 feign 接口注解形式的拓展，研究有关动态代理的投辫断流，但研究只注重如何使系统调用其他服务的接口像调用方法一

样简单，不深入研究分布式系统的内容。

1.4 论文结构

本论文一共分为六章，具体介绍如下：

第一章：绪论部分先介绍了 ioc 对象管理容器框架的背景及意义，再简述了现有研究存在的问题，最后对论文研究内容进行了介绍。

第二章：对本文使用得较多的理论知识进行简要的讲解。

第三章：设计整个框架的设计方向和基础底座，保证让整个系统像搭积木一样由下到上搭建成功。

第四章：设计框架的核心部分。

第五章：由框架的核心部分作为基础，设计框架的扩展部分。

第六章：框架各个模块的测试举例。

第 2 章 基础知识理论

2.1 设计模式

设计模式是一套在为了可重用并提高可读性，在特定环境下特定问题的处理方法集合。设计模式主要分为三种类型：创建型、结构型、行为型。本节介绍一些会出现在本文中的设计模式[8]。

单例模式：在确保程序运行的过程中，单例类绝对只有单个实例被创建。通常用在单窗口程序、全局上下文容器、全局计数器等。

原型模式：通过预先实例生成的实例为模板，通过复制来产生一个新的实例。通过用在实例难以构建或需要在不知道构建方式的情况下生成一模一样但地址不相同的实例。

工厂模式：将实例交给一个工厂类来生成，在工厂类中决定生成一个对象的具体流程。在需要生成实例时调用工厂方法来代替直接新建对象。

策略模式：整体的替换算法，可以使一个类的行为通过依赖接口的形式来实现动态的替换。

适配器：根据实际需求将多种不相容的类连接起来工作，来达到拥有将多种功能杂糅到一个类中实现。适配器分为继承适配与委托适配。

代理模式：代理类帮助被代理类去完成处理，也可以通过被代理类的方法来实现原方法调用。一般用作对某个类的行为增强，可严格遵循开闭原则来增加功能，有对被代理类的隔离保护的作用。分为静态与动态代理。静态代理需要声明是对具体的一个类代理。而动态代理是通过反射机制，在运行时才确定需要被代理的对象。

职责链模式：将整体切分为多个职责分发给多个担责对象，该模式可弱化请求方和处理方之间的联系，让其各自成为独立复用的一个组件。通常用于处理 web 请求时的过滤器实现，其内部请求方一直找到需要拦截该请求的担责对象，该请求就被拦截了。

2.2 控制反转与依赖注入

由图 2.1 中我们可以看到，在简单的四个类中，基于他们的互相依赖的程度较高，存在的耦合较强的相互引用的关系，在大型的系统中的相互引用复杂程度远远不止这四个类这么简单。在互相引用的过程中，还要对满足依赖的每个类的依赖来构建对象。假如类 A 依赖的类 D，但类 D 的构建又依赖类 B，而类 A 不依赖类 B 却因为要构建类 D 的对象不得不依赖类 B。而该图也表现出了循环依

赖的问题。类 B 依赖类 D 的同时类 D 也同时依赖类 B。

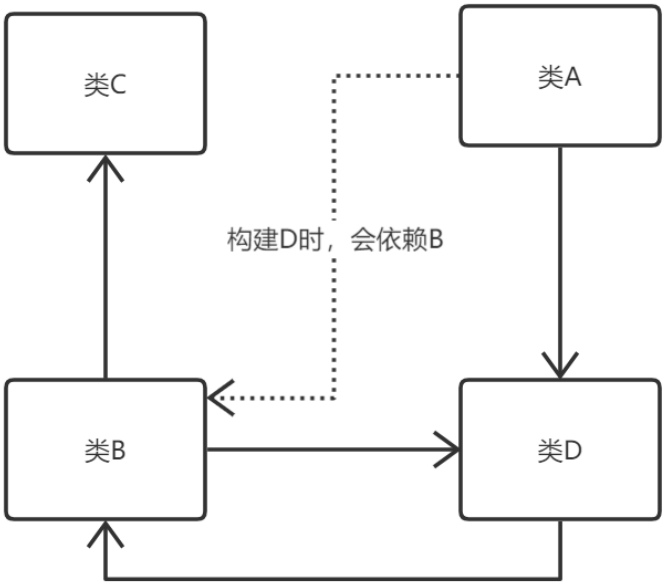

图 2.1 无 IOC 容器中依赖关系（箭头为依赖方向）

IOC 是一种面向对象中的设计原则，实际上是由依赖注入和依赖查找共同组成的一种实现方式[9]。借助容器与对象注入来对具有依赖关系对象之间的解耦[10]。在引入了 IOC 容器后，各个类只要与容器相互建立联系就可以了。如图 2.2 所示，类 D 只要向容器中注册自己的实例，当类 A 依赖类 D 时，只需要从容器中获取类 D 对应的实例并注入到类 A 的实例对象中。这时候 A 中并不再需要依赖 B 去构建 D。并同时解决了 B 与 D 的循环依赖问题。

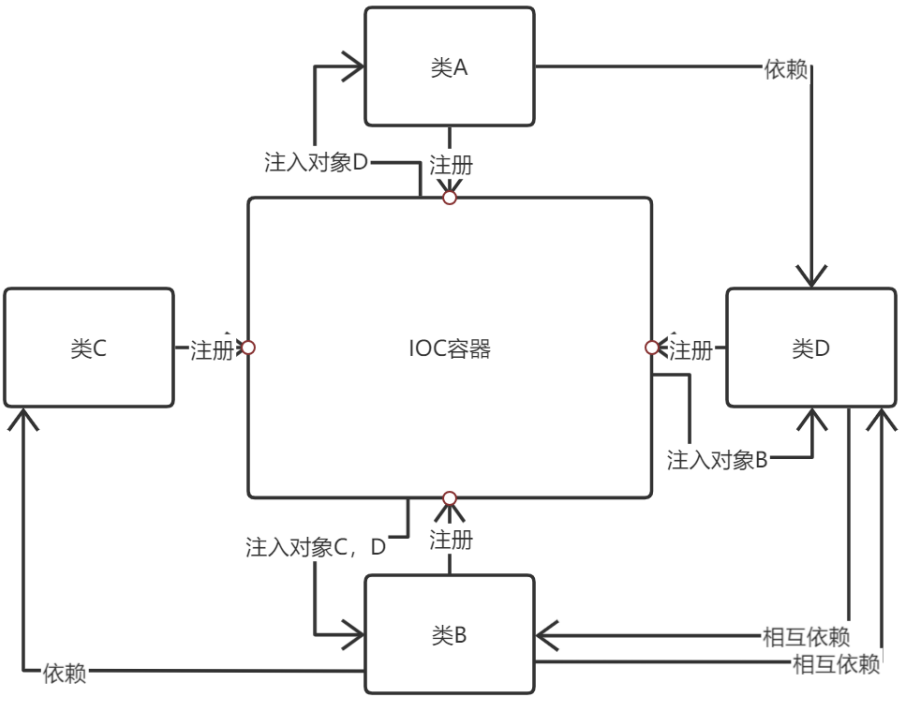

图 2.2 引入 IOC 容器后的通过依赖注入的构建行为

如图 2.3 所示，因为拥有多态这个性质的原因，我们可以将具体实现的子类 D、E 对象从容器中取出，通过反射的方式赋值给在类 A、B 中声明的抽象父类 D1 变量来避免对具体实现的依赖，这就是依赖注入。

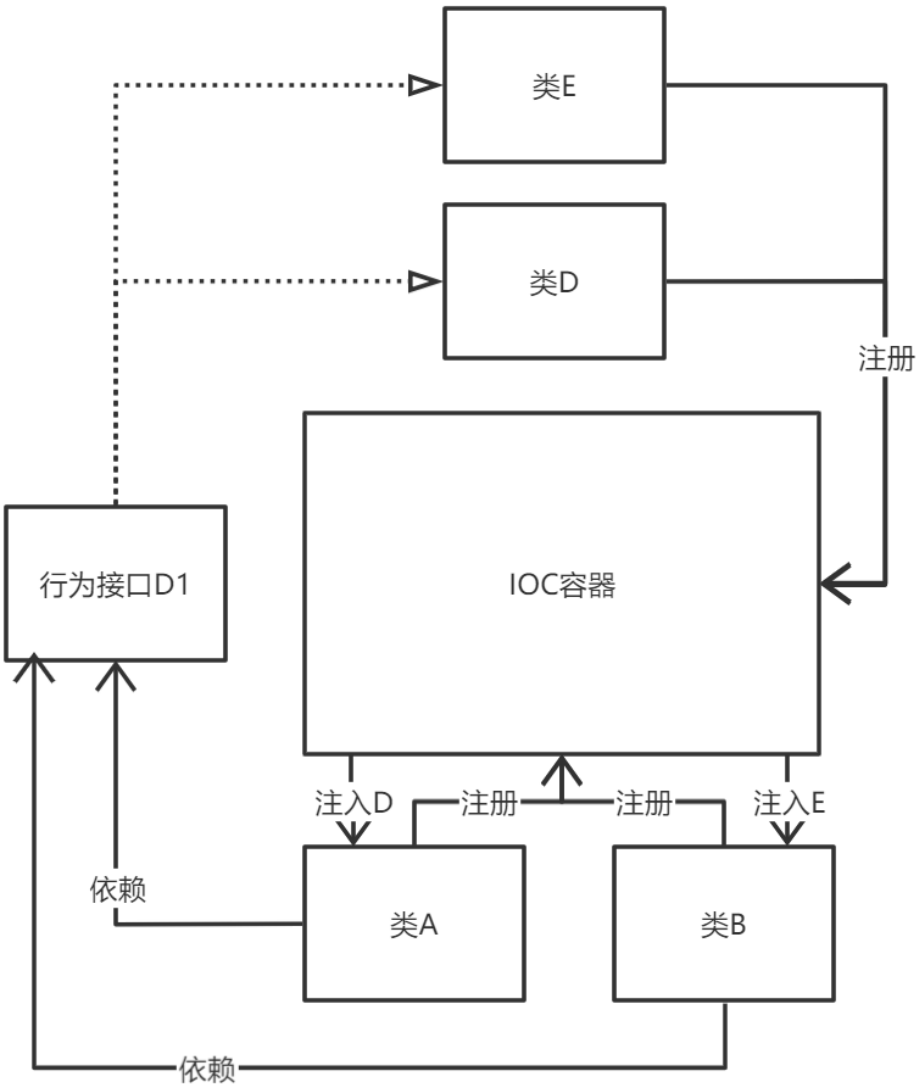

图 2.3 依赖注入使我们只需依赖抽象

2.3 本章小结

讲述了本文中的使用到的开发方法和核心思想，粗略的描述了开发中所需要使用到的设计模式，描述依赖注入的必要性。

第 3 章 基础架构设计

3.1 工程模块化

本文使用 maven 包管理工具来管理项目，项目根据模块层次划分来代表包层次划分，根据职能的不同将代码分为多个模块，每个模块相当于一个项目。可以将通用的部分抽离出来作为一个模块，其他模块通过依赖这些通用模块来达到对代码的复用的效果。在依赖这些模块时，可有选择的只对某些职能模块进行依赖。

框架需要呈现扩展性，而在二次开发进行扩展时必然会导致项目越来越庞大，在通过其扩展性进行水平延展时，可以通过依赖框架的最核心的通用部分来进行扩展，让开发者不必将所有的扩展引入，而是做选择去依赖这些扩展功能。

本文项目中的模块分布与功能介绍如表 3.1 所示，模块依赖关系如图 3.1 所示。

表 3.1 模块功能介绍

模块名	作用描述
springz-annotation	注解模块，用于存放 core 依赖的注解类
springz-util	工具包模块，编写可在各模块之间复用的工具类
springz-scanner	全局包扫描模块，core 模块是在这个模块提供的包扫描基础上编写
springz-core	框架核心模块，包含着框架的核心功能
springz-aop	在 core 的基础上拓展出的面向切面编程模块
springz-web	在 core 的基础上拓展出的提供 http 协议的 web 模块
springz-feign	在 core 的基础上拓展出的用于调用网络中其他服务的模块

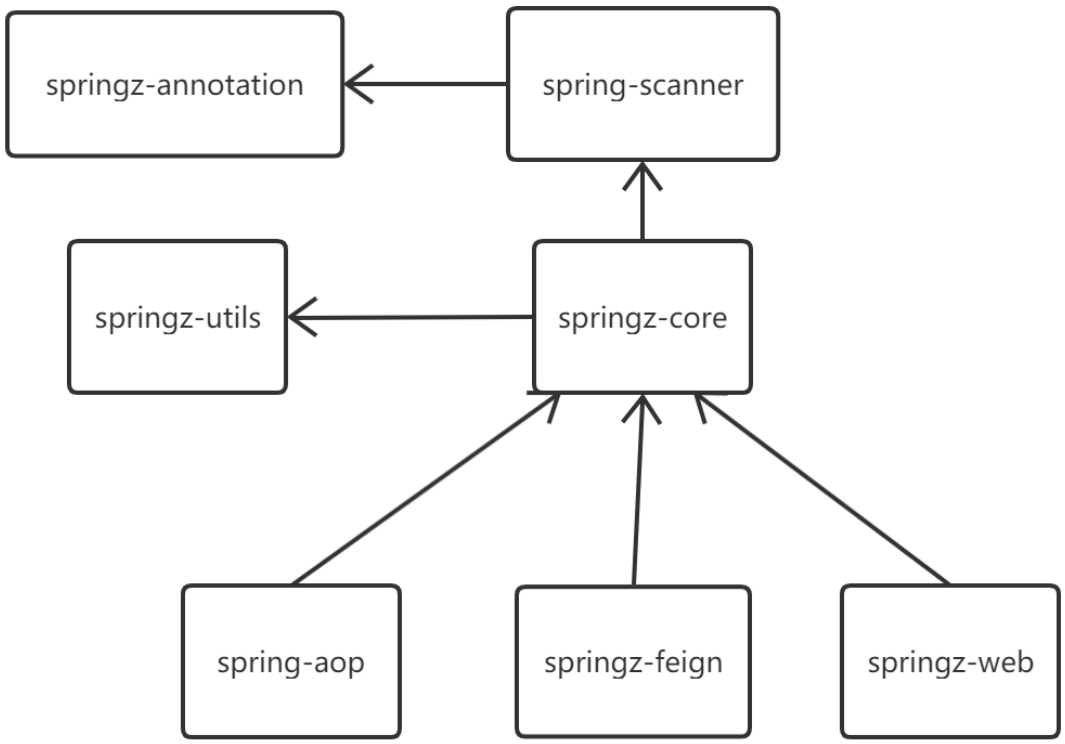

图 3.1 项目模块之间的依赖关系

3.2 基于注解驱动

注解（Annotation）一般有三个作用：编写文档、代码分析、编译检查。但其实注解并不能对程序造成影响，只有在与配套的工具一起使用时，才能通过工具对这些注解信息进行处理和访问。

基本注解（元注解）有五种：@Target、@Retention、@Documented、@Inherited、@Repeatable，其余的注解都属于自定义注解。

我们可以通过自定义注解来标注在需要被我们程序处理的类上，编写程序来认知这些类上的注解，每个自定义注解都可以对应一个自定义处理，这样注解就是我们的处理过程与被处理类连接的纽带。

所以，我们最后的程序就会变成容器、各类分别与注解耦合。如图 3.2 所示，我们规定@Component 是用在需要被注册类上，并且 IOC 容器中负责注册的处理方法会对这些被标注为@Component 的类进行处理。举个例子，看以下代码所示，根据以上规定，这相当于声明了 People 这个类需要被注册到容器中。

```
@Component
public class People {
    private String name;
}
```

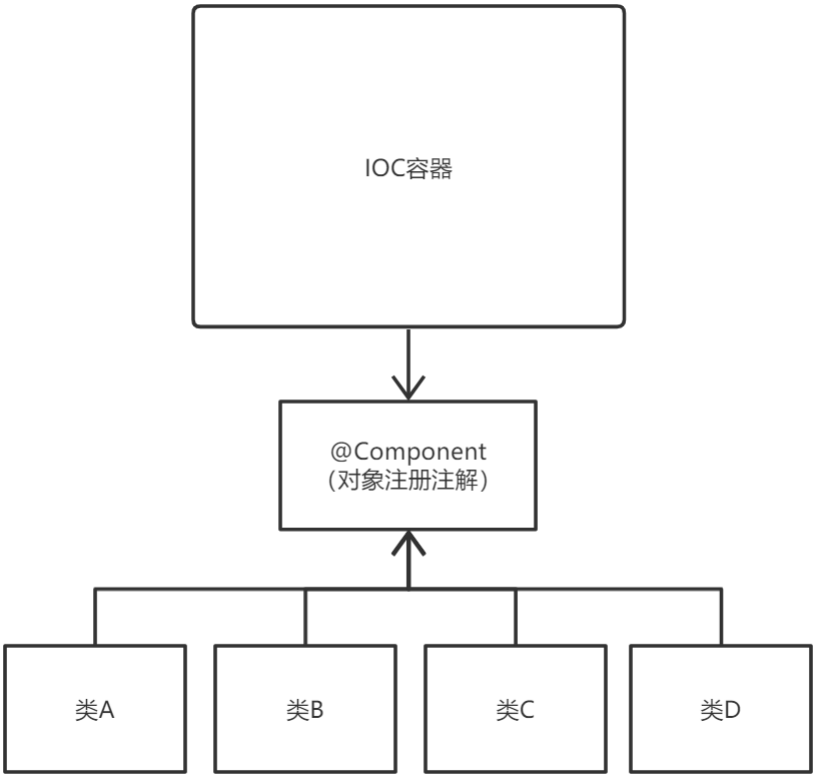

图 3.2 各类与容器处理程序都依赖自定义注解

在这里先交代本文中可能会出现关于注解中容易造成误解的概念：

一、某类上包含某个注解，该“包含”的含义指的是直接包含与间接包含。直接包含即是直接标注在类上，间接包含指的是在该类上一直往下深度查找其注解的注解，如果能找到，就是该类间接包含该注解。

二、注解中没有字段的声明，但每个方法都能获取到元数据值的，所以本文中有可能出现用属性或字段的字眼来代替方法。文中出现的注解属性或注解字段指的都是注解中的方法。

3.3 包扫描与扫描行为（scanner 模块）

基于注解驱动下，必须要有一个将注解下的类与处理逻辑相关联的设计，扫描器模块（springz-scanner）在这里充当对每个类进行扫描的角色。扫描器 Scanner 是框架的核心组件之一，扫描器初始化时可以设置扫描范围。下面新建了一个对 org.example.controller 包与 org.example.service 包扫描的扫描器。

```
Scanner scanner = new Scanner(“org.example.controller”,“org.example.service”);
```

扫描器每次扫描都可以配置一个 ScanAction（扫描行为），扫描行为在构造时，会定义一个 ScanFilter 来过滤掉一些不符合条件的类。

程序在编译器运行时与打包成 Jar 时文件结构是不一致的，即扫描器扫描时

存在扫描文件与扫描 Jar 包的差异，差异大致以下两点：。

1. 扫描文件时可以通过初始化得到扫描路径递归构造 File 获取文件内容，而打包成 Jar 后需要通过类加载器获取 JarFile 来获取 JarEntry。这里的 JarFile 相当于文件夹，JarEntry 相当于文件夹下的每个 File。
2. 扫描出来构造的 File 对象中的 name 属性是文件名，而 JarEntry 对象的 name 属性是是文件路径+文件名。

扫描器在扫描到一个类后，会构造一个类信息 ClassInfo，在经过扫描行为的过滤器过滤后，会接收这个 ClassInfo 进行处理，一般这些处理大都为通过识别类上注解，然后加工并注册到容器中，扫描器中的扫描行为其实就实现了我们上文所说的，通过注解将处理过程与类相关联。

扫描器是复用型组件，一个扫描行为的执行都要进行一次全局扫描，若存在多个扫描扫描行为需要被执行时，就需要多次扫描，这些扫描是有顺序的，ScanAction 继承 Order 接口来返回这个行为的序号，所有的 ScanAction 会被 ScanActionManager 管理调度，Order 序号越小就越早被执行。因为多次执行扫描的原因，需要有一个缓存链表来保存第一次扫描得到的 ClassInfo。除了第一次扫描需要扫包以外，往后的扫描只需要从缓存中获取 ClassInfo 来传递给扫描行为。表 3.2 为 ClassInfo 的属性列表。

表 3.2 ClassInfo 的属性列表

属性	类型	描述
definitionName	String	类别名
className	String	类全名
clazz	Class<?>	类对象

整个包扫描流程如图 3.3 所示，先进行扫描类型的判断是扫描文件还是扫描 jar 包，根据不同的类型来进行不同的扫描处理，处理完的结果封装成 ClassInfo，再根据 ClassInfo 的信息来进行过滤器 ScanFilter 的过滤，最后才执行扫描行为 ScanAction。

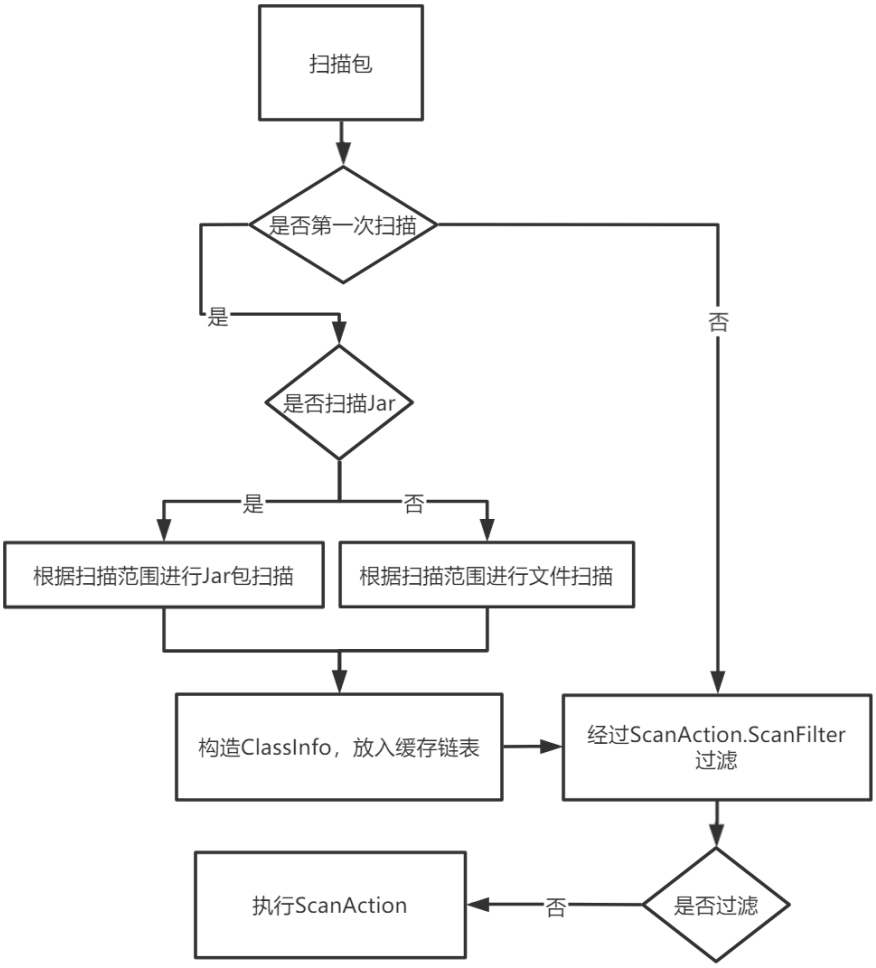

图 3.3 包扫描流程图

3.4 本章小结

介绍了基础架构的选择与开发。其中描述了抽象的注解驱动思想，根据代码职能来划分成多模块项目，解释了包扫描作为框架基础底座的具体处理过程。

第 4 章 框架核心设计（core 模块）

springz-core 模块作为框架的核心模块，其中组件注册与注入，后置增强，对象注入完毕初始化，对象销毁，加载拓展模块等功能都在这个模块里完成。整个框架的生命周期都在 core 中进行控制。框架的生命周期如图 4.1。

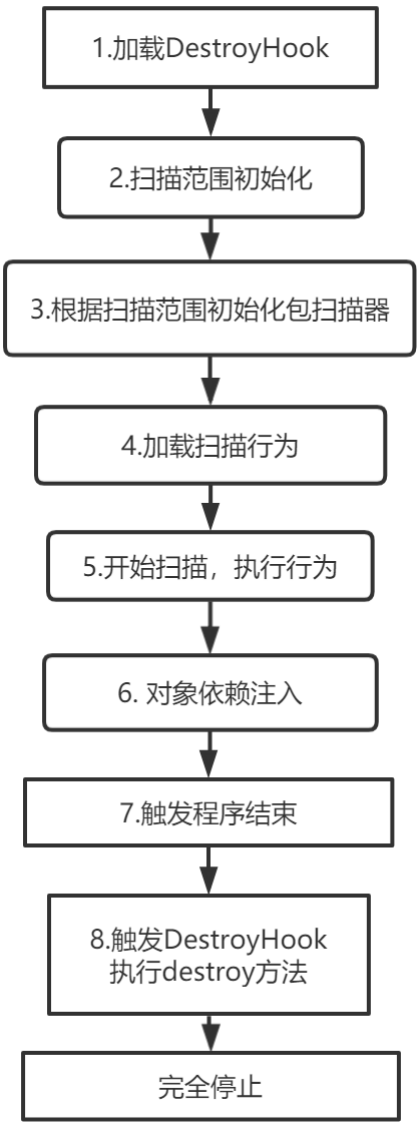

图 4.1 生命周期各阶段

1. 加载容器销毁时触发的钩子函数。
2. 对启动类目录下的包扫描，获取需要容器管理的包路径目录。
3. 根据第 2 步获取的包路径目录来初始化全局扫描器。
4. 使用全局扫描器进行单次扫描，通过判断是否实现 ScanAction 来加载扫

描行为。

5. 使用全局扫描器进行多次扫描，这时候的多次扫描是因为需要遍历执行第 4 步的扫描行为。
6. 对象注册、对象注入、加载与执行后置增强就在本阶段被执行的。
7. 判断到用户执行结束指令：ctrl+c、编译器结束调试时发送的结束指令、kill 指令等。
8. 触发第一步加载的函数，其中@Destroy 注解下的方法就在这一步被执行的。

4.1 加载配置文件

采用懒加载的形式进行配置文件加载。在读取配置文件 Yaml 对象后将其序列化成 Map 对象驻留在 PropertiesContainer 中。并且支持读取多个配置文件，默认读取 classpath:application.yml，多个配置文件通过以配置文件名为 key，读出的配置 Map 为 value 存储到 Map 中。读取配置文件时，在获取配置项时可以通过传递 Class 对象来将配置项自动转换成你需要类。在需要获取深层级的配置项时，通过“.”号来连接各层级的 key 来获取对应的配置项值。

使用例子：通过以下代码从 classpath:application.yml 文件的配置获取 user 项下的 name 的值，转换成 String 类型。

```
PropertiesContainer.getPropertis(“user.name”,String.class,“classpath:application.yml”)
```

在 PropertiesContainer 中加载的配置将用在类注册扫描行为适配器与后置处理器执行的注入配置项到对象中，在本文 4.3 节中会对这两个处理器进行描述。通过在@Bean 注解下的方法或@Component 注解下的类添加@Properties 注解来声明这是一个需要注入配置项的类，@Properties 中可以声明是从哪个配置文件中读取什么前缀的配置项。@Properties 属性如表 4.1 所示。

表 4.1 @Properties 的属性表

属性名	类型	描述
source	String	从哪个配置文件中获取
prefix	String	配置文件前缀层级的 key

从 source 文件获取 key 为 prefix 加上字段名的配置项的值注入到该字段上。也可以使用@Value 来对字段打上别名，在获取注入配置项时，会尝试获取@Value 注解的值，通过正则表达式(?={(.*)})来获取${subKey}中的 subKey 代替字段名的来进行拼接。下面展示通过类注册扫描行为适配器和后置处理器注入配置项的使用例子。

以下代码是在类上进行@Properties 标注，通过类注册扫描行为适配器中进

行配置项注入。

```
@Component
@Properties(prefix = "user")
public class User {
    private String name;
}
```

以下代码是在方法上进行@Properties 标注，通过后置处理器进行配置项注入。

```
@Configuration
public class TestPropertyConfig {
    @Bean
    @Properties(prefix = "user")
    public User lisi() {
        return new User();
    }
}
```

4.2 注册器与扫描行为适配器

注册器（Registrar）是框架的核心组件之一，负责根据需要注册的类解析出的组件信息封装成 BeanInfo 后注册进组件容器（BeanContainer）中，其中属性包括：别名，注册对象实例（并不在注册器中构造这个实例，并且注入前会对该实例做处理），注入策略（单例或原型），对象实例的类，注册时传入的类（这个类会与对象实例的类相同，也有可能不同。因为根据类注册时也要将其父类、实现的接口类注册，这样才能实现根据父类或接口类注入其子类对象），如表 4.1 为 BeanInfo 的属性列表。

表 4.2 BeanInfo 的属性列表

属性	类型	描述
definitionName	String	注册别名
bean	Object	注册对象实例
clazz	Class<?>	对象实例的类
realClass	Class<?>	注册时传入的类（有可能是 class 的父类或实现接口类）
className	String	注册类全名
scope	String	注入策略（单例或原型）

在本文中可能会出现各种各样的注册器，其实都是继承抽象 Registrar 实现的各自的注册流程。本节只讲述在 Registrar 类中的最基本注册情况与注册流程。

BeanContainer 中存放组件注册的数据结构其实是以 definitionName 为 key，BeanInfo 为 value 的一个 HashMap。在一次注册方法调用中，其实会涉及到两次注册行为的调用。第一次是获取 bean 的类名来作为 key，将 BeanInfo 注册到 BeanContainer 中。第二次是获取 bean 的父类与实现接口的集合进行与第一次注册处理步骤一致的递归注册，注册时会将 BeanInfo 拷贝并将 realClass 修改为注册时的父类或接口类名。图 4.2 为基本注册流程。

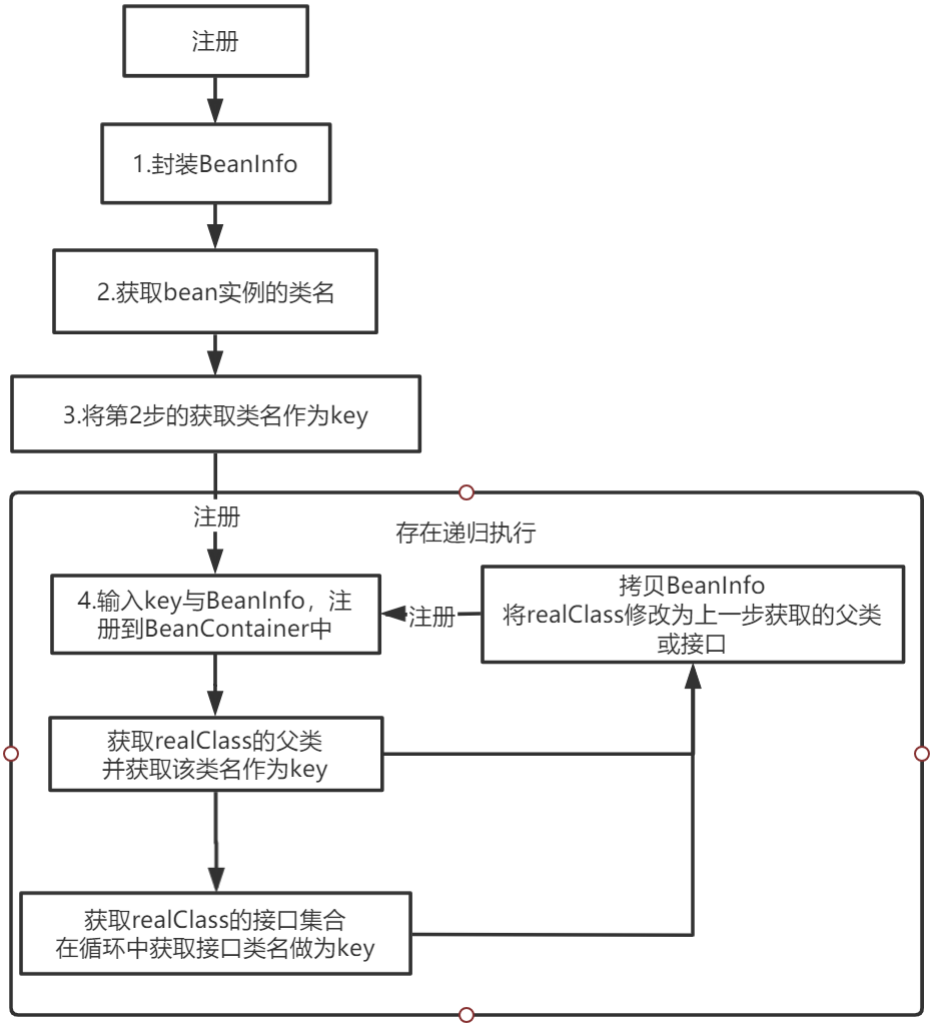

图 4.2 注册流程

在扫描时执行扫描行为时，需要在行为处理中同时将组件注册到容器中，使用继承适配来使扫描行为具有注册器的功能，这就是扫描行为适配器。具体做法，继承 Registrar 后，实现 ScanAction 接口重写 action 方法中调用 register 方法就可以在扫描中将组件注册到容器中。图 4.3 所示为类图。继图 3.3 中执行 ScanAction 后的流程如图 4.4 所示。

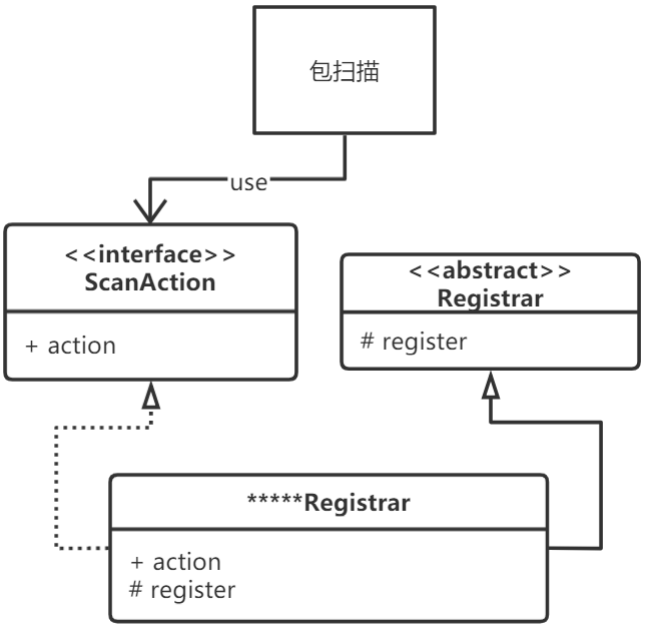

图 4.3 注册行为适配器类图

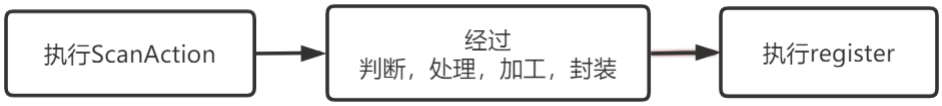

图 4.4 扫描行为适配器中的过程

4.3 核心扫描行为设计

springz-core 模块围绕着扫描行为来根据实际需要实现了不同的扫描器或适配器（以下简称为处理器）。每个处理器中都与接口或注解相联系。表 4.2 是 core 模块中实现的注解相关处理器列表，表 4.3 是 core 模块中实现的接口相关处理器列表。本节会详细讲述这些处理器的处理流程。

表 4.3 注解相关处理器

注解	处理器	功能描述
@IocMain、@IocScans、@IocScan	BaseClassesScanner	获取需要扫描的范围。用于扫描范围的初始化
@Component、@Repository、@Service、@Controller	TypeComponentRegistrar	标注在需要注册进容器的类上
@Bean	MethodComponentRegistrar	标注在被容器管理的类的方法上

表 4.4 接口相关处理器

接口	处理器	功能描述
ScanAction	ScanActionScanner	全局扫描来注册扫描行为
BeanPostProcessor	PostProcessorScanner	根据配置文件注入属性

在 3.3 节中曾经描述到，ScanAction 的执行是有顺序的，以上所提到的处理器的执行顺序如图 4.5 所示。

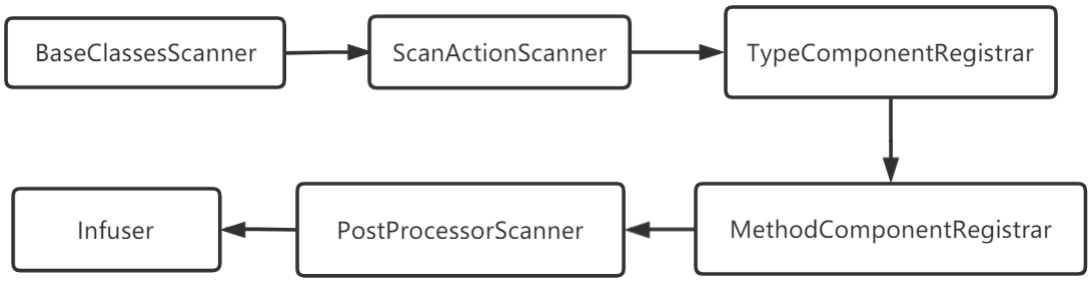

图 4.5 扫描行为执行顺序

4.3.1 扫描范围初始化

BaseClassesScanner（扫描范围扫描器）是以启动类所在的包开始扫描，根据所有类上的@IocMain，@IocScan、嵌套有@IocScan 注解的注解的值来获取扫描范围。@IocScan 注解属性如表 4.4，其中包含的@ComponentFilter 注解如表 4.5。

表 4.5 @IocScan 注解属性表

属性名	类型	描述
basePackage	String[]	根据包名扫描
basePackageClasses	Class<?>[]	根据某个类下的包扫描
useDefaultFilters	Boolean	是否使用默认过滤器
includeFilters	ComponentFilter[]	自定义过滤器，扫描包含 ComponentFilter 条件的包
excludeFilters	ComponentFilter[]	自定义过滤器，扫描不包含 ComponentFilter 条件的包

表 4.6 @ComponentFilter 注解属性表

属性名	类型	描述
type	FilterType	FilterType 是过滤类型枚举，可选择按注解类型过滤、按类过滤、按自定义规则过滤
classes	Class<?>[]	根据 type 选择的过滤方式来提供注解类型、普通类类型、实现了自定义规则的类

在 BaseClassesScanner 中会对类进行扫描，判断是否包含@IocMain、直接或间接的包含@IocScan。对@IocMain 会使用到默认值来构建 BasePackageInfo（扫描包基础信息），对@IocScan 会解析字段属性值来构建 BasePackageInfo，最后构建的 BasePackageInfo 都会被注册到 BasePackageContainer。

BasePackageContainer 有两个作用：一、为核心模块往后的处理提供需要扫描的目录。二、提供给 BaseScanFilter（实现了 ScanFilter，是核心模块使用的扫描包默认过滤器）使用，在涉及到对象的注册的 TypeComponentRegistrar、MethodComponentRegistrar 处理器与涉及对象注入的 Infuser 处理器都是使用 BaseFilter 作为过滤器的。

其中@IocScan 字段中包含着有@ComponentFilter 注解数组，该注解数组会在解析@IocScan 时被转换成两种 FilterMeta（扫描器过滤规则），其中过滤规则分为放行规则和拦截规则，BaseScanFilter 会根据@IocScan 得出是否使用默认过滤规则，如果使用默认规则则直接判断类上是否有包含@Component，否则就要通过放行规则，拦截规则来对扫描的包进行判断放行（ScanFilter 的 isAgree 方法）。判断放行其实就是在图 3.3 中提到的 ScanAction.ScanFilter 中被执行的。其中 FilterMeta 和 BasePackageInfo 的属性如下表 4.6 和表 4.7。

表 4.7 FilterMeta 属性

属性名	类型	描述
type	FilterType	FilterType 是过滤类型枚举，可选择按注解类型过滤、按类过滤、按自定义规则过滤
classes	Class<?>[]	根据 type 选择的过滤方式来提供注解类型、普通类类型、实现了自定义规则的类

表 4.8 BasePackageInfo 属性

属性名	类型	描述
useDefaultFilters	boolean	是否使用默认过滤规则
includeFiltersInfo	FilterMeta[]	放行规则
excludeFiltersInfo	FilterMeta[]	拦截规则
annotation	Annotation	从哪个注解解析 BasePackageInfo
annotationValues	Map<String, Object>	以 annotation 的字段名作为 key，属性值作为 value 的 map

最后 BasePackageScanner 扫描器执行完毕后得到的范围是：@IocMain 注解下的类所在的包，解析@IocScan 与嵌套有@IocScan 注解的注解经过 BaseScanFilter 过滤后得到的范围相加。最后得出的 BasePackageScanner 的处理流程如图 4.6。

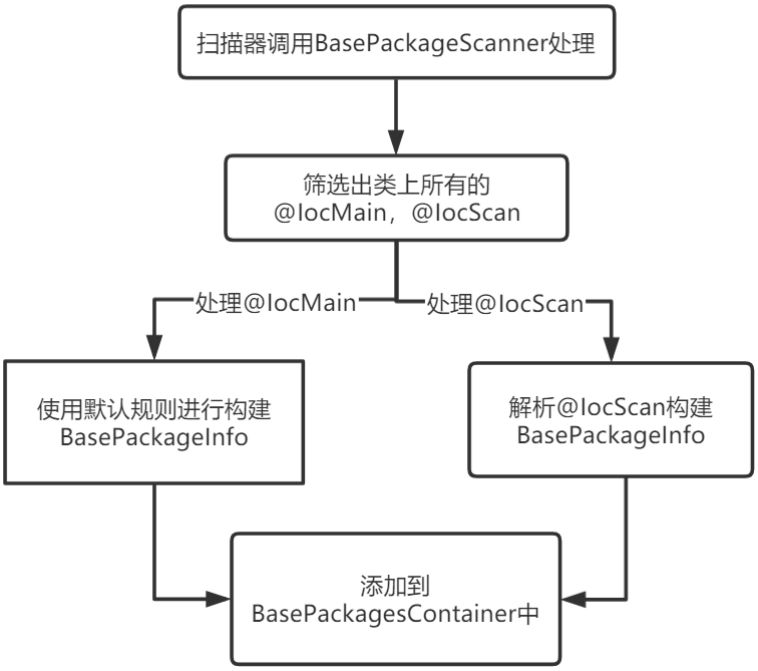

图 4.6 BasePackageScanner 的处理流程

在得到扫描基础路径后，涉及到对象的注册与注入的注册是要经过 BaseScanFilter 来对处理进行过滤的，其中又涉及到三种过滤方式：

1. 判断类的类型来进行过滤。
2. 判断类上是否有某些注解进行过滤。

3. 自定义过滤，根据指定类的 match 方法的 boolean 返回值来过滤。

BaseScanFilter 的处理流程如图 4.7 所示，携带 ClassInfo 执行该扫描行为，判断类所在的包是否注册到 BasePackageContainer 中，然后获取这个注册包的信息 Ba 色 PackageInfo，如果使用默认规则，判断是否包含@Component 来放行，若不使用默认规则，则根据其设定的三种过滤规则的其中一种进行过滤拦截。

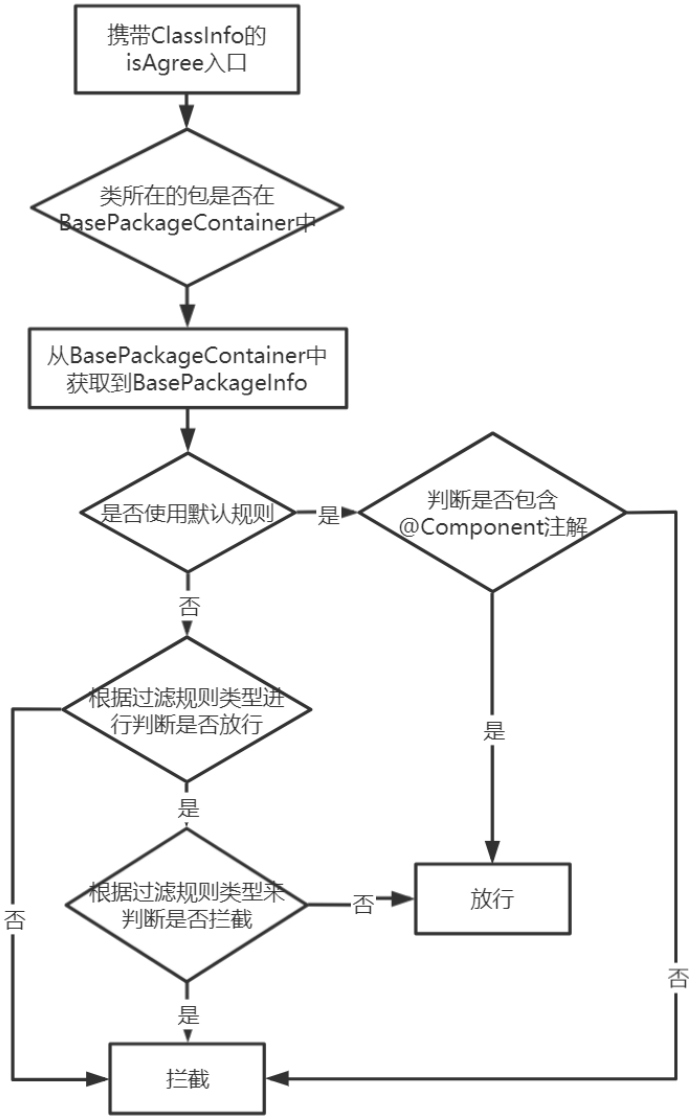

图 4.7 BaseScanFilter 的处理流程

如下面代码中，在@EnableAop 注解包含@IocScan 的情况下，把@EnableAop 和@IocScan 注在 App 类上，在没有配置任何过滤规则，经过 BaseClaseeseScanner 的处理后获取到的扫描范围是：org.jdragon.springz.aop.core 、org.jdragon.springz.test 这两个包。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
```

```
@IocScan(basePackage = "org.jdragon.springz.aop.core")
public @interface EnableAop {

}
@EnableAop
@IocScan(basePackage = {"org.jdragon.springz.test"})
public class App {
    public static void main(String[] args) {
        IocContext.run(App.class);
    }
}
```

4.3.2 全局加载扫描行为

全局加载扫描行为分为三步：一、对核心模块中除了 ScanActionScanner 的已知扫描行为进行注册到 ScanActionContainer 中。二、通过 ScanActionScanner 来对除核心模块中的扩展扫描行为注册，这一步是扩展功能实现基础。三、对以上两步注册的扫描行为根据 order 从低到高进行排序，这个排序决定了扫描行为的执行顺序。核心模块中的已知扫描行为 order 值如 4.8 表所示。

表 4.9 核心模块中的扫描行为 order 值

扫描行为	order 值
ScanActionScanner	强制最先执行
TypeComponentRegistrar	-99
MethodComponentRegistrar	-90
PostProcessorScanner	-80
Infuser	100

整个全局加载扫描行为的过程如图 4.8 所示，在 ScanActionScanner 中进行判断类是否有实现 ScanAction 接口，如果实现了就将其进行无参构造后注册到 ScanActionContainer 中。

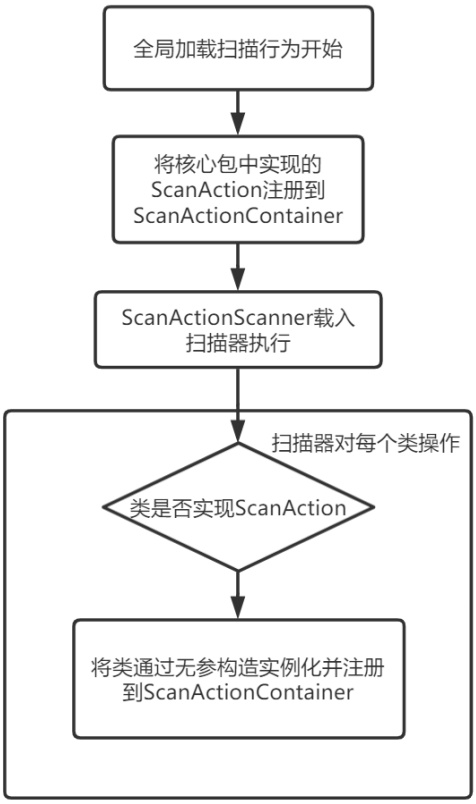

图 4.8 全局加载扫描流程

4.3.3 类注册扫描行为适配器

TypeComponentRegistrar（类注册扫描行为适配器）是继承了注册器的扫描行为，角色为核心模块的扫描适配器。是用于将包扫描得到的 ClassInfo 来提取类注册组件的注册信息封装成 BeanInfo，并为其构造一个别名注册到 BeanContainer 中。。

整个类注册扫描行为处理流程如图 4.9 所示，通过判断类上是否包含@Component，如果符合条件，则创建一个该类的实例对象。获取 value 属性作为 key 来注册到容器中，若 value 为空，则会使用类名作为 key 来注册。在注册前会尝试获取@Scope 来判断该对象注入时的注入策略，支持单例注入和原型注入，当没有获取到@Scope 时，默认使用的是单例注入策略。

@Import 注解可以设置一个 Class 数组，如果类上有@Import 注解，会获取到这个 Class 数组，对这些类进行使用默认选项进行注册（使用类名作为 key，采用单例注入策略）。

如果类上还有@Properties 注解，那么在注册后还会通过注入的一些通过配置文件读取的属性值，这一部分可以翻阅本文的 4.1 节。

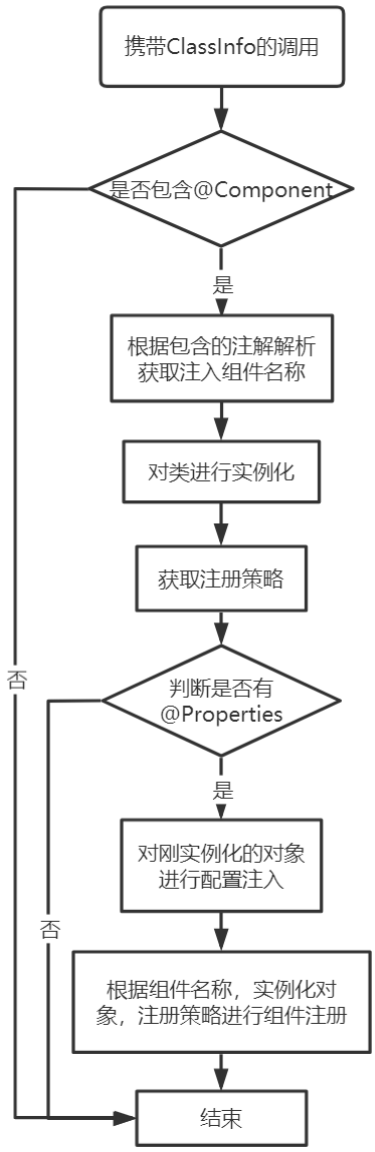

图 4.9 类注册扫描行为流程

4.3.4 方法注册扫描行为适配器

MethodComponentRegistrar（方法注册扫描行为适配器）是继承了注册器的扫描行为，角色为核心模块的扫描适配器。用于对@Bean 注解标注下的方法进行处理，在默认情况下，将方法名作为 key，返回值作为 value 注册到容器中。其中如果该方法有参数的话，会根据参数的类名尝试从已注册的组件中获取，并对获取的组件提前执行注入和后置处理等操作后将其注入到参数中来执行该方法。这里注入到参数也支持使用@Resourse 注解（在 4.6 节中会提到）来注入对象。

使@Bean 注解生效的前提是，该类必须是已经被容器管理的对象，即这个类上要包含@Component。

在容器进行参数注入的时候，因为提前了注入的时机，所以有可能会出现的一种情况：使用组件比注册组件的时机更早了，导致无法获取到需要用到的组件，但是这个组件的确会在整个生命周期中被注册。框架中使用到了等待唤醒的注册列表来解决这个问题。每次尝试从已注册的组件获取失败时，会将收集失败的组件名加入到 needBeanName（等待依赖对象名链表）中，并收集一些处理信息封装成 WaitBeanInfo 保存在 BeanContainer 中。WaitBeanInfo 的属性如表 4.10。

表 4.10 WaitBeanInfo 属性表

属性名	类型	描述
waitBean	Object	被容器管理的对象，即执行方法的对象
clazz	Class<?>	正在等待唤醒注册的类
constructMethod	Method	触发等待唤醒时执行的方法
scope	String	注册时的注入策略，单例或原型
paramsNameList	List<String>	constructMethod 方法需要的参数名字列表
needBeanName	List<String>	需要的组件名称，通过该名称到容器中获取已注册的组件
beanName	String	注册唤醒组件使用的名称

需要改造注册器，在注册后添加一个调用钩子，在每次进行组件注册后，需要将注册的组件名与已存在需要唤醒属性列表中的 needBeanName 进行比对删除，当某个等待唤醒组件的 needBeanName 全部删除完毕后，就代表这个注件的依赖组件已经全部注册完毕，这时候通过容器获取到这些依赖对象，构建参数数组来反射调用 constructMethod 构建出需要注册的对象，成功唤醒注册。

方法注册扫描行为的整体过程如图 4.10 所示，携带 ClassInfo 执行该扫描行为，判断该类是否包含@Component 注解，是否被注册过（可能存在某些原因注册失败，就算有@Component 注解也不一定是被注册过的），被注册过的话将遍历该类的所有方法，判断方法上是否有@Bean 注解。如果有，则再进行方法参数加入到一个存放构建依赖对象名的链表中，再进行遍历，通过该参数的类型名或@Resouce 注解的值来获取 key 值，通过该 key 值来尝试从注册容器中获取已注册的对象信息，如果能获取到 key 对应的对象，则将该参数从链表中删除，否则则将其加入到 needBeanName 链表中。最后根据这个 needBeanName 是否为空来判断这次注册是否需要等待依赖唤醒，如果需要等待唤醒，则构建等待注册信息 WaitBeanInfo 保存到容器 BeanContainer 中。

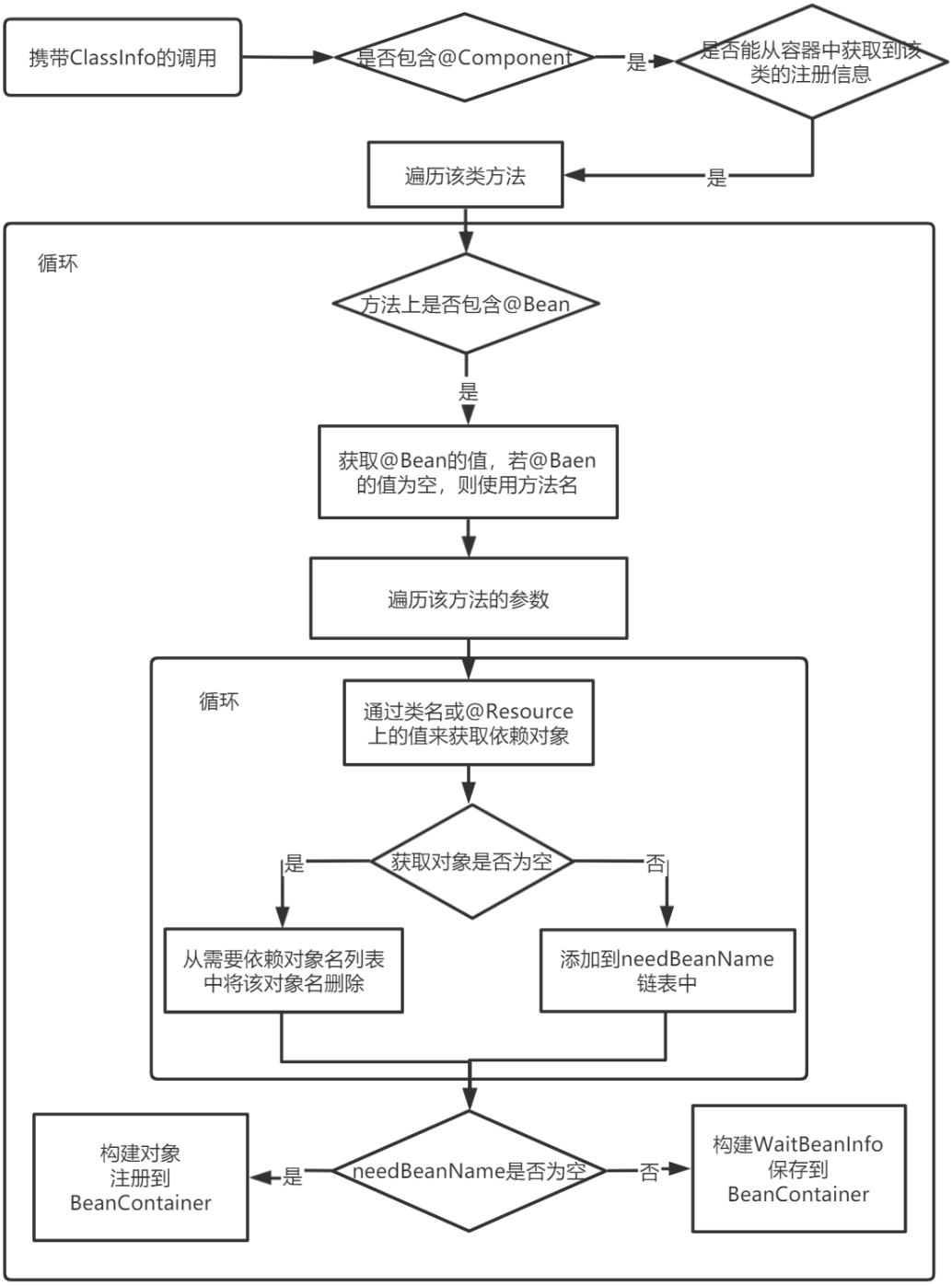

图 4.10 方法注册扫描行为处理过程

其中为注册器添加的唤醒注册处理如图 4.11 所示，在任意一个对象进行注册完毕之后，会去遍历所有的等待唤醒列表，再判断这次注册的对象是否在每一个等待唤醒对象的 needBeanName 列表中，如果存在，则将该注册对象的名字从 needBeanName 列表中移除。然后判断唤醒对象的 needBeanName 是否都被移除了，如果是则将该对象唤醒注册。

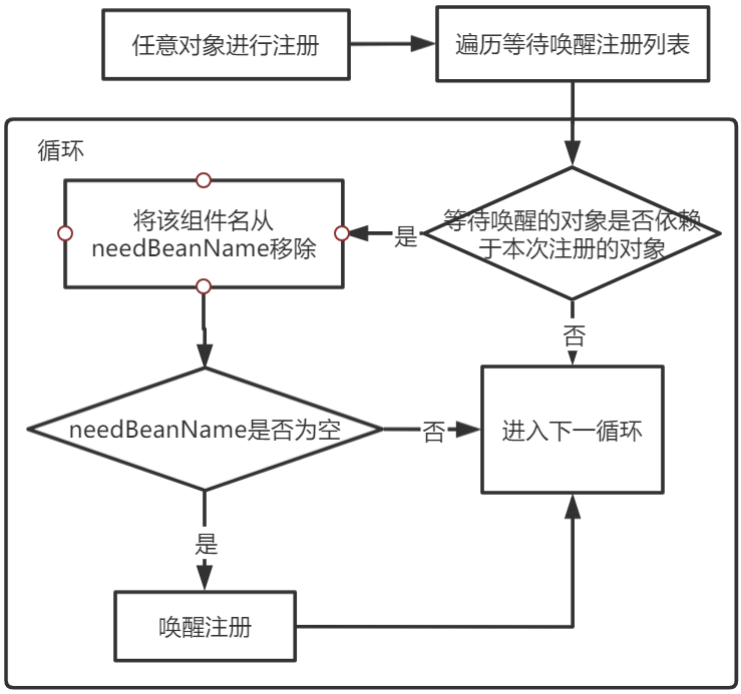

图 4.11 唤醒注册过程

4.3.5 加载后置处理器扫描行为

BeanPostProcessor（后置处理器）是在对象的依赖全部被注入完成之后与执行对象初始化之前之间对对象进行处理的。在这个阶段只是加载这些处理器，并没有真正的执行，这些后置处理器是在 PostExecutor（后置处理器执行器）中执行的，PostExecutor 在对象注入器中被调用，所以实际上，后置处理器就是在对象注入时被执行的。

在扫描时会判断类是否实现了 BeanPostProcessor 接口，若实现了，就把他注册到 PostProcessorContainer 中。具体执行会在对象注入器中进行描述。

4.4 对象注入器

Infuser（对象注入器）用于对已经注册到容器中的对象的进行依赖注入。这一步处于整个程序启动完成前的最后一步，在所有扫描行为执行完成之后。其中包含了：依赖注入、注入后置处理器执行、执行@PostConstruct 标注的初始化方法、跟踪被注入的对象（包括所有原型注入，将用于程序的销毁清理方法执行）。

其中依赖注入的部分较为复杂，要解决两个问题：

1. 注入时要保证单例与原型注入的特性。单例注册时，注入在任何对象中都应该是同一引用，原型注入需要对注册对象进行克隆，但原型中的单

例依赖的引用依旧要保持相同。

2. 对象与对象之间的循环依赖。如何在 A 依赖 B，B 也依赖 A 的情况下，不论 A 与 B 的策略是什么，他们最后注入结果都是能够成环的。不允许出现以下情况：在 A 为原型注册策略下，A 注入 B，而 B 中注入的是 A 的克隆对象。

最后采用的解决方法：将已注册进容器的组件，根据依赖关系构成有向图，通过深度遍历节点的方式，对节点从最底层依赖往上注入直至最高层。并且建立一个先进先出栈来控制注入并保存当前注入对象的上层对象。

举例来展现系统的注入逻辑，现有 A、B、C、D、E、F、H，七个类注册到容器中，F 设置为原型注入，其他皆为单例注入。其中的依赖关系构建成图后如图 4.12 所示。

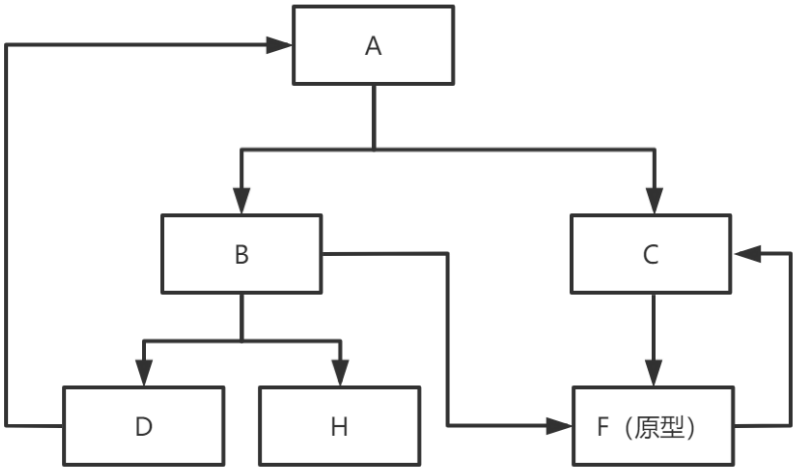

图 4.12 七个类的依赖关系图，箭头方向为依赖方向

先将底层 A 入栈，再去判断 A 有没有依赖需要注入，依赖 B，则再将 B 入栈。进行该图的深度遍历，在一直入栈都有依赖对象时，以此类推。如图 4.13 所示。

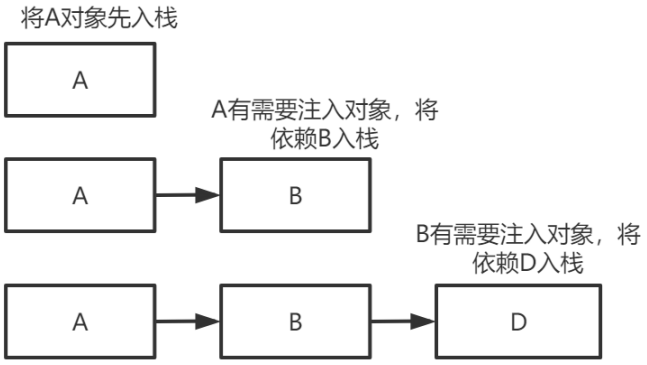

图 4.13 依赖入栈，实心箭头方向指向栈顶

因为栈是用来保存上层对象，在接下来会在为 D 注入依赖 A 时，在栈中发

现有 A 的存在，所以说明 A 的向下依赖存在 D，并且 D 也依赖 A，图中存在环，则发生了循环依赖，循环依赖的处理如图 4.14。

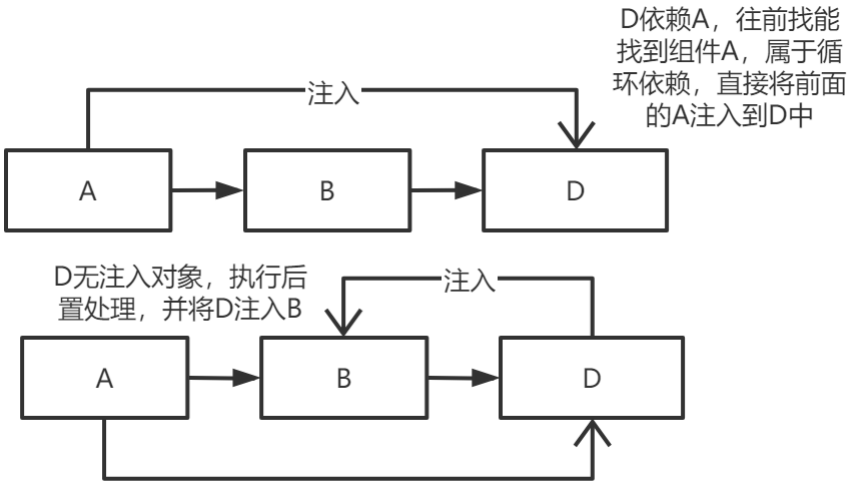

图 4.14 循环依赖的解决，空心箭头代表着正在注入或已经注入

H 中没有依赖可注入了，直接将他们注入到栈的上一个对象中。如图 4.15。

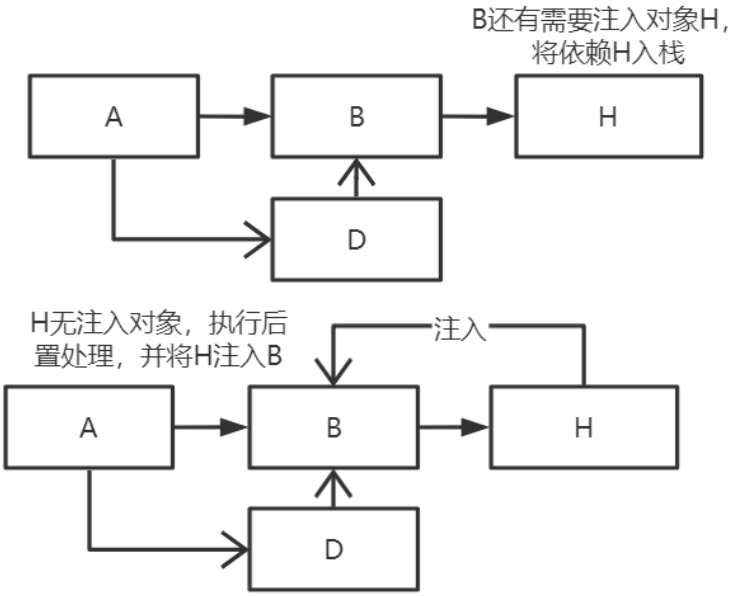

图 4.15 将 H 注入到 B 中

当遇到需要进行原型注入的 F，与前面相似，其中的区别仅仅是需要先将 F 克隆出 F（1）对象，再将 F（1）对象入栈，然后对栈中 F（1）对象进行依赖注入。在本次例子中是同时存在 F 与 C 的循环依赖的，处理如下图 4.16 所示，根据图文理解，不再重复描述。

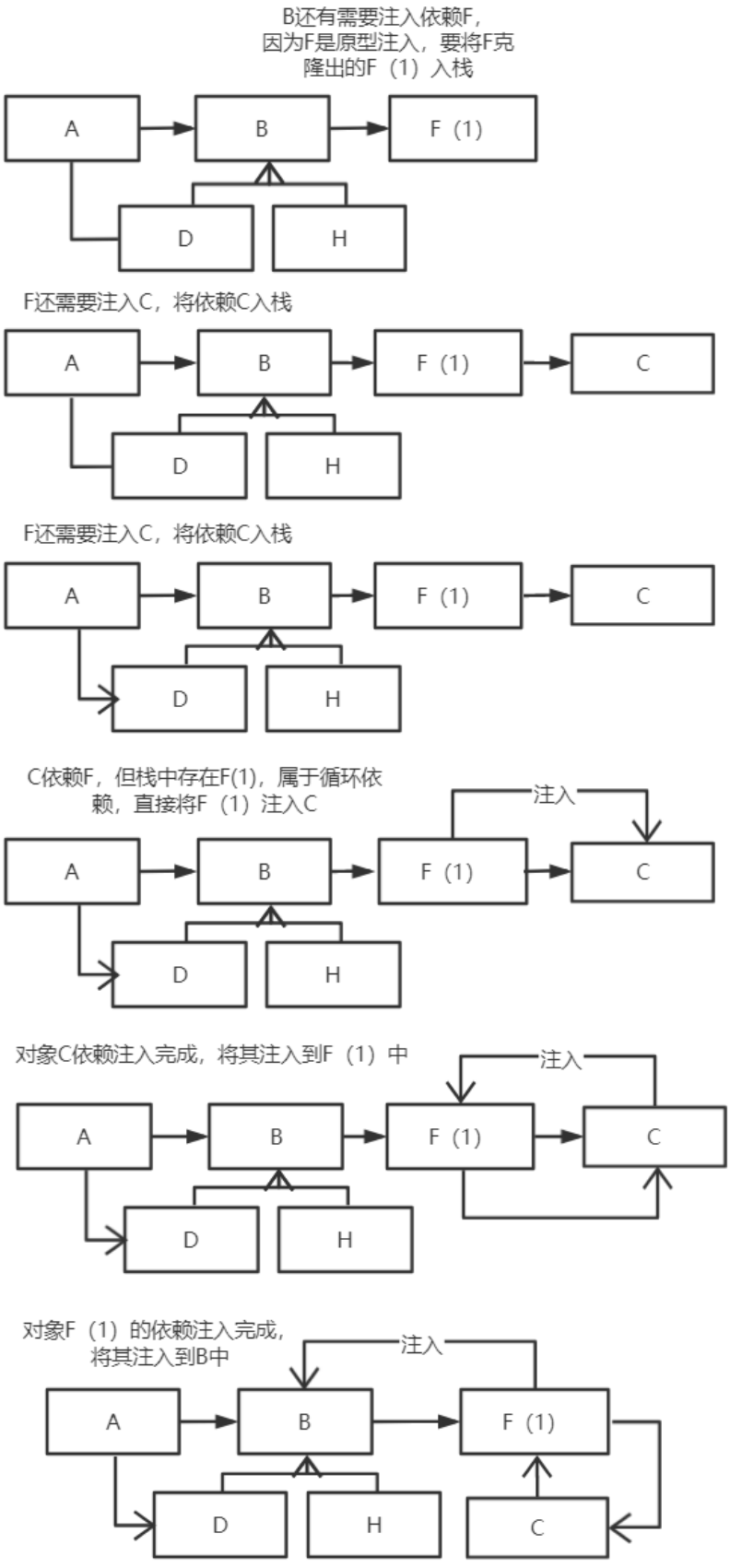

图 4.16 对 F 注入到 B 时的处理

B 的所有依赖已经注入完成了，将 B 注入到 A 中，如图 4.17。

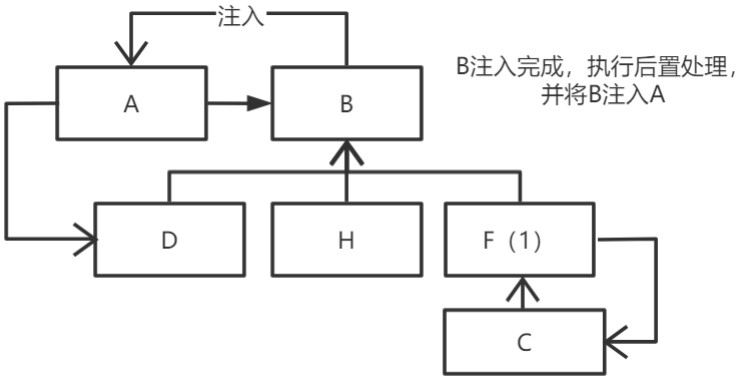

图 4.17 将 B 注入到 A

上面已经用图例来说明了在注入时所会遇到的所有情况，在注入 C 时更为简单，这里不再画图描述，最后注入结果就是如下图 4.18 所示。

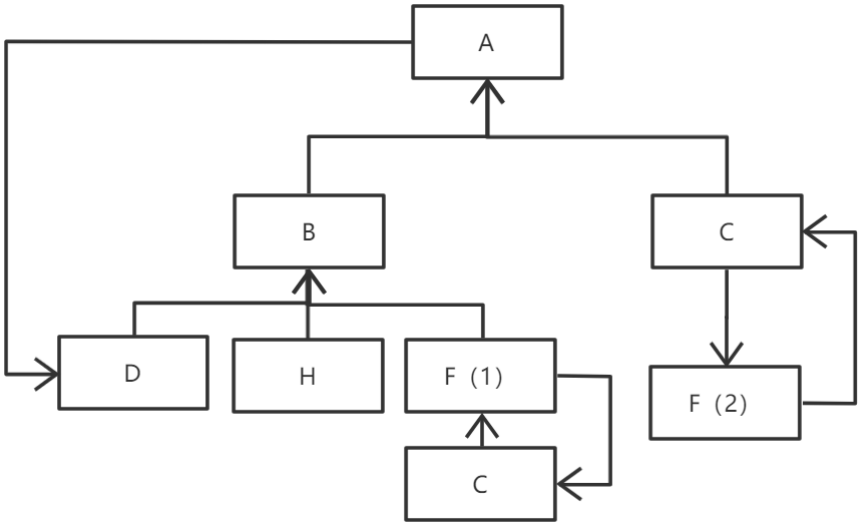

图 4.18 所有依赖注入完成后的对象引用关系，箭头方向为被引用方向

4.5 后置处理执行器

在每次对对象的依赖注入完成后，都会将该对象给后置处理执行器进行执行后置处理。后置处理执行器的执行分为三步：一、对在 4.3.5 中加载的后置处理器进行执行。二、执行对象的@PostConstruct 标注的对象初始化方法。三、将对象的@Destroy 标注的方法销毁清理方法注册到 DestroyHook（销毁钩子）中。

4.6 程序结束销毁钩子

在框架启动的第一步其实就是注册这个销毁钩子，这个钩子主要是为了调用我们在 4.5 中注册到钩子进来的销毁方法来进行系统的清理动作。

4.7 本章小结

本章进行了框架核心的设计：如何加载配置文件、各个核心扫描行为的设计、实现对象注入、初始化与销毁方法调用设计。

第 5 章 框架扩展设计

5.1 扩展面向切面（aop 模块）

springz-aop 模块为框架简单的提供面向切面编程的能力，在引入 spring-core 模块的基础上，使用 jdk 动态代理与 cglib 动态代理来使框架支持切面编程，切面范围的筛选支持类与方法级别。在该模块中新添加的注解如表 5.1。

表 5.1 springz-aop 模块中的注解

注解	描述
@EnableAop	标注在启动类上，开启 aop 功能
@Aop	标注在类上，用作标注该类为切面处理类
@Pointcut	在@Aop 中使用，用作声明该切面处理类的切点
@Around	标注在拥有@Aop 的类下的方法上。用作标注作用于该切面处理类下切点环绕增强方法
@Before	标注在拥有@Aop 的类下的方法上。用作标注作用于该切面处理类下切点前置增强方法
@After	标注在拥有@Aop 的类下的方法上，用作标注作用于该切面处理类下切点后置增强方法
@Throw	标注在拥有@Aop 的类下的方法上，用作标注作用于该切面处理类下切点出现未捕获异常调用方法

该功能实现的核心在于，自定义一个扫描行为 AopScanner（Aop 扫描行为），在扫描到的类上有@Aop 注解，会将注解信息与该类下的方法分类组装成 PointCutInfo（切面信息）并注册到 AopContext（切面上下文）中，切面信息包括切面类的注册对象、@Around 注解下的环绕增强方法、@Before 注解下的前置增强方法、@After 注解下的后置增强方法、@Throw 注解下的异常处理方法，切面执行优先级 order（与扫描行为 ScanAction 的 order 不是同一个 order）等。在注册到切面上下文后，Aop 扫描行为就结束了，并没有做其他的处理。

再定义一个 Aop 代理工厂后置处理器（AopProxyPostProcessorFactory）获取 AopContext 中已注册的切面，对切面解析，将符合代理的包范围的对象进行动态代理。所以该扫描行为必须在注册对象之后，加载后置处理器之前。

在进行依赖注入前，后置处理器 AopProxyPostProcessorFactory 会根据已注册的切面信息来判断该依赖是否处在已注册的切面范围，决定是否对该依赖生成代理对象注入。判断分为两级，一级包类通配和二级方法通配。举例说明：一级

包类通配“org.example.AopService*”就匹配 org.example 包下以 AopService 开头类。二级方法通配“get*”就匹配根据一级包类通配下符合的类的以 get 开头的所有方法。

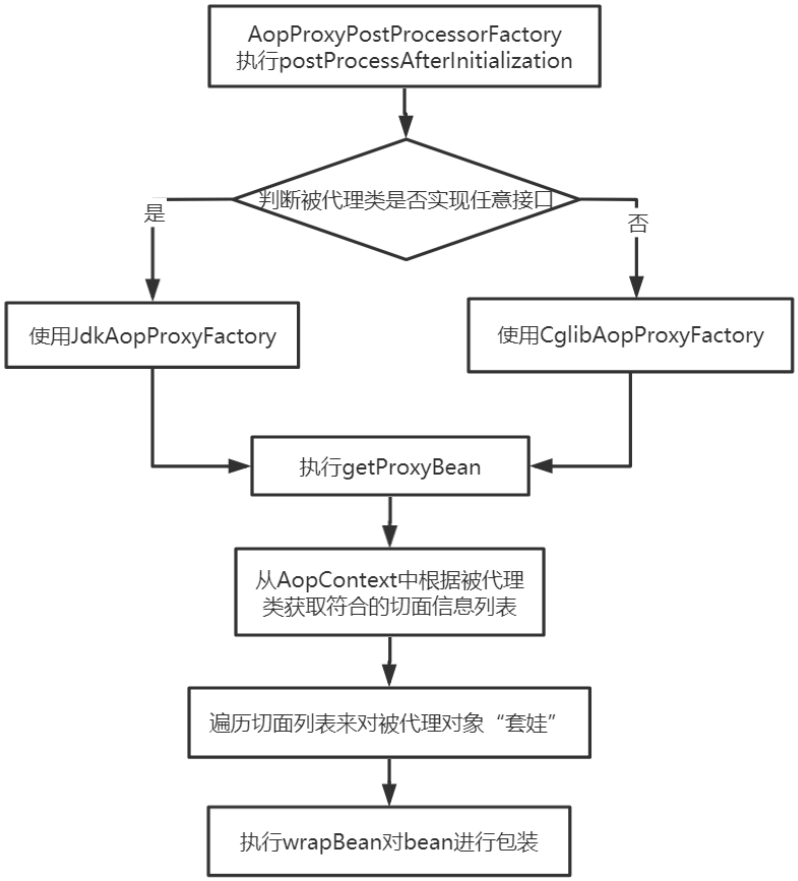

图 5.1 对被代理对象的包装流程

其中 AopProxyPostProcessorFactory 中涉及到了抽象工厂模式的使用，如图 5.1 所示，会根据传进来的 bean 判断其是否是接口代理还是继承代理来选择使用 jdk 动态代理工厂还是使用 cglib 动态代理工厂来生成代理对象。在抽象工厂内，会由 AopContext 根据该对象来获取切面信息，然后根据这些切面信息来执行抽象方法 wrapBean 来对被代理对象“套娃”。这个“套娃”的最终的行为取决于你一开始使用的是 jdk 还是 cglib 工厂。类图如下图 5.2 所示

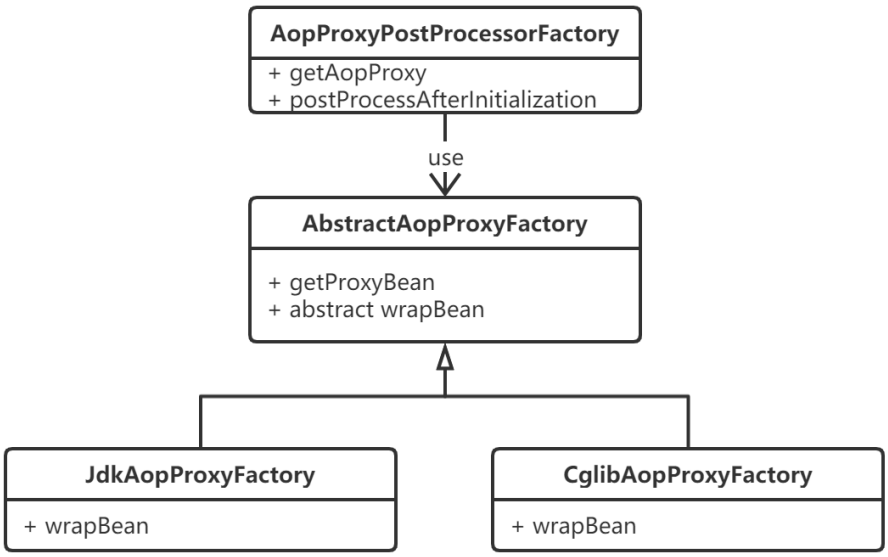

图 5.2 代理工厂类图

最终会根据切面执行优先级 order 的值从小到大排序，然后将被代理对象用切面包装起来再注入，执行时会先执行外层切面，然后由上层切面执行下一层切面，直到调用到最内层的对象方法，如下图 5.3 所示。

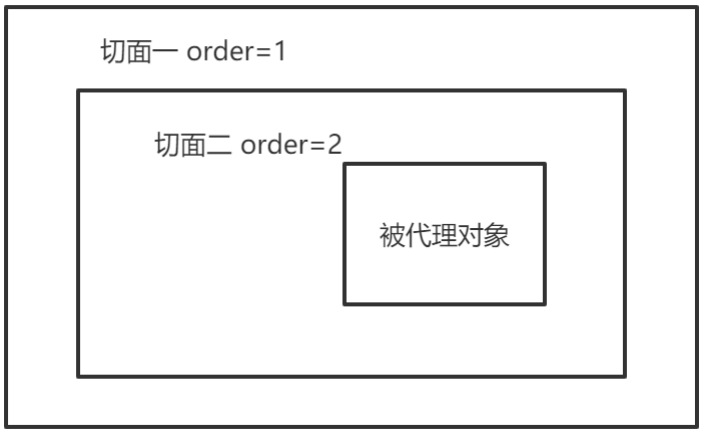

图 5.3 对被代理对象的包装后

5.2 扩展 http 服务器（web 模块）

springz-web 模块为框架简单的提供 http 服务的功能，在 spring-core 模块的基础上，使用 netty 作为提供 http 服务通信的底座，添加路径映射方法路由的概念，来为请求路径与处理方法添加映射关系，在对服务器请求某个路径时，可以通过映射表来找到处理这个请求的方法，将处理结果返回。

在该模块中新添加的注解如表 5.2。

表 5.2 springz-web 模块中的注解

注解	描述
@EnableWeb	标注在启动类上，开启 web 服务功能
@WebRoute	标注在类上，用作标注该类注册路由映射。该类也存在@Get、@Post 上，意味着他们也是路由。有两个属性：value 代表着请求路径，method 代表是请求方法
@WebParam	标注在参数类型注解上，代表这个注解是用在路由映射方法的参数上的
@Get	标注在方法上，代表这个为 Get 请求映射方法，该注解的值拼上类上@WebRoute 的值就是请求路径，注解标注的方法就是请求处理方法。与@WebRoute(method = RequestMethod.GET)等价
@Post	标注在方法上，代表这个为 Post 请求映射方法，该注解的值拼上类上@WebRoute 的值就是请求路径，注解标注的方法就是请求处理方法。与@WebRoute(method = RequestMethod.POST)等价
@RequestParam	参数类型注解，标注在路由方法的参数上，表示该参数是的 param 参数
@RequestBody	参数类型注解，标注在路由方法的参数上，表示该参数是请求的 body
@RequestHeader	参数类型注解，标注在路由方法的参数上，表示该参数是从 header 取值
@PathVariable	参数类型注解，标注在路由方法的参数上，表示该参数是 restful 风格的路径参数

模块中涉及到注解的设计时，考虑到注解包含，比如说@Get、@Post 注解是包含着@WebRoute 注解，这表示这两个注解属于路由注解。@RequestParam、@RequuestBody、@RequestHeader、@PathVariable 是包含@WebParam，这表示这四个注解是属于参数注解。

通过路由注册扫描注册器（RouteRegistrar）来判断类上是否有@WebRoute 注解，如果有该注解，代表是需要该类是路由类，需要被注册到 RouteMethodMapper（路由方法映射）中，再判断该类中的方法是否包含@WebRoute，最后的映射到方法的路径等于类上的@WebRoute 的值拼接上方法上@WebRoute 的值。例如以下代码中的 get 请求“/view/test”就调用方法 test 来处理。

```
@WebRoute("/view")
public class ViewController {
    @Get("/test")
    public void test() {
    }
}
```

与 spring-aop 模块只实现扫描行为来扩展的扩展方式不同，在该模块的内部处理已经使用到@Component 注解将相关处理类注册与注入了，这已经涉及到了 spring-core 模块的基本功能的使用了。关键在于 HttpServer 的注入完毕后的自动启动。

在 HttpServer 上有@ PostConstruct 注解，在 HttpServer 注入依赖完毕后，会自动启用新的线程去将 HttpServer 开启。其中核心是 HttpServerHandler，负责接受请求信息。

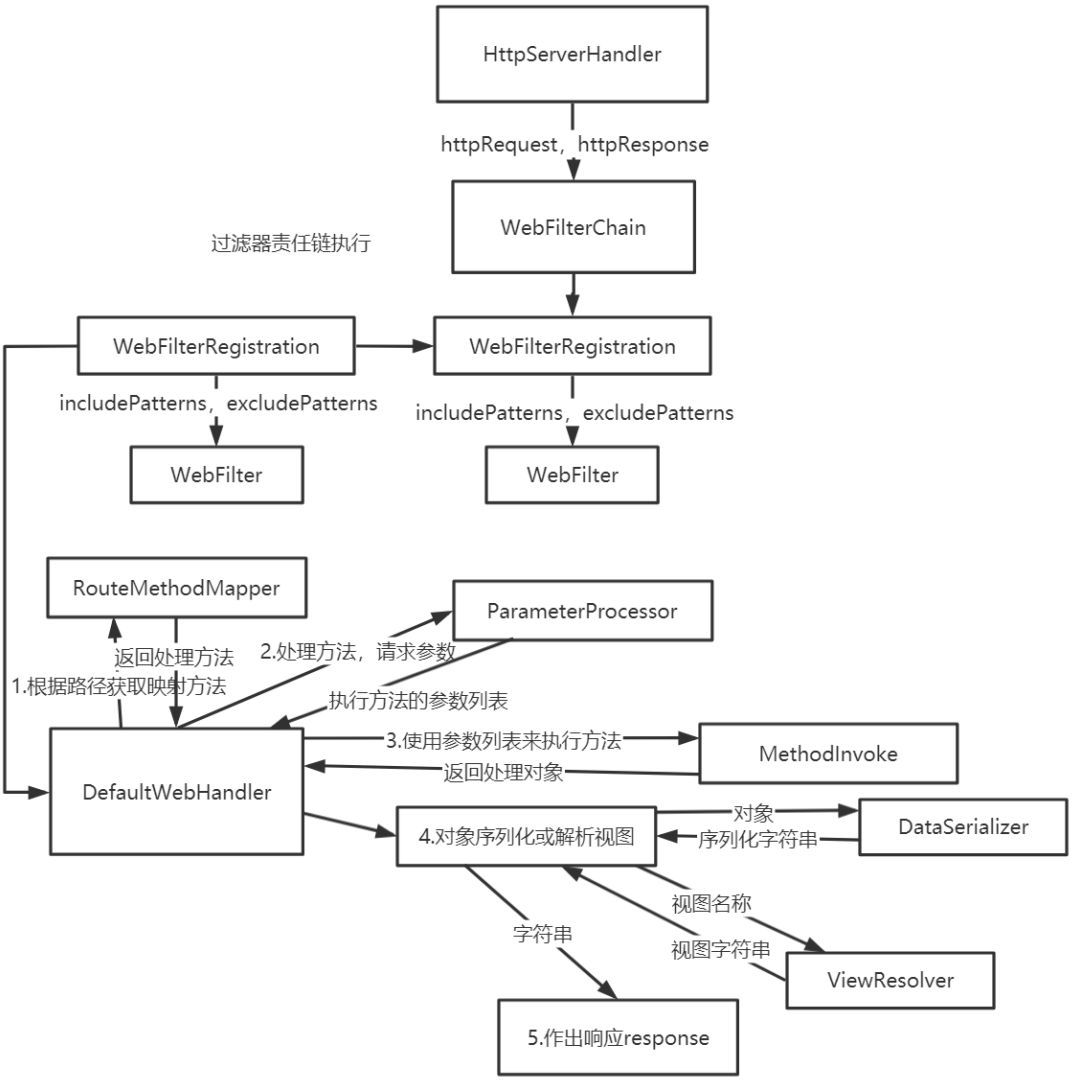

图 5.4 处理请求流程

其中处理流程如图 5.4 所示，在 HttpServerHandler 中主要任务是将 netty 接受到的请求信息封装成HttpRequest与HttpReponse，然后将其交给WebFilterChain（请求过滤链）进行每一个 WebFilter 的过滤处理。在过滤通过后的请求，会交给 DefaultWebHandler（继承了 WebHandler 的默认处理器）进行具体的处理。HttpRequest 的属性如表 5.3 所示。

表 5.3 HttpRequest 属性表

属性名	类型	描述
uri	String	请求 uri
path	String	请求路径
method	RequestMethod	请求方法
matchingPath	String	匹配路由的值
localAddress	InetSocketAddress	本地地址
remoteAddress	InetSocketAddress	客户端请求地址
requestParams	RequestParams	请求参数
resourceMappingRegistrations	List	静态资源映射列表

在 DefaultWebHandler 中，执行以下步骤。

1. 通过 HttpRequest 获取这次请求的路径，通过路径从 RouteMethodMapper（路由方法映射）中获取到对应的处理方法。
2. 构建一个 WebContent 上下文对象，通过第 1 步得到的处理方法，获取方法上的参数，解析参数上是否需要注入 WebContent（用于获取请求参数或设置返回视图等）或是否包含有四种参数注解的其中一种，再通过注解的种类和注解的值从 HttpRequest 中取出值，并按顺序组成数组并返回。
3. 使用第 2 步获取到的数组作为参数，如果参数调用第 1 步得到的处理方法，保存处理结果。
4. 经过第 3 步的执行，如果 WebContent 上下文中有保存视图名字，则将处理结果交给 ViewResolver（视图解析器，默认使用 freemarker 引擎解析）处理，否则交给 DataSerializer（对象序列化器，默认为序列化为 json）进行序列化处理。最终两种处理都会生成一个字符串，
5. 将这个字符串通过 HttpResponse 发送回浏览器进行展示。但所返回的 Content-Type 响应头是不一样的，经过视图解析器返回的 Content-Type 是 text/html，经过对象序列化器返回的 Content-Type 是 application/json。

存在另外的情况，在请求时，通过 RouteMethodMapper 找不到对应的方法进行处理，就会去寻找静态资源，静态资源映射目录是可以通过 WebConfiguration 进行配置的。如果在静态资源映射目录中也无法找到请求路径对应的文件，那就会返回 404 错误。

启用了 Web 功能之后，必须要有一个继承了 WebConfiguration 的类被注册到容器中作为注入到 HttpServer 的配置类。以下代码为必须需要的配置类。

```
@Configuration
```

```
public class WebConfig extends WebConfiguration {


}
```

以下代码为添加 testFilter 过滤器，作为范围为除了/static/目录下的所有请求路径。

```
protected void addWebFilters(WebFilterRegistry webFilterRegistry) {
    webFilterRegistry.addWebFilter(testFilter)
                        .addPathPatterns("/**")
                        .addExcludePatterns("/static/**");
}
```

下面代码为添加两个资源映射：任意/static/请求都会映射到/var/java/springz/文件目录下。任意/classpath/请求都会被映射到项目资源文件的/static/下。

```
protected void addResourceMapping(ResourceMappingRegistry registry) {
    registry
       .addResourceMapping("/static/**")
       .setResourceLocation("/var/java/springz/");
    registry
       .addResourceMapping("/classpath/**")
       .setResourceLocation("classpath:/static/");
}
```

下面代码为配置自定义对象序列化器和视图解析器。

```
protected DataSerializer setDataSerializer() {
    return new FastJsonSerializer();
}
protected ViewResolver setViewResolver() {
    return new FreemarkerViewResolver();
}
```

5.3 扩展请求外部数据方法（feign 模块）

springz-feign 为框架提供能调用外部系统接口的接口方法定义。在 web 模块的基础上添加了一个新的注解@ZFeign 来定义一个外部系统接口类。该注解属性

如表 5.4 所示。

表 5.4 @ZFeign 注解属性表

属性名	类型	描述
baseUrl	String	请求地址
basePath	String	请求路径
depth	String[]	对返回结果解析层次
fallback	String	处理请求异常的统一处理方法

该模块逻辑比较简单，利用 jdk 动态代理的“投辫断流”实现对无实现方法的抽象接口进行代理。虽然为代理，但是只执行代理中的处理，并不对被代理对象的方法调用。正常的动态代理处理一个 UserService 的 save 方法流程如图 5.5 所示。

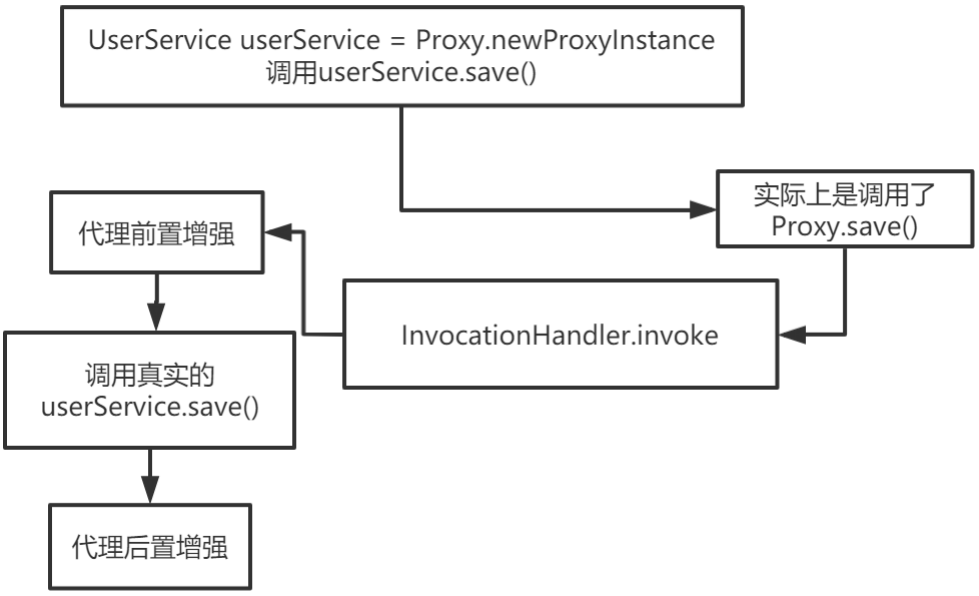

图 5.5 jdk 动态代理的执行流程

“投辫断流”的实质就是撤掉调用真实的 userService.save()这一步，达到了虽然代理了目标，但执行时并没有目标的执行，被代理类“独掌大权”，所有的处理逻辑都处于代理类中。这也就实现了我们对没有实现类的接口类代理。

FeignRegistrar（Feign 接口注册器）扫描类上是否有@ZFeign 注解，将注解上的 baseUrl、basePath 拼接起来作为基础请求路径。将 DynaProxyHttp（动态 http 请求代理）注册到容器中，然后在注入时会对这个被代理的接口类的声明注入这个对象。

这个对象的执行，实际上就是调用 DynaProxyHttp 中的 invoke 方法，在 invoke

的调用中可以获取到调用的方法 method。解析这个 method 对象（接口类下的接口方法）。在 ZFeign 接口方法上还标注了@Post 或@Get 路由注解。最后执行这个方法就等于请求基础请求路径加上路由注解上的值这个网络接口。得到返回结果后根据 depth 属性去解构对象后进行序列化为返回类对象。

```
@ZFeign(baseUrl = "https://cl.tyu.wiki", basePath = "/robot", depth = "result")
public interface ClGroupMap {
    @Post("/query/getGroupMap")
    List<Map<String, String>> getGroupMap();
}
```

举例如上代码，执行 getGroupMap 方法，相当于请求 https://cl.tyu.wiki/robot/query/getGroupMap 这个接口，这个接口返回的结果如下，然后对这个返回结果根据 depth 解构，最后会提取出 result 的值，然后序列化成 List<Map<String,String>>进行返回。

```
{
  "code": 200,
  "message": "获取成功",
  "result": [
    {
      "groupName": "聚贤",
      "groupId": 12941287
    },
    {
      "groupName": "五林",
      "groupId": 26053477
    }
}
```

5.4 本章小结

本章进行了框架的扩展功能设计，在 core 模块的基础上利用扫描行为延展的三大功能模块。

第 6 章 框架测试举例

6.1 core 模块测试举例

在 core 模块中各注解的作用如表 6.1。

表 6.1 core 模块所有注解表

注解	作用
@IocMain	声明一个类为框架启动类。会被注册到容器中
@IocScan	设定需要扫描的包，只有这个包下的类会进行扫描
@Configuration	声明一个类为配置类，这个类中可以使用@Bean 来通过方法来注册组件
@Component、@Repository、@Service、@Controller	声明一个类需要被注册到容器中，这四个注解的作用与用法都一致，只是用于区分作用
@Bean	声明方法的返回值需要注册到容器中
@Scope	与注册类注解一起使用，声明这个注册组件的注入策略是单例还是原型
@Value	声明在字段上，通过表达式的方式从 yml 配置中读取配置项注入到字段中
@Properties	与注册注解一起使用，声明这个类的字段需要从 yml 配置中读取配置注入到对应相同名字的字段中，可以使用@Value 来对名字不对应的字段建立关联
@Inject	根据类型进行对象注入
@Resource	根据字段名进行对象注入
@PostConstruct	标注在方法上，在依赖注入完成后自动调用
@Destroy	标注在方法上，在程序销毁时自动调用

6.1.1 快速启动

在拥有 main 启动方法的 App 类上加上@IocMain 代表这是启动类。在@IocMain 注解下加入@IocScan 来告诉框架要扫描主启动类 App 所在目录的所有包。以下代码就是利用@IocMain 和@IocScan 快速启动例子。

```
@IocMain
```

```
@IocScan
public class App {
	public static void main(String[] args) {
		IocContext.run(App.class);
	}
}
```

6.1.2 基本的注册与注入

以下代码使用@Component 与@Bean 注册的例子，使用@Component 注解在 People 类上，将 People 类使用原型策略注册到容器中。使用@Configuration 和@Bean 通过方法名将 People 类注册到容器中。将 TestBean 类以默认的单例策略注册到容器中，最后使用@Inject 将通过@Compoenet 注册的 people 与使用@Resource 将通过@Bean 注册的 people1 和 people2 注入到 TestBean 中。

```
@Component
@Scope(value = BeanInfo.PROTOTYPE)
public class People {

}

@Configuration
public class TestBeanConfig {
	@Bean
	public People people1() {
		return new People(1,"张三","440000******");
	}
	@Bean
	public People people2() {
		return new People(2, "李四", "440106*****");
	}
}

@Component
public class TestBean {
	@Inject
	private People people;
```

```
    @Resource
    private People people1;
    @Resource
    private People people2;
}
```

6.1.3 配置文件注入

见下面的 application.yml 的配置文件内容，其中代表着有两个 user 的配置项，一个是 zhang-san，一个是 lisi。

```
user:
  zhang-san:
    id: 11111111
    shortId: 22
    longId: 3333333333
    name: 张三
    takeEffect: true
    nickName:
      - 老张
      - 阿三
      - 李四的大哥
    reading:
      - ISBN: 123-1235-2324
        name: 语文书
      - ISBN: 321-1234-1234
        name: 数学书
    loadClass: org.jdragon.springz.test2.testProperty.Book
  lisi:
    id: 4444444
    shortId: 55
    longId: 6666666666
    name: 李四
    takeEffect: true
    nickName:
      - 老李
```

```
    - 阿四
    - 张三的小弟
  reading:
    - ISBN: 234-5234-233
      name: 英语书
    - ISBN: 321-1234-1234
      name: 数学书
  loadClass: org.jdragon.springz.test2.testProperty.Book
```

以下代码通过使用@Properties 标注在类上来注入配置，并使用@Value 来映射 name 与 username 不相同名称的属性，并使用 zhangsan 为 key 注册到容器中。

```
@Component("zhangsan")
@Properties(prefix = "user.zhang-san")
public class User {
    private Integer id;

    private Short shortId;

    private Long longId;

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

    private Boolean takeEffect;

    private List<String> nickName;

    private List<Book> reading;

    private Class<?> loadClass;
}
```

以下代码通过使用@Properties 标注在@Configuration 标注类的方法上注入配置。这时 User 在注册会自动注入配置到字段上，并使用方法名 lisi 为 key 注册到容器中。

```
@Configuration
public class TestPropertyConfig {
    @Bean
    @Properties(prefix = "user.lisi")
    public User lisi() {
        return new User();
    }
}
```

以下代码就是将上面两个配置实体注入到 TestProperty 中使用。

```
@Component
public class TestProperty {
    @Resource
    private User zhangsan;

    @Resource
    private User lisi;
}
```

6.1.4 初始化与销毁方法

以下代码为@PostContruct 和@Destroy 的使用，在 Cycle 的依赖注入完毕后会调用 init 方法，输出“Cycle 的 PostConstruct 方法被初始化调用了”。

在程序结束前会调用 destroy 方法，输出“Cycle 的 Destroy 方法在销毁时被调用了”。

```
@Component
public class Cycle {
    private final Logger logger = LoggerFactory.getLogger(getClass());

    @PostConstruct
    public void init(){
        logger.trace("Cycle 的 PostConstruct 方法被初始化调用了");
    }

    @Destroy
    public void destroy(){
```

```
        logger.trace("Cycle 的 Destroy 方法在销毁时被调用了");
    }
}
```

6.2 扩展模块测试举例

6.2.1 aop 模块测试举例

在 aop 模块中各注解的作用如表 6.2 所示。

表 6.2 aop 模块所有注解表

注解	描述
@EnableAop	标注在启动类上，开启 aop 功能
@Aop	标注在类上，用作标注该类为切面处理类
@Pointcut	标注在@Aop 下的方法上，用作标注该切面处理类的切点
@Around	标注在拥有@Aop 的类下的方法上。用作标注作用于该切面处理类下切点环绕增强方法
@Before	标注在@Aop 下的方法上。用作标注作用于该切面处理类下切点前置增强方法
@After	标注在@Aop 下的方法上，用作标注作用于该切面处理类下切点后置增强方法
@Throw	标注在@Aop 下的方法上，用作标注作用于该切面处理类下切点出现未捕获异常调用方法

以下代码在快速启动的基础上添加@EnableAop 来启动 Aop 模块的功能。

```
@IocMain
@IocScan
@EnableAop
public class App {
    public static void main(String[] args) {
        IocContext.run(App.class);
    }
}
```

以下代码会对 org.jdragon.springz.test2.testApp.AopService 开头的类中以 get

开头的方法进行代理，在执行 AopService 开头的类其中以 get 开头的方法时，会在执行前进行 beforeMethod 方法的执行，在执行后进行 afterMethod 方法的执行。执行 aroundMethod 方法来代替被代理方法的执行，在 aroundMethod 方法中对被代理方法调用，达到环绕增强的目的。并且在其出现未抛出解决的异常时，会将异常抛到@Throw 方法下被处理。

```
@Component
@Aop(value = @Pointcut(value = "org.jdragon.springz.test2.testAop.AopService*",
method = "get*"), order = 1)
public class AopPointCut2 {
    private final Logger logger = LoggerFactory.getLogger(getClass());
    @Around
    public Object aroundMethod(JoinPoint joinPoint) {
        logger.debug("前置环绕  AopService1*切面");
        Object proceed = joinPoint.proceed();
        logger.debug("后置环绕  AopService1*切面");
        return proceed;
    }
    @Before
    public void beforeMethod(JoinPoint joinPoint) {
        logger.debug("前置 AopService1*切面");
    }

    @After
    public void afterMethod(JoinPoint joinPoint) {
        logger.debug("后置 AopService1*切面");
    }

    @Throw
    public void throwable(Throwable throwable) {
        logger.warn("执行异常处理");
    }
}
```

6.2.2 web 模块测试举例

在 web 模块中各注解的作用如表 6.3 所示。

表 6.3 web 模块所有注解表

注解	描述
@EnableWeb	标注在启动类上，开启 web 服务功能
@WebRoute	标注在类上，用作标注该类注册路由映射。该类也存在@Get、@Post 上，意味着他们也是路由。有两个属性：value 代表着请求路径，method 代表是请求方法
@WebParam	标注在参数类型注解上，代表这个注解是用在路由映射方法的参数上的
@Get	标注在方法上，代表这个为 Get 请求映射方法，该注解的值拼上类上 @WebRoute 的值就是请求路径，注解标注的方法就是请求处理方法。与 @WebRoute(method = RequestMethod.GET)等价
@Post	标注在方法上，代表这个为 Post 请求映射方法，该注解的值拼上类上 @WebRoute 的值就是请求路径，注解标注的方法就是请求处理方法。与 @WebRoute(method = RequestMethod.POST)等价
@RequestParam	参数类型注解，标注在路由方法的参数上，表示该参数是的 param 参数
@RequestBody	参数类型注解，标注在路由方法的参数上，表示该参数是请求的 body
@RequestHeader	参数类型注解，标注在路由方法的参数上，表示该参数是从 header 中取值
@PathVariable	参数类型注解，标注在路由方法的参数上，表示该参数是 restful 风格的路径参数

以下代码在快速启动的基础上添加@EnableWeb 来启动 Web 模块的功能。

```
@IocMain
@IocScan
@EnableWeb
public class App {
    public static void main(String[] args) {
        IocContext.run(App.class);
    }
}
```

在使用 Web 模块之前，要在 application.yml 中添加一些配置项，配置文件添加如下内容。代表着将 Http 服务访问端口设置为 18099，并且从/template/中读取以 html 为后缀的视图模板。

```
server:
  http:
    port: 18099
    prefix: /template/
    suffix: html
```

以下代码使用@WebRoute 来创建一个/user 路径的 UserController 路由类。get 请求/user/test 会调用 getTest 方法处理，post 请求/user/test 会调用 postTest 方法并注入四种不同的参数进行处理。在没有为 WebContent 设置视图名称时，会直接将对象序列化为 json 字符串返回。

```
@Controller
@WebRoute("/user")
public class UserController {
    @Get("/test")
    public User getTest() {
        return new User();
    }
    @Post("/test/{path}")
    public String postTest(@RequestParam String add,
                           @PathVariable String path,
                           @RequestHeader String header,
                           @RequestBody User user) {
        return "add:" + add +
                " path:" + path +
                " header:" + header +
                " user:" + user;
    }
}
```

以下代码是对 WebContent 设置视图，这时候返回的通过视图解析器注入了 atrributes 的视图。

```
@Controller
@WebRoute("/view")
public class ViewController {
    @Get("/test")
```

```
    public void test(WebContent webContent) {
        webContent.setViewName("test");
    }

    @Get("/webRequest")
    public void test(WebContent webContent, @RequestParam("msg") String msg) {
        webContent.addAttributes("message", msg);
        webContent.setViewName("webRequest");
    }
}
```

以下代码是实现了 WebFilter 的自定义过滤器，可以通过 httpRequest 来获取一些请求参数，也可以通过 httpResponse 进行信息响应，可以通过执行 webFilterChain.filter 来将请求传递给下一个过滤器，如果你要拦截这个请求，则不执行这个 filter 方法。

```
@Component
public class TestFilter implements WebFilter {
    @Override
    public void filter(HttpRequest httpRequest, HttpResponse httpResponse, WebFilterChain webFilterChain) {
        log.debug("测试拦截器");
        webFilterChain.filter(httpRequest, httpResponse);
    }
}
```

可以通过继承 WebConfiguration 来为 Web 服务添加过滤器和资源映射目录。以下代码为服务添加刚刚实现的 TestFilter 与添加资源映射。

```
@Configuration
public class WebConfig extends WebConfiguration {
    @Resource
    private WebFilter testFilter;

    @Override
    protected void addWebFilters(WebFilterRegistry webFilterRegistry) {
        webFilterRegistry.addWebFilter(testFilter)
```

```
                .addPathPatterns("/**").addExcludePatterns("/static/**");
    }
    @Override
    protected void addResourceMapping(ResourceMappingRegistry registry) {
        registry.addResourceMapping("/static/**")
                .setResourceLocation("/var/java/springz/");

        registry.addResourceMapping("/classpath/**")
                .setResourceLocation("classpath:/static/");
    }
}
```

6.2.3 feign 模块测试举例

在 feign 模块中各注解的作用如表 6.4 所示。

表 6.4feign 模块所有注解表

注解	描述
@EnableFeign	标注在启动类上，开启 feign 功能
@ZFeign	标注在类上，用作标注该类为外部系统接口类

以下代码在快速启动的基础上添加@EnableFeign 来启动 Feign 模块的功能。

```
@IocMain
@IocScan
@EnableFeign
public class App {
    public static void main(String[] args) {
        IocContext.run(App.class);
    }
}
```

通过以上代码，执 ClGroupMap 接口的 getGroupMap 方法，相当于请求 https://cl.tyu.wiki/robot/query/getGroupMap 这个接口。

```
@ZFeign(baseUrl = "https://cl.tyu.wiki", basePath = "/robot", depth = "result")
public interface ClGroupMap {
```

```
    @Post("/query/getGroupMap")
    List<Map<String, String>> getGroupMap();
}
```

这个接口返回的结果如下

```
{
  "code": 200,
  "message": "获取成功",
  "result": [
    {
      "groupName": "聚贤",
      "groupId": 12941287
    },
    {
      "groupName": "五林",
      "groupId": 26053477
    }
}
```

然后对这个返回结果根据 depth 解构，最后会提取出 result 的值，然后序列化成 List<Map<String,String>>进行返回。最后得到结果如下。

```
[{groupName=聚贤,groupId=12941287},{groupName=五林,groupId=26053477}]
```

6.3 本章小结

本章进行了框架的 core、aop、web、feign 模块的简单使用示例，可以使用户更快速上手框架使用。

结论

基于 java 注解和反射结合众多设计模式，使用 maven 的进行多模块管理，设计并实现了 ioc 对象管理容器框架，对象注册到对象管理容器中，利用控制反转的方式来对对象依赖的对象的注入。

在框架核心设计中，解决了因对象传递关系紊乱，需要参数较多较杂时的主动构建难度较大的开发难题。同时也最大程度的遵循了依赖倒置原则，可以利用向上转型的特性来对抽象声明注入具体实现，使对象之间的依赖关系尽可能的呈现出依赖抽象而不依赖具体实现。其中还支持对对象的生命周期的初始化方法与销毁方法的注册与执行。用户可以利用 application.yml 配置文件来初始化配置项目。

在原有设计基础上注重框架可扩展性，在框架扩展设计中对这一概念进行了实施。其中包含提供面向切面编程 Aop、提供 Http 接口服务能力、提供访问外部网络 Api 像调用方法一样方便的 Feign 服务。

参 考 文 献

[1] 周洪斌.IoC 模式及其应用[J].微型机与应用,2016,35(06):82-84.

[2] Rod Johnson / Juergen Hoeller.Expert One-on-One J2EE Development without EJB[M].电子工业出版社:北京,2005:1.

[3] Martin Fowler.Inversion of Control Containers and the Dependency Injection pattern[EB/OL].https://martinfowler.com/articles/injection.html,2004 年 1 月.

[4] 王咏武.向依赖关系宣战——依赖倒置、控制反转和依赖注入辨析[J].程序员,2005,1:98-103.

[5] 温立辉.Spring 框架在模型层的应用及原理[J].福建电脑,2017,5:147-148.

[6] 姜林美,李国刚,杜勇前.结合 AOP 思想和依赖注入技术的轻量级 MVC 框架[J].华侨大学学报(自然科学版),2016,37(1):1.

[7] 史梦安,马壮.一种基于 Servlet 的控制层软件框架设计[J].软件导刊,2017,16(3):1.

[8] 结城浩.图解设计模式[M].人民邮电出版社:北京,2017:1.

[9] 王臻,郭芊羽.基于 Spring 框架的依赖注入研究[J].才智,2014,34:158-158.

[10] 刘双.Spring 框架中 IOC 的实现[J].Program Design,2018,21:231-231.

致　　谢

衷心的感谢计算机专业各位老师，在大学学习期间，给予了我极大地鼓励和帮助，在学习上给予了我严谨、耐心的指导，在生活上给与了我亲切、热情的关怀。老师们渊博的学识、谦逊、谨慎的治学作风，一丝不苟、尽职尽责的工作态度以及正直的为人之道，都将是我终身受益，并激励我始终刻苦努力。在此，我向各位老师表示崇高的敬意和衷心的感谢！