**史上最全 Java 面试题 30 专题 总目录**

| **精心梳理、吐血推荐、史上最强、建议收藏** | **阿里、京东、美团、头条.... *随意挑、横着走！！！*** |
| --- | --- |
| 1.[Java算法面试题（史上最强、持续更新、吐血推荐）](https://www.cnblogs.com/crazymakercircle/p/14377410.html) | 2.[Java基础面试题（史上最全、持续更新、吐血推荐）](https://www.cnblogs.com/crazymakercircle/p/14366081.html) |
| 3.[**JVM面试题**（史上最强、持续更新、吐血推荐）](https://www.cnblogs.com/crazymakercircle/p/14365820.html) | 4、[**架构设计面试题** （史上最全、持续更新、吐血推荐）](https://www.cnblogs.com/crazymakercircle/p/14367907.html) |
| 5、[Spring面试题 专题](https://www.cnblogs.com/crazymakercircle/p/14465630.html) | 6、[SpringMVC面试题 专题](https://www.cnblogs.com/crazymakercircle/p/14465630.html) |
| 7.[SpringBoot - 面试题（史上最强、持续更新）](https://www.cnblogs.com/crazymakercircle/p/14365487.html) | 8、[Tomcat面试题 专题部分](https://www.cnblogs.com/crazymakercircle/p/14465630.html) |
| 9.[**网络协议面试题**（史上最全、持续更新、吐血推荐）](https://www.cnblogs.com/crazymakercircle/p/14370335.html) | 10、[**TCP/IP协议**（图解+秒懂+史上最全）](https://www.cnblogs.com/crazymakercircle/p/14499211.html) |
| 11.[**JUC并发包与容器** - 面试题（史上最强、持续更新）](https://www.cnblogs.com/crazymakercircle/p/13904029.html) | 12、[**设计模式面试题** （史上最全、持续更新、吐血推荐）](https://www.cnblogs.com/crazymakercircle/p/14367101.html) |
| 13.[死锁面试题（史上最强、持续更新）](https://www.cnblogs.com/crazymakercircle/p/14323919.html) | 15.[**Zookeeper 分布式锁** （图解+秒懂+史上最全）](https://www.cnblogs.com/crazymakercircle/p/14504520.html) |
| 14、[Redis 面试题 - 收藏版（史上最强、持续更新）](https://www.cnblogs.com/crazymakercircle/p/13900198.html) | 16、[Zookeeper 面试题（史上最强、持续更新）](https://www.cnblogs.com/crazymakercircle/p/13900183.html) |
| 17、[分布式事务面试题 （史上最全、持续更新、吐血推荐）](https://www.cnblogs.com/crazymakercircle/p/14375424.html) | 18、[**一致性协议** （史上最全）](https://www.cnblogs.com/crazymakercircle/p/14334422.html) |
| 19、[Zab协议 (史上最全)](https://www.cnblogs.com/crazymakercircle/p/14339702.html) | 20、[Paxos 图解 (秒懂)](https://www.cnblogs.com/crazymakercircle/p/14341015.html) |
| 21、[raft 图解 (秒懂)](https://www.cnblogs.com/crazymakercircle/p/14343154.html) | 26、[消息队列、RabbitMQ、Kafka、RocketMQ面试题 （史上最全、持续更新）](https://www.cnblogs.com/crazymakercircle/p/14367425.html) |
| 22.[Linux面试题（史上最全、持续更新、吐血推荐）](https://www.cnblogs.com/crazymakercircle/p/14366893.html) | 23、[Mysql 面试题（史上最强、持续更新）](https://www.cnblogs.com/crazymakercircle/p/13900186.html) |
| 24、[SpringCloud 面试题 - 收藏版（史上最强、持续更新）](https://www.cnblogs.com/crazymakercircle/p/13900212.html) | 25、[Netty 面试题 （史上最强、持续更新）](https://www.cnblogs.com/crazymakercircle/p/13903625.html) |
| 27、[**内存泄漏 内存溢出**（史上最全）](https://www.cnblogs.com/crazymakercircle/p/14389201.html) | 28、[JVM 内存溢出 实战 （史上最全）](https://www.cnblogs.com/crazymakercircle/p/14389389.html) |
| 29、[**多线程面试题**（史上最全）](https://www.cnblogs.com/crazymakercircle/p/14655412.html) | 30、[HR面经：过五关斩六将后，小心阴沟翻船！（史上最全）](https://www.cnblogs.com/crazymakercircle/p/14659738.html) |

**Spring面试题 专题部分**

**什么是spring?**

Spring是**一个轻量级Java开发框架**，最早有**Rod Johnson**创建，目的是为了解决企业级应用开发的业务逻辑层和其他各层的耦合问题。它是一个分层的JavaSE/JavaEE full-stack（一站式）轻量级开源框架，为开发Java应用程序提供全面的基础架构支持。Spring负责基础架构，因此Java开发者可以专注于应用程序的开发。

Spring最根本的使命是**解决企业级应用开发的复杂性，即简化Java开发**。

Spring可以做很多事情，它为企业级开发提供给了丰富的功能，但是这些功能的底层都依赖于它的两个核心特性，也就是**依赖注入（dependency injection，DI）和面向切面编程（aspect-oriented programming，AOP）**。

为了降低Java开发的复杂性，Spring采取了以下4种关键策略

* 基于POJO的轻量级和最小侵入性编程；
* 通过依赖注入和面向接口实现松耦合；
* 基于切面和惯例进行声明式编程；
* 通过切面和模板减少样板式代码。

**Spring框架的设计目标，设计理念，和核心是什么**

**Spring设计目标**：Spring为开发者提供一个一站式轻量级应用开发平台；

**Spring设计理念**：在JavaEE开发中，支持POJO和JavaBean开发方式，使应用面向接口开发，充分支持OO（面向对象）设计方法；Spring通过IoC容器实现对象耦合关系的管理，并实现依赖反转，将对象之间的依赖关系交给IoC容器，实现解耦；

**Spring框架的核心**：IoC容器和AOP模块。通过IoC容器管理POJO对象以及他们之间的耦合关系；通过AOP以动态非侵入的方式增强服务。

IoC让相互协作的组件保持松散的耦合，而AOP编程允许你把遍布于应用各层的功能分离出来形成可重用的功能组件。

**Spring的优缺点是什么？**

优点

* 方便解耦，简化开发

Spring就是一个大工厂，可以将所有对象的创建和依赖关系的维护，交给Spring管理。

* AOP编程的支持

Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能。

* 声明式事务的支持

只需要通过配置就可以完成对事务的管理，而无需手动编程。

* 方便程序的测试

Spring对Junit4支持，可以通过注解方便的测试Spring程序。

* 方便集成各种优秀框架

Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架的直接支持（如：Struts、Hibernate、MyBatis等）。

* 降低JavaEE API的使用难度

Spring对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低。

缺点

* Spring明明一个很轻量级的框架，却给人感觉大而全
* Spring依赖反射，反射影响性能
* 使用门槛升高，入门Spring需要较长时间

**Spring有哪些应用场景**

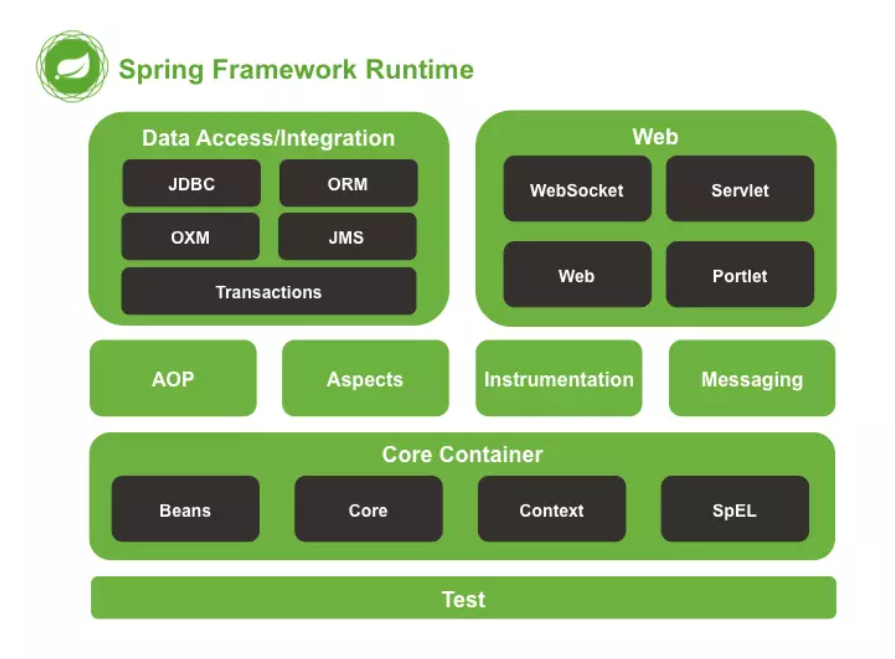
**应用场景**：JavaEE企业应用开发，包括SSH、SSM等

**Spring价值**：

* Spring是非侵入式的框架，目标是使应用程序代码对框架依赖最小化；
* Spring提供一个一致的编程模型，使应用直接使用POJO开发，与运行环境隔离开来；
* Spring推动应用设计风格向面向对象和面向接口开发转变，提高了代码的重用性和可测试性；

**Spring由哪些模块组成？**

Spring 总共大约有 20 个模块， 由 1300 多个不同的文件构成。 而这些组件被分别整合在**核心容器（Core Container）** 、 **AOP（Aspect Oriented Programming）和设备支持（Instrmentation）** 、**数据访问与集成（Data Access/Integeration）** 、 **Web**、 **消息（Messaging）** 、 **Test**等 6 个模块中。 以下是 Spring 5 的模块结构图：

[](https://img-blog.csdnimg.cn/2019102923475419.png)

* spring core：提供了框架的基本组成部分，包括控制反转（Inversion of Control，IOC）和依赖注入（Dependency Injection，DI）功能。
* spring beans：提供了BeanFactory，是工厂模式的一个经典实现，Spring将管理对象称为Bean。
* spring context：构建于 core 封装包基础上的 context 封装包，提供了一种框架式的对象访问方法。
* spring jdbc：提供了一个JDBC的抽象层，消除了烦琐的JDBC编码和数据库厂商特有的错误代码解析， 用于简化JDBC。
* spring aop：提供了面向切面的编程实现，让你可以自定义拦截器、切点等。
* spring Web：提供了针对 Web 开发的集成特性，例如文件上传，利用 servlet listeners 进行 ioc 容器初始化和针对 Web 的 ApplicationContext。
* spring test：主要为测试提供支持的，支持使用JUnit或TestNG对Spring组件进行单元测试和集成测试。

**Spring 框架中都用到了哪些设计模式？**

1. 工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；
2. 单例模式：Bean默认为单例模式。
3. 代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；
4. 模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。
5. 观察者模式：定义对象键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会得到通知被制动更新，如Spring中listener的实现–ApplicationListener。

**详细讲解一下核心容器（spring context应用上下文) 模块**

这是基本的Spring模块，提供spring 框架的基础功能，BeanFactory 是 任何以spring为基础的应用的核心。Spring 框架建立在此模块之上，它使Spring成为一个容器。

Bean 工厂是工厂模式的一个实现，提供了控制反转功能，用来把应用的配置和依赖从真正的应用代码中分离。最常用的就是org.springframework.beans.factory.xml.XmlBeanFactory ，它根据XML文件中的定义加载beans。该容器从XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

**Spring框架中有哪些不同类型的事件**

Spring 提供了以下5种标准的事件：

1. 上下文更新事件（ContextRefreshedEvent）：在调用ConfigurableApplicationContext 接口中的refresh()方法时被触发。
2. 上下文开始事件（ContextStartedEvent）：当容器调用ConfigurableApplicationContext的Start()方法开始/重新开始容器时触发该事件。
3. 上下文停止事件（ContextStoppedEvent）：当容器调用ConfigurableApplicationContext的Stop()方法停止容器时触发该事件。
4. 上下文关闭事件（ContextClosedEvent）：当ApplicationContext被关闭时触发该事件。容器被关闭时，其管理的所有单例Bean都被销毁。
5. 请求处理事件（RequestHandledEvent）：在Web应用中，当一个http请求（request）结束触发该事件。如果一个bean实现了ApplicationListener接口，当一个ApplicationEvent 被发布以后，bean会自动被通知。

**Spring 应用程序有哪些不同组件？**

Spring 应用一般有以下组件：

* 接口 - 定义功能。
* Bean 类 - 它包含属性，setter 和 getter 方法，函数等。
* Bean 配置文件 - 包含类的信息以及如何配置它们。
* Spring 面向切面编程（AOP） - 提供面向切面编程的功能。
* 用户程序 - 它使用接口。

**使用 Spring 有哪些方式？**

使用 Spring 有以下方式：

* 作为一个成熟的 Spring Web 应用程序。
* 作为第三方 Web 框架，使用 Spring Frameworks 中间层。
* 作为企业级 Java Bean，它可以包装现有的 POJO（Plain Old Java Objects）。
* 用于远程使用。

**Spring控制反转(IOC)（13）**

**什么是Spring IOC 容器？**

控制反转即IoC (Inversion of Control)，它把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的“控制反转”概念就是对组件对象控制权的转移，从程序代码本身转移到了外部容器。

Spring IOC 负责创建对象，管理对象（通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。

**控制反转(IoC)有什么作用**

* 管理对象的创建和依赖关系的维护。对象的创建并不是一件简单的事，在对象关系比较复杂时，如果依赖关系需要程序猿来维护的话，那是相当头疼的
* 解耦，由容器去维护具体的对象
* 托管了类的产生过程，比如我们需要在类的产生过程中做一些处理，最直接的例子就是代理，如果有容器程序可以把这部分处理交给容器，应用程序则无需去关心类是如何完成代理的

**IOC的优点是什么？**

* IOC 或 依赖注入把应用的代码量降到最低。
* 它使应用容易测试，单元测试不再需要单例和JNDI查找机制。
* 最小的代价和最小的侵入性使松散耦合得以实现。
* IOC容器支持加载服务时的饿汉式初始化和懒加载。

**Spring IoC 的实现机制**

Spring 中的 IoC 的实现原理就是工厂模式加反射机制。

示例：

interface Fruit {

public abstract void eat();

}

class Apple implements Fruit {

public void eat(){

System.out.println("Apple");

}

}

class Orange implements Fruit {

public void eat(){

System.out.println("Orange");

}

}

class Factory {

public static Fruit getInstance(String ClassName) {

Fruit f=null;

try {

f=(Fruit)Class.forName(ClassName).newInstance();

} catch (Exception e) {

e.printStackTrace();

}

return f;

}

}

class Client {

public static void main(String[] a) {

Fruit f=Factory.getInstance("io.github.dunwu.spring.Apple");

if(f!=null){

f.eat();

}

}

}

**Spring 的 IoC支持哪些功能**

Spring 的 IoC 设计支持以下功能：

* 依赖注入
* 依赖检查
* 自动装配
* 支持集合
* 指定初始化方法和销毁方法
* 支持回调某些方法（但是需要实现 Spring 接口，略有侵入）

其中，最重要的就是依赖注入，从 XML 的配置上说，即 ref 标签。对应 Spring RuntimeBeanReference 对象。

对于 IoC 来说，最重要的就是容器。容器管理着 Bean 的生命周期，控制着 Bean 的依赖注入。

**BeanFactory 和 ApplicationContext有什么区别？**

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。

其中ApplicationContext是BeanFactory的子接口。

**依赖关系**

BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。

ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

* 继承MessageSource，因此支持国际化。
* 统一的资源文件访问方式。
* 提供在监听器中注册bean的事件。
* 同时加载多个配置文件。
* 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

**加载方式**

BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

**创建方式**

BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

**注册方式**

BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

**Spring 如何设计容器的，BeanFactory和ApplicationContext的关系详解**

Spring 作者 Rod Johnson 设计了两个接口用以表示容器。

* BeanFactory
* ApplicationContext

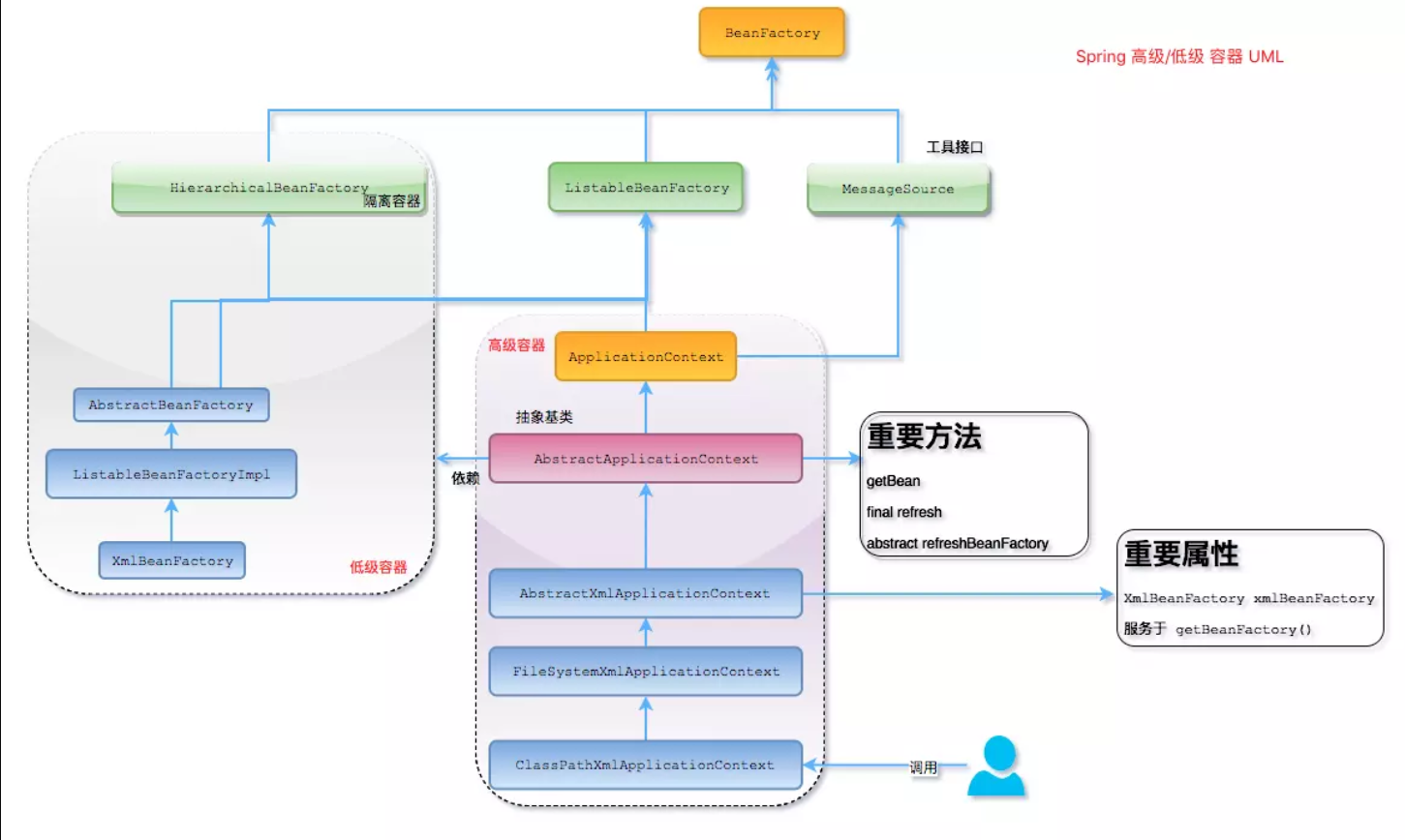
BeanFactory 简单粗暴，可以理解为就是个 HashMap，Key 是 BeanName，Value 是 Bean 实例。通常只提供注册（put），获取（get）这两个功能。我们可以称之为 **“低级容器”**。

ApplicationContext 可以称之为 **“高级容器”**。因为他比 BeanFactory 多了更多的功能。他继承了多个接口。因此具备了更多的功能。例如资源的获取，支持多种消息（例如 JSP tag 的支持），对 BeanFactory 多了工具级别的支持等待。所以你看他的名字，已经不是 BeanFactory 之类的工厂了，而是 “应用上下文”， 代表着整个大容器的所有功能。该接口定义了一个 refresh 方法，此方法是所有阅读 Spring 源码的人的最熟悉的方法，用于刷新整个容器，即重新加载/刷新所有的 bean。

当然，除了这两个大接口，还有其他的辅助接口，这里就不介绍他们了。

BeanFactory和ApplicationContext的关系

为了更直观的展示 “低级容器” 和 “高级容器” 的关系，这里通过常用的 ClassPathXmlApplicationContext 类来展示整个容器的层级 UML 关系。

[](https://img-blog.csdnimg.cn/20191105111441363.png)

有点复杂？ 先不要慌，我来解释一下。

最上面的是 BeanFactory，下面的 3 个绿色的，都是功能扩展接口，这里就不展开讲。

看下面的隶属 ApplicationContext 粉红色的 “高级容器”，依赖着 “低级容器”，这里说的是依赖，不是继承哦。他依赖着 “低级容器” 的 getBean 功能。而高级容器有更多的功能：支持不同的信息源头，可以访问文件资源，支持应用事件（Observer 模式）。

通常用户看到的就是 “高级容器”。 但 BeanFactory 也非常够用啦！

左边灰色区域的是 “低级容器”， 只负载加载 Bean，获取 Bean。容器其他的高级功能是没有的。例如上图画的 refresh 刷新 Bean 工厂所有配置，生命周期事件回调等。

小结

说了这么多，不知道你有没有理解Spring IoC？ 这里小结一下：IoC 在 Spring 里，只需要低级容器就可以实现，2 个步骤：

1. 加载配置文件，解析成 BeanDefinition 放在 Map 里。
2. 调用 getBean 的时候，从 BeanDefinition 所属的 Map 里，拿出 Class 对象进行实例化，同时，如果有依赖关系，将递归调用 getBean 方法 —— 完成依赖注入。

上面就是 Spring 低级容器（BeanFactory）的 IoC。

至于高级容器 ApplicationContext，他包含了低级容器的功能，当他执行 refresh 模板方法的时候，将刷新整个容器的 Bean。同时其作为高级容器，包含了太多的功能。一句话，他不仅仅是 IoC。他支持不同信息源头，支持 BeanFactory 工具类，支持层级容器，支持访问文件资源，支持事件发布通知，支持接口回调等等。

**ApplicationContext通常的实现是什么？**

**FileSystemXmlApplicationContext** ：此容器从一个XML文件中加载beans的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。

**ClassPathXmlApplicationContext**：此容器也从一个XML文件中加载beans的定义，这里，你需要正确设置classpath因为这个容器将在classpath里找bean配置。

**WebXmlApplicationContext**：此容器加载一个XML文件，此文件定义了一个WEB应用的所有bean。

**什么是Spring的依赖注入？**

控制反转IoC是一个很大的概念，可以用不同的方式来实现。其主要实现方式有两种：依赖注入和依赖查找

依赖注入：相对于IoC而言，依赖注入(DI)更加准确地描述了IoC的设计理念。所谓依赖注入（Dependency Injection），即组件之间的依赖关系由容器在应用系统运行期来决定，也就是由容器动态地将某种依赖关系的目标对象实例注入到应用系统中的各个关联的组件之中。组件不做定位查询，只提供普通的Java方法让容器去决定依赖关系。

**依赖注入的基本原则**

依赖注入的基本原则是：应用组件不应该负责查找资源或者其他依赖的协作对象。配置对象的工作应该由IoC容器负责，“查找资源”的逻辑应该从应用组件的代码中抽取出来，交给IoC容器负责。容器全权负责组件的装配，它会把符合依赖关系的对象通过属性（JavaBean中的setter）或者是构造器传递给需要的对象。

**依赖注入有什么优势**

依赖注入之所以更流行是因为它是一种更可取的方式：让容器全权负责依赖查询，受管组件只需要暴露JavaBean的setter方法或者带参数的构造器或者接口，使容器可以在初始化时组装对象的依赖关系。其与依赖查找方式相比，主要优势为：

* 查找定位操作与应用代码完全无关。
* 不依赖于容器的API，可以很容易地在任何容器以外使用应用对象。
* 不需要特殊的接口，绝大多数对象可以做到完全不必依赖容器。

**有哪些不同类型的依赖注入实现方式？**

依赖注入是时下最流行的IoC实现方式，依赖注入分为接口注入（Interface Injection），Setter方法注入（Setter Injection）和构造器注入（Constructor Injection）三种方式。其中接口注入由于在灵活性和易用性比较差，现在从Spring4开始已被废弃。

**构造器依赖注入**：构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。

**Setter方法注入**：Setter方法注入是容器通过调用无参构造器或无参static工厂 方法实例化bean之后，调用该bean的setter方法，即实现了基于setter的依赖注入。

**构造器依赖注入和 Setter方法注入的区别**

| **构造函数注入** | **setter 注入** |
| --- | --- |
| 没有部分注入 | 有部分注入 |
| 不会覆盖 setter 属性 | 会覆盖 setter 属性 |
| 任意修改都会创建一个新实例 | 任意修改不会创建一个新实例 |
| 适用于设置很多属性 | 适用于设置少量属性 |

两种依赖方式都可以使用，构造器注入和Setter方法注入。最好的解决方案是用构造器参数实现强制依赖，setter方法实现可选依赖。

**Spring Beans（19）**

**什么是Spring beans？**

Spring beans 是那些形成Spring应用的主干的java对象。它们被Spring IOC容器初始化，装配，和管理。这些beans通过容器中配置的元数据创建。比如，以XML文件中 的形式定义。

**一个 Spring Bean 定义 包含什么？**

一个Spring Bean 的定义包含容器必知的所有配置元数据，包括如何创建一个bean，它的生命周期详情及它的依赖。

**如何给Spring 容器提供配置元数据？Spring有几种配置方式**

这里有三种重要的方法给Spring 容器提供配置元数据。

* XML配置文件。
* 基于注解的配置。
* 基于java的配置。

**Spring配置文件包含了哪些信息**

Spring配置文件是个XML 文件，这个文件包含了类信息，描述了如何配置它们，以及如何相互调用。

**Spring基于xml注入bean的几种方式**

1. Set方法注入；
2. 构造器注入：①通过index设置参数的位置；②通过type设置参数类型；
3. 静态工厂注入；
4. 实例工厂；

**你怎样定义类的作用域？**

当定义一个 在Spring里，我们还能给这个bean声明一个作用域。它可以通过bean 定义中的scope属性来定义。如，当Spring要在需要的时候每次生产一个新的bean实例，bean的scope属性被指定为prototype。另一方面，一个bean每次使用的时候必须返回同一个实例，这个bean的scope 属性 必须设为 singleton。

**解释Spring支持的几种bean的作用域**

Spring框架支持以下五种bean的作用域：

* **singleton :** bean在每个Spring ioc 容器中只有一个实例。
* **prototype**：一个bean的定义可以有多个实例。
* **request**：每次http请求都会创建一个bean，该作用域仅在基于web的Spring ApplicationContext情形下有效。
* **session**：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。
* **global-session**：在一个全局的HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。

**注意：** 缺省的Spring bean 的作用域是Singleton。使用 prototype 作用域需要慎重的思考，因为频繁创建和销毁 bean 会带来很大的性能开销。

**Spring框架中的单例bean是线程安全的吗？**

不是，Spring框架中的单例bean不是线程安全的。

spring 中的 bean 默认是单例模式，spring 框架并没有对单例 bean 进行多线程的封装处理。

实际上大部分时候 spring bean 无状态的（比如 dao 类），所有某种程度上来说 bean 也是安全的，但如果 bean 有状态的话（比如 view model 对象），那就要开发者自己去保证线程安全了，最简单的就是改变 bean 的作用域，把“singleton”变更为“prototype”，这样请求 bean 相当于 new Bean()了，所以就可以保证线程安全了。

* 有状态就是有数据存储功能。
* 无状态就是不会保存数据。

**Spring如何处理线程并发问题？**

在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域，因为Spring对一些Bean中非线程安全状态采用ThreadLocal进行处理，解决线程安全问题。

ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而ThreadLocal采用了“空间换时间”的方式。

ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

**问题：解释Spring框架中bean的生命周期 (重要)**

Spring Bean的生命周期是Spring面试热点问题。这个问题即考察对Spring的微观了解，又考察对Spring的宏观认识，想要答好并不容易！本文希望能够从源码角度入手，帮助面试者彻底搞定Spring Bean的生命周期。

**参考答案:**

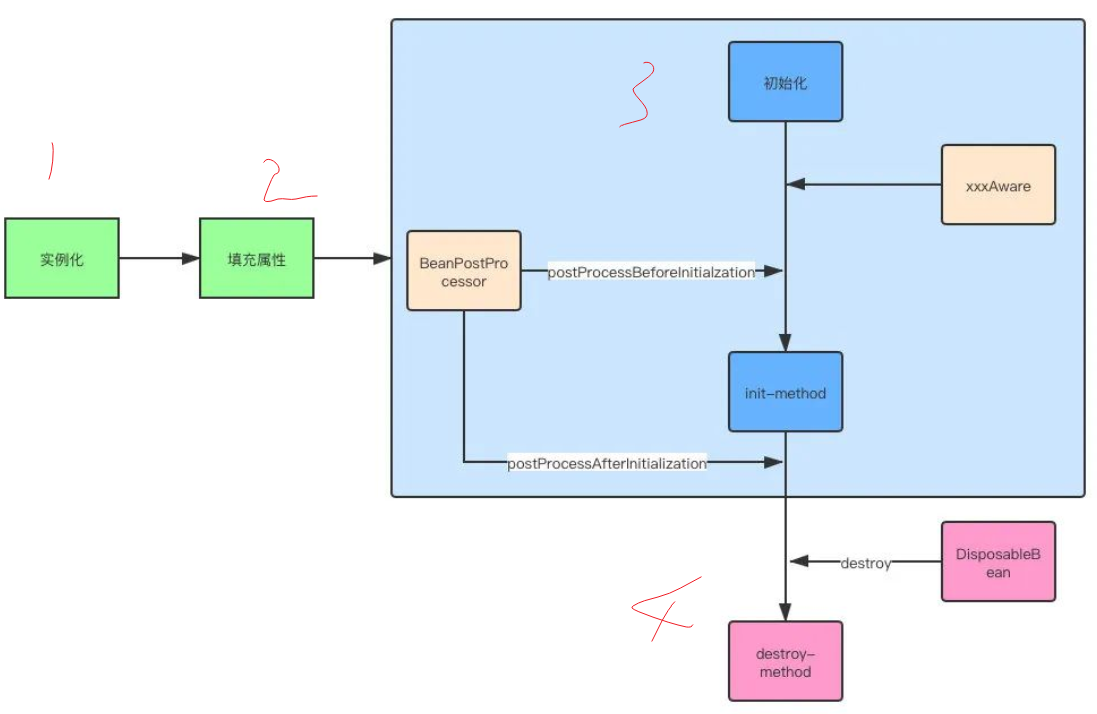
首先，回答阶段的数量：**只有四个！**

是的，Spring Bean的生命周期只有这四个阶段。把这四个阶段和每个阶段对应的扩展点糅合在一起虽然没有问题，但是这样非常凌乱，难以记忆。

要彻底搞清楚Spring的生命周期，首先要把这四个阶段牢牢记住。实例化和属性赋值对应构造方法和setter方法的注入，初始化和销毁是用户能自定义扩展的两个阶段。在这四步之间穿插的各种扩展点，稍后会讲。

1. 实例化 Instantiation
2. 属性赋值 Populate
3. 初始化 Initialization
4. 销毁 Destruction

**实例化 -> 属性赋值 -> 初始化 -> 销毁**

[](https://img-blog.csdnimg.cn/20210613122656176.png)

**各个阶段的工作:**

1. 实例化，创建一个Bean对象
2. 填充属性，为属性赋值
3. 初始化
   * 如果实现了**xxxAware**接口，通过不同类型的Aware接口拿到Spring容器的资源
   * 如果实现了BeanPostProcessor接口，则会回调该接口的**postProcessBeforeInitialzation**和**postProcessAfterInitialization**方法
   * 如果配置了**init-method**方法，则会执行**init-method**配置的方法
4. 销毁
   * 容器关闭后，如果Bean实现了**DisposableBean**接口，则会回调该接口的**destroy**方法
   * 如果配置了**destroy-method**方法，则会执行**destroy-method**配置的方法

**源码学习：**

前三个阶段，主要逻辑都在doCreate()方法中，逻辑很清晰，就是顺序调用以下三个方法，这三个方法与三个生命周期阶段一一对应，非常重要，在后续扩展接口分析中也会涉及。

1. createBeanInstance() -> 实例化
2. populateBean() -> 属性赋值
3. initializeBean() -> 初始化

注：bean的生命周期是从将bean定义全部注册到BeanFacotry中以后开始的。

源码如下，能证明实例化，属性赋值和初始化这三个生命周期的存在。关于本文的Spring源码都将忽略无关部分，便于理解：

**前三个阶段的源码：**

*// 忽略了无关代码*

protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullable Object[] args)

throws BeanCreationException {

*// Instantiate the bean.*

BeanWrapper instanceWrapper = null;

if (instanceWrapper == null) {

*// 实例化阶段！*

instanceWrapper = createBeanInstance(beanName, mbd, args);

}

*// Initialize the bean instance.*

Object exposedObject = bean;

try {

*// 属性赋值阶段！*

populateBean(beanName, mbd, instanceWrapper);

*// 初始化阶段！*

exposedObject = initializeBean(beanName, exposedObject, mbd);

}

}

上面这些这个实例化Bean的方法是在getBean()方法中调用的，而getBean是在finishBeanFactoryInitialization方法中调用的，用来实例化单例非懒加载Bean，源码如下：

@Override

public void refresh() throws BeansException, IllegalStateException {

synchronized (this.startupShutdownMonitor) {

try {

*// Allows post-processing of the bean factory in context subclasses.*

postProcessBeanFactory(beanFactory);

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

invokeBeanFactoryPostProcessors(beanFactory);

*// Register bean processors that intercept bean creation.*

*// 所有BeanPostProcesser初始化的调用点*

registerBeanPostProcessors(beanFactory);

*// Initialize message source for this context.*

initMessageSource();

*// Initialize event multicaster for this context.*

initApplicationEventMulticaster();

*// Initialize other special beans in specific context subclasses.*

onRefresh();

*// Check for listener beans and register them.*

registerListeners();

*// Instantiate all remaining (non-lazy-init) singletons.*

*// 所有单例非懒加载Bean的调用点*

finishBeanFactoryInitialization(beanFactory);

*// Last step: publish corresponding event.*

finishRefresh();

}

}

**销毁Bean阶段:**

至于销毁，是在容器关闭时调用的，详见 ConfigurableApplicationContext#close()

**高分答题的技巧:**

如果回答了上面的答案可以拿到100分的话，加上下面的内容，就是120分

**生命周期常用扩展点**

Spring生命周期相关的常用扩展点非常多，所以问题不是不知道，而是记不住或者记不牢。其实记不住的根本原因还是不够了解，这里通过源码+分类的方式帮大家记忆。

区分影响一个bean或者多个bean是从源码分析得出的.

以BeanPostProcessor为例：

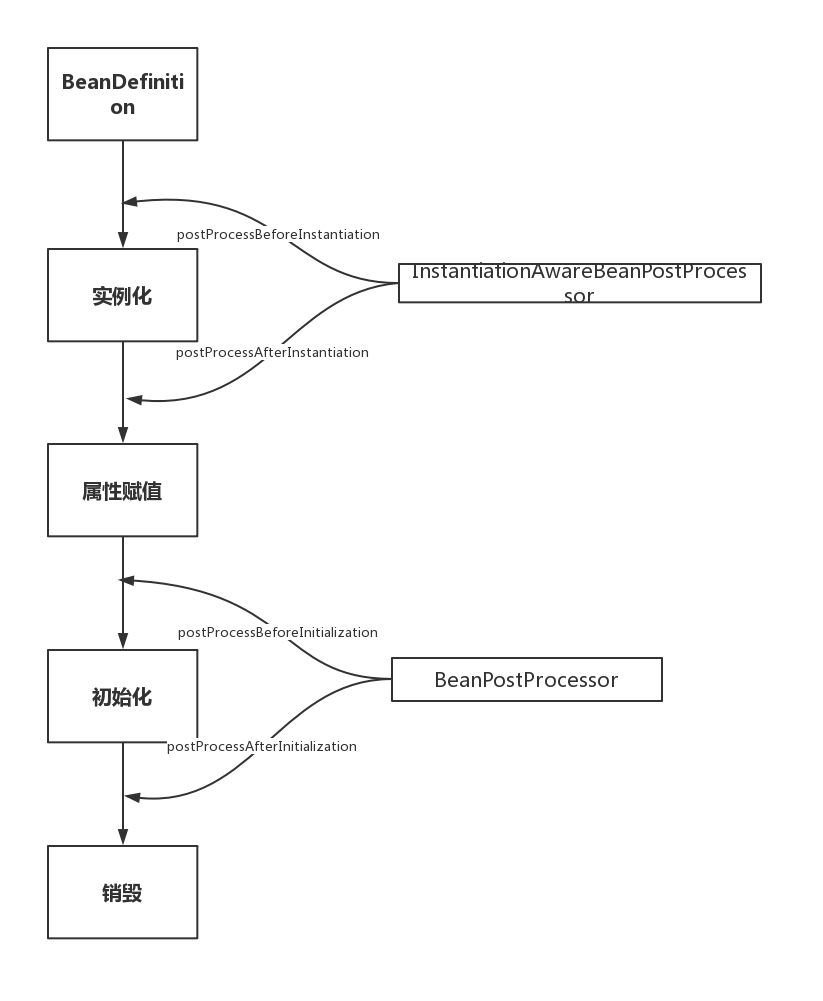
1. 从refresh方法来看,BeanPostProcessor 实例化比正常的bean早.
2. 从initializeBean方法看,每个bean初始化前后都调用所有BeanPostProcessor的postProcessBeforeInitialization和postProcessAfterInitialization方法.

**第一大类：影响多个Bean的接口**

实现了这些接口的Bean会切入到多个Bean的生命周期中。正因为如此，这些接口的功能非常强大，Spring内部扩展也经常使用这些接口，例如自动注入以及AOP的实现都和他们有关。

* InstantiationAwareBeanPostProcessor
* BeanPostProcessor

这两兄弟可能是Spring扩展中**最重要**的两个接口！InstantiationAwareBeanPostProcessor作用于**实例化**阶段的前后，BeanPostProcessor作用于**初始化**阶段的前后。正好和第一、第三个生命周期阶段对应。通过图能更好理解：

[](https://img-blog.csdnimg.cn/20200313090614543.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L2N5OTczMDcxMjYz,size_16,color_FFFFFF,t_70)

**InstantiationAwareBeanPostProcessor**

InstantiationAwareBeanPostProcessor实际上继承了BeanPostProcessor接口，严格意义上来看他们不是两兄弟，而是两父子。但是从生命周期角度我们重点关注其特有的对实例化阶段的影响，图中省略了从BeanPostProcessor继承的方法。

InstantiationAwareBeanPostProcessor extends BeanPostProcessor

**InstantiationAwareBeanPostProcessor源码分析：**

* **postProcessBeforeInstantiation调用点**，忽略无关代码：

@Override

protected Object createBean(String beanName, RootBeanDefinition mbd, @Nullable Object[] args)

throws BeanCreationException {

try {

*// Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.*

*// postProcessBeforeInstantiation方法调用点，这里就不跟进了，*

*// 有兴趣的同学可以自己看下，就是for循环调用所有的InstantiationAwareBeanPostProcessor*

Object bean = resolveBeforeInstantiation(beanName, mbdToUse);

if (bean != null) {

return bean;

}

}

try {

*// 上文提到的doCreateBean方法，可以看到*

*// postProcessBeforeInstantiation方法在创建Bean之前调用*

Object beanInstance = doCreateBean(beanName, mbdToUse, args);

if (logger.isTraceEnabled()) {

logger.trace("Finished creating instance of bean '" + beanName + "'");

}

return beanInstance;

}

}

可以看到，postProcessBeforeInstantiation在doCreateBean之前调用，也就是在bean实例化之前调用的，英文源码注释解释道该方法的返回值会替换原本的Bean作为代理，这也是Aop等功能实现的关键点。

* **postProcessAfterInstantiation调用点，**忽略无关代码：

protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw) {

*// Give any InstantiationAwareBeanPostProcessors the opportunity to modify the*

*// state of the bean before properties are set. This can be used, for example,*

*// to support styles of field injection.*

boolean continueWithPropertyPopulation = true;

*// InstantiationAwareBeanPostProcessor#postProcessAfterInstantiation()*

*// 方法作为属性赋值的前置检查条件，在属性赋值之前执行，能够影响是否进行属性赋值！*

if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

for (BeanPostProcessor bp : getBeanPostProcessors()) {

if (bp instanceof InstantiationAwareBeanPostProcessor) {

InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {

continueWithPropertyPopulation = false;

break;

}

}

}

}

*// 忽略后续的属性赋值操作代码*

}

可以看到该方法在属性赋值方法内，但是在真正执行赋值操作之前。其返回值为boolean，返回false时可以阻断属性赋值阶段（continueWithPropertyPopulation = false;）。

**BeanPostProcessor**

关于BeanPostProcessor执行阶段的源码穿插在下文Aware接口的调用时机分析中，因为部分Aware功能的就是通过他实现的!只需要先记住BeanPostProcessor在初始化前后调用就可以了。

**接口源码：**

public interface BeanPostProcessor {

*//bean初始化之前调用*

@Nullable

default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

return bean;

}

*//bean初始化之后调用*

@Nullable

default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

return bean;

}

}

**第二大类：只调用一次的接口**

这一大类接口的特点是功能丰富，常用于用户自定义扩展。

第二大类中又可以分为两类：

1. Aware类型的接口
2. 生命周期接口

**无所不知的Aware**

Aware类型的接口的作用就是让我们能够拿到Spring容器中的一些资源。基本都能够见名知意，Aware之前的名字就是可以拿到什么资源，例如BeanNameAware可以拿到BeanName，以此类推。调用时机需要注意：**所有的Aware方法都是在初始化阶段之前调用的！**

Aware接口众多，这里同样通过分类的方式帮助大家记忆。

Aware接口具体可以分为两组，至于为什么这么分，详见下面的源码分析。如下排列顺序同样也是Aware接口的执行顺序，能够见名知意的接口不再解释。

**Aware Group1**

1. BeanNameAware
2. BeanClassLoaderAware
3. BeanFactoryAware

**Aware Group2**

1. EnvironmentAware
2. EmbeddedValueResolverAware 这个知道的人可能不多，实现该接口能够获取Spring EL解析器，用户的自定义注解需要支持spel表达式的时候可以使用，非常方便。
3. ApplicationContextAware(ResourceLoaderAware\ApplicationEventPublisherAware\MessageSourceAware) 这几个接口可能让人有点懵，实际上这几个接口可以一起记，其返回值实质上都是当前的ApplicationContext对象，因为ApplicationContext是一个复合接口，如下：

public interface ApplicationContext extends EnvironmentCapable, ListableBeanFactory, HierarchicalBeanFactory,

MessageSource, ApplicationEventPublisher, ResourcePatternResolver {}

这里涉及到另一道面试题，ApplicationContext和BeanFactory的区别，可以从ApplicationContext继承的这几个接口入手，除去BeanFactory相关的两个接口就是ApplicationContext独有的功能，这里不详细说明。

**Aware调用时机源码分析**

详情如下，忽略了部分无关代码。代码位置就是我们上文提到的initializeBean方法详情，这也说明了Aware都是在初始化阶段之前调用的！

*// 见名知意，初始化阶段调用的方法*

protected Object initializeBean(final String beanName, final Object bean, @Nullable RootBeanDefinition mbd) {

*// 这里调用的是Group1中的三个Bean开头的Aware*

invokeAwareMethods(beanName, bean);

Object wrappedBean = bean;

*// 这里调用的是Group2中的几个Aware，*

*// 而实质上这里就是前面所说的BeanPostProcessor的调用点！*

*// 也就是说与Group1中的Aware不同，这里是通过BeanPostProcessor（ApplicationContextAwareProcessor）实现的。*

wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);

*// 这个是初始化方法，下文要介绍的InitializingBean调用点就是在这个方法里面*

invokeInitMethods(beanName, wrappedBean, mbd);

*// BeanPostProcessor的另一个调用点*

wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);

return wrappedBean;

}

可以看到并不是所有的Aware接口都使用同样的方式调用。Bean××Aware都是在代码中直接调用的，而ApplicationContext相关的Aware都是通过BeanPostProcessor#postProcessBeforeInitialization()实现的。感兴趣的可以自己看一下ApplicationContextAwareProcessor这个类的源码，就是判断当前创建的Bean是否实现了相关的Aware方法，如果实现了会调用回调方法将资源传递给Bean。

至于Spring为什么这么实现，应该没什么特殊的考量。也许和Spring的版本升级有关。基于对修改关闭，对扩展开放的原则，Spring对一些新的Aware采用了扩展的方式添加。

BeanPostProcessor的调用时机也能在这里体现，包围住invokeInitMethods方法，也就说明了在初始化阶段的前后执行。

关于Aware接口的执行顺序，其实只需要记住第一组在第二组执行之前就行了。每组中各个Aware方法的调用顺序其实没有必要记，有需要的时候点进源码一看便知。

**简单的两个生命周期接口**

至于剩下的两个生命周期接口就很简单了，实例化和属性赋值都是Spring帮助我们做的，能够自己实现的有初始化和销毁两个生命周期阶段。

**InitializingBean接口**

InitializingBean顾名思义，是初始化Bean相关的接口。

接口定义：

public interface InitializingBean {

void afterPropertiesSet() throws Exception;

}

看方法名，是在读完Properties文件，之后执行的方法。afterPropertiesSet()方法是在初始化过程中被调用的。

InitializingBean 对应生命周期的初始化阶段，在上面源码的invokeInitMethods(beanName, wrappedBean, mbd);方法中调用。

有一点需要注意，因为Aware方法都是执行在初始化方法之前，所以可以在初始化方法中放心大胆的使用Aware接口获取的资源，这也是我们自定义扩展Spring的常用方式。

除了实现InitializingBean接口之外还能通过注解（@PostConstruct）或者xml配置的方式指定初始化方法（init-method），至于这几种定义方式的调用顺序其实没有必要记。因为这几个方法对应的都是同一个生命周期，只是实现方式不同，我们一般只采用其中一种方式。

**三种实现指定初始化方法的方法：**

* 使用@PostConstruct注解，该注解作用于void方法上
* 在配置文件中配置init-method方法

<bean id="student" class="com.demo.Student" init-method="init2">

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

<property name="age" value="20"></property>

<property name="school" ref="school"></property>

</bean>

* 将类实现InitializingBean接口

@Component("student")

public class Student implements InitializingBean{

private String name;

private int age;

…

}

**执行：**

@Component("student")

public class Student implements InitializingBean{

private String name;

private int age;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

*//1.使用postconstrtct注解*

@PostConstruct

public void init(){

System.out.println("执行 init方法");

}

*//2.在xml配置文件中配置init-method方法*

public void init2(){

System.out.println("执行init2方法 ");

}

*//3.实现InitializingBean接口*

public void afterPropertiesSet() throws Exception {

System.out.println("执行init3方法");

}

}

通过测试我们可以得出结论，三种实现方式的执行顺序是：

**Constructor > @PostConstruct > InitializingBean > init-method**

**DisposableBean接口**

DisposableBean 类似于InitializingBean，对应生命周期的销毁阶段，**以ConfigurableApplicationContext#close()方法作为入口**，实现是通过循环获取所有实现了DisposableBean接口的Bean然后调用其destroy()方法 。

接口定义：

public interface DisposableBean {

void destroy() throws Exception;

}

定义一个实现了DisposableBean接口的Bean：

public class IndexBean implements InitializingBean,DisposableBean {

public void destroy() throws Exception {

System.out.println("destroy");

}

public void afterPropertiesSet() throws Exception {

System.out.println("init-afterPropertiesSet()");

}

public void test(){

System.out.println("init-test()");

}

}

执行：

public class Main {

public static void main(String[] args) {

AbstractApplicationContext applicationContext=new ClassPathXmlApplicationContext("classpath:application-usertag.xml");

System.out.println("init-success");

applicationContext.registerShutdownHook();

}

}

执行结果：

init-afterPropertiesSet()

init-test()

init-success

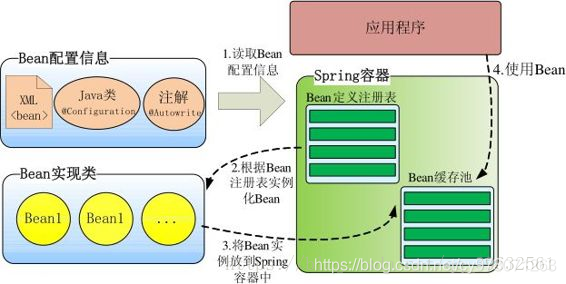
destroy

也就是说，在对象销毁的时候，会去调用DisposableBean的destroy方法。在进入到销毁过程时先去调用一下DisposableBean的destroy方法，然后后执行 destroy-method声明的方法（用来销毁Bean中的各项数据）。

**扩展阅读: BeanPostProcessor注册时机与执行顺序**

首先要明确一个概念，在spring中一切皆bean

所有的组件都会被作为一个bean装配到spring容器中，过程如下图：

[](https://img-blog.csdnimg.cn/20200313090614571.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L2N5OTczMDcxMjYz,size_16,color_FFFFFF,t_70)

所以我们前面所讲的那些拓展点，也都会被作为一个个bean装配到spring容器中

**注册时机**

我们知道BeanPostProcessor也会注册为Bean，那么Spring是如何保证BeanPostProcessor在我们的业务Bean之前初始化完成呢？

请看我们熟悉的refresh()方法的源码，省略部分无关代码（refresh的详细注解见refresh()）：

@Override

public void refresh() throws BeansException, IllegalStateException {

synchronized (this.startupShutdownMonitor) {

try {

*// Allows post-processing of the bean factory in context subclasses.*

postProcessBeanFactory(beanFactory);

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

invokeBeanFactoryPostProcessors(beanFactory);

*// Register bean processors that intercept bean creation.*

*// 注册所有BeanPostProcesser的方法*

registerBeanPostProcessors(beanFactory);

*// Initialize message source for this context.*

initMessageSource();

*// Initialize event multicaster for this context.*

initApplicationEventMulticaster();

*// Initialize other special beans in specific context subclasses.*

onRefresh();

*// Check for listener beans and register them.*

registerListeners();

*// Instantiate all remaining (non-lazy-init) singletons.*

*// 所有单例非懒加载Bean的创建方法*

finishBeanFactoryInitialization(beanFactory);

*// Last step: publish corresponding event.*

finishRefresh();

}

}

可以看出，Spring是先执行registerBeanPostProcessors()进行BeanPostProcessors的注册，然后再执行finishBeanFactoryInitialization创建我们的单例非懒加载的Bean。

**执行顺序**

BeanPostProcessor有很多个，而且每个BeanPostProcessor都影响多个Bean，其执行顺序至关重要，必须能够控制其执行顺序才行。关于执行顺序这里需要引入两个排序相关的接口：PriorityOrdered、Ordered

* PriorityOrdered是一等公民，首先被执行，PriorityOrdered公民之间通过接口返回值排序
* Ordered是二等公民，然后执行，Ordered公民之间通过接口返回值排序
* 都没有实现是三等公民，最后执行

在以下源码中，可以很清晰的看到Spring注册各种类型BeanPostProcessor的逻辑，根据实现不同排序接口进行分组。优先级高的先加入，优先级低的后加入。

*// First, invoke the BeanDefinitionRegistryPostProcessors that implement PriorityOrdered.*

*// 首先，加入实现了PriorityOrdered接口的BeanPostProcessors，顺便根据PriorityOrdered排了序*

String[] postProcessorNames =

beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);

for (String ppName : postProcessorNames) {

if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {

currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class));

processedBeans.add(ppName);

}

}

sortPostProcessors(currentRegistryProcessors, beanFactory);

registryProcessors.addAll(currentRegistryProcessors);

invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);

currentRegistryProcessors.clear();

// Next, invoke the BeanDefinitionRegistryPostProcessors that implement Ordered.

// 然后，加入实现了Ordered接口的BeanPostProcessors，顺便根据Ordered排了序

postProcessorNames = beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);

for (String ppName : postProcessorNames) {

if (!processedBeans.contains(ppName) && beanFactory.isTypeMatch(ppName, Ordered.class)) {

currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class));

processedBeans.add(ppName);

}

}

sortPostProcessors(currentRegistryProcessors, beanFactory);

registryProcessors.addAll(currentRegistryProcessors);

invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);

currentRegistryProcessors.clear();

// Finally, invoke all other BeanDefinitionRegistryPostProcessors until no further ones appear.

// 最后加入其他常规的BeanPostProcessors

boolean reiterate = true;

while (reiterate) {

reiterate = false;

postProcessorNames = beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);

for (String ppName : postProcessorNames) {

if (!processedBeans.contains(ppName)) {

currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class));

processedBeans.add(ppName);

reiterate = true;

}

}

sortPostProcessors(currentRegistryProcessors, beanFactory);

registryProcessors.addAll(currentRegistryProcessors);

invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);

currentRegistryProcessors.clear();

}

根据排序接口返回值排序，默认升序排序，返回值越低优先级越高。

*/\*\**

*\* Useful constant for the highest precedence value.*

*\* @see java.lang.Integer#MIN\_VALUE*

*\*/*

int HIGHEST\_PRECEDENCE = Integer.MIN\_VALUE;

*/\*\**

*\* Useful constant for the lowest precedence value.*

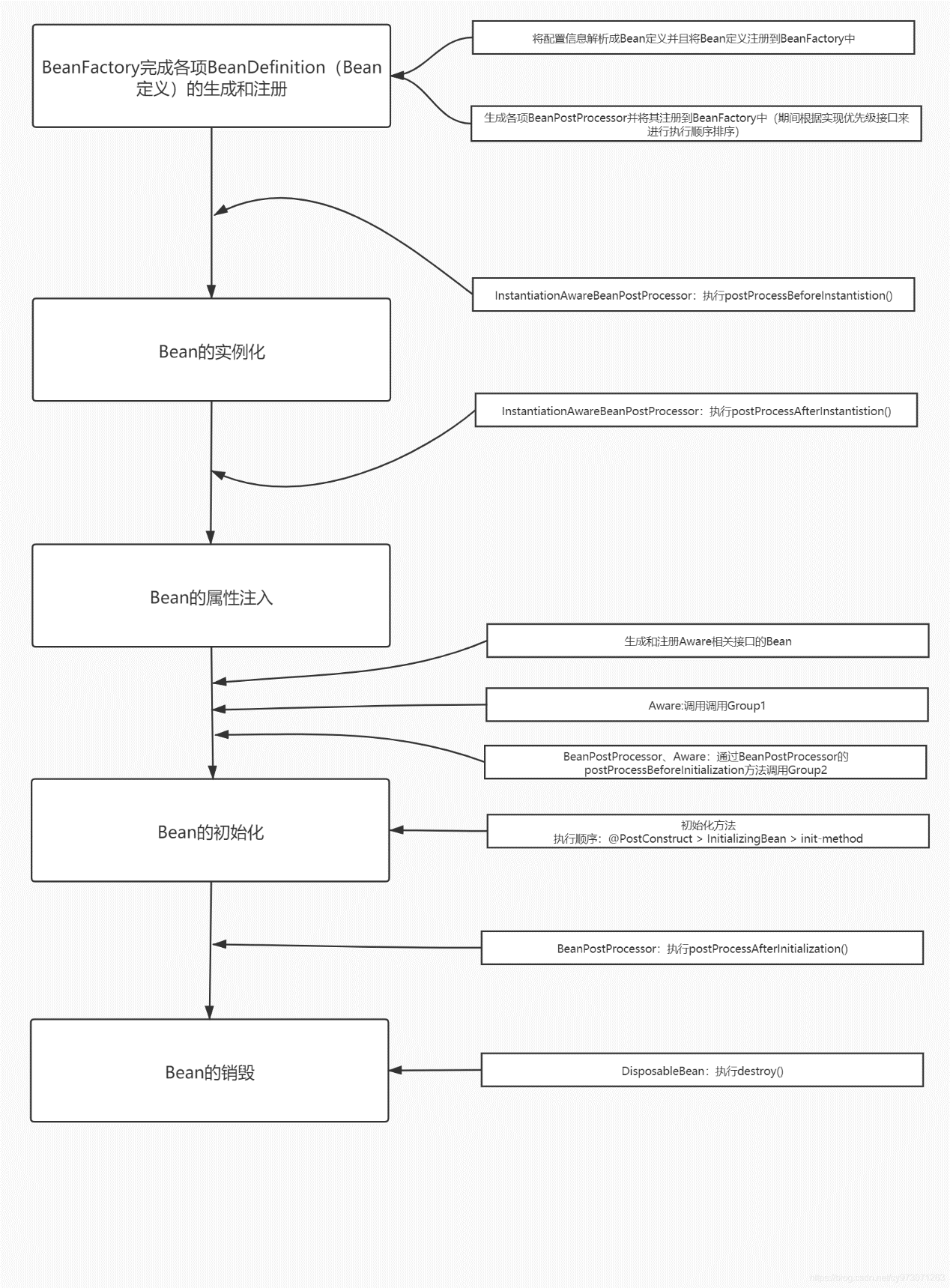
*\* @see java.lang.Integer#MAX\_VALUE*

*\*/*

int LOWEST\_PRECEDENCE = Integer.MAX\_VALUE;

PriorityOrdered、Ordered接口作为Spring整个框架通用的排序接口，在Spring中应用广泛，也是非常重要的接口。

**Bean的生命周期流程图**

[](https://img-blog.csdnimg.cn/20200313092801142.jpg?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L2N5OTczMDcxMjYz,size_16,color_FFFFFF,t_70)

**总结**

Spring Bean的生命周期分为**四个阶段**和**多个扩展点**。扩展点又可以分为**影响多个Bean**和**影响单个Bean**。整理如下：

**四个阶段**

* 实例化 Instantiation
* 属性赋值 Populate
* 初始化 Initialization
* 销毁 Destruction

**多个扩展点**

* 影响多个Bean
  + BeanPostProcessor
  + InstantiationAwareBeanPostProcessor
* 影响单个Bean
  + Aware
    - Aware Group1
      * BeanNameAware
      * BeanClassLoaderAware
      * BeanFactoryAware
    - Aware Group2
      * EnvironmentAware
      * EmbeddedValueResolverAware
      * ApplicationContextAware(ResourceLoaderAware\ApplicationEventPublisherAware\MessageSourceAware)
  + 生命周期
    - InitializingBean
    - DisposableBean

**哪些是重要的bean生命周期方法？ 你能重载它们吗？**

有两个重要的bean 生命周期方法，第一个是setup ， 它是在容器加载bean的时候被调用。第二个方法是 teardown 它是在容器卸载类的时候被调用。

bean 标签有两个重要的属性（init-method和destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct和@PreDestroy）。

**什么是Spring的内部bean？什么是Spring inner beans？**

在Spring框架中，当一个bean仅被用作另一个bean的属性时，它能被声明为一个内部bean。内部bean可以用setter注入“属性”和构造方法注入“构造参数”的方式来实现，内部bean通常是匿名的，它们的Scope一般是prototype。

**在 Spring中如何注入一个java集合？**

Spring提供以下几种集合的配置元素：

类型用于注入一列值，允许有相同的值。

类型用于注入一组值，不允许有相同的值。

类型用于注入一组键值对，键和值都可以为任意类型。

类型用于注入一组键值对，键和值都只能为String类型。

**什么是bean装配？**

装配，或bean 装配是指在Spring 容器中把bean组装到一起，前提是容器需要知道bean的依赖关系，如何通过依赖注入来把它们装配到一起。

**什么是bean的自动装配？**

在Spring框架中，在配置文件中设定bean的依赖关系是一个很好的机制，Spring 容器能够自动装配相互合作的bean，这意味着容器不需要和配置，能通过Bean工厂自动处理bean之间的协作。这意味着 Spring可以通过向Bean Factory中注入的方式自动搞定bean之间的依赖关系。自动装配可以设置在每个bean上，也可以设定在特定的bean上。

**解释不同方式的自动装配，spring 自动装配 bean 有哪些方式？**

在spring中，对象无需自己查找或创建与其关联的其他对象，由容器负责把需要相互协作的对象引用赋予各个对象，使用autowire来配置自动装载模式。

在Spring框架xml配置中共有5种自动装配：

* no：默认的方式是不进行自动装配的，通过手工设置ref属性来进行装配bean。
* byName：通过bean的名称进行自动装配，如果一个bean的 property 与另一bean 的name 相同，就进行自动装配。
* byType：通过参数的数据类型进行自动装配。
* constructor：利用构造函数进行装配，并且构造函数的参数通过byType进行装配。
* autodetect：自动探测，如果有构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。

**使用@Autowired注解自动装配的过程是怎样的？**

使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在Spring配置文件进行配置，<context:annotation-config />。

在启动spring IoC时，容器自动装载了一个AutowiredAnnotationBeanPostProcessor后置处理器，当容器扫描到@Autowied、@Resource或@Inject时，就会在IoC容器自动查找需要的bean，并装配给该对象的属性。在使用@Autowired时，首先在容器中查询对应类型的bean：

* 如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据；
* 如果查询的结果不止一个，那么@Autowired会根据名称来查找；
* 如果上述查找的结果为空，那么会抛出异常。解决方法时，使用required=false。

**自动装配有哪些局限性？**

自动装配的局限性是：

**重写**：你仍需用 和 配置来定义依赖，意味着总要重写自动装配。

**基本数据类型**：你不能自动装配简单的属性，如基本数据类型，String字符串，和类。

**模糊特性**：自动装配不如显式装配精确，如果有可能，建议使用显式装配。

**你可以在Spring中注入一个null 和一个空字符串吗？**

可以。

**问题： FactoryBean 和 BeanFactory有什么区别？**

**简要的答案：**

BeanFactory 是 Bean 的工厂， ApplicationContext 的父类，IOC 容器的核心，负责生产和管理 Bean 对象。

FactoryBean 是 Bean，可以通过实现 FactoryBean 接口定制实例化 Bean 的逻辑，通过代理一个Bean对象，对方法前后做一些操作。

**具体的介绍：**

**（1） BeanFactory 是ioc容器的底层实现接口，是ApplicationContext 顶级接口**

spring不允许我们直接操作 BeanFactory bean工厂，所以为我们提供了ApplicationContext 这个接口 此接口集成BeanFactory 接口，ApplicationContext包含BeanFactory的所有功能,同时还进行更多的扩展。

BeanFactory 接口又衍生出以下接口，其中我们经常用到的是ApplicationContext 接口

**ApplicationContext 继承图**

[](https://img-blog.csdnimg.cn/20190619231304280.png)

**ConfiguableApplicationContext** 中添加了一些方法：

... 其他省略

*//刷新ioc容器上下文*

void refresh() throws BeansException, IllegalStateException;

*// 关闭此应用程序上下文，释放所有资源并锁定，销毁所有缓存的单例bean。*

@Override

void close();

*//确定此应用程序上下文是否处于活动状态，即，是否至少刷新一次且尚未关闭。*

boolean isActive();

... 其他省略

主要作用在ioc容器进行相应的刷新，关闭等操作！

FileSystemXmlApplicationContext 和ClassPathXmlApplicationContext 是用来读取xml文件创建bean对象

ClassPathXmlApplicationContext ： 读取类路径下xml 创建bean

FileSystemXmlApplicationContext ：读取文件系统下xml创建bean

AnnotationConfigApplicationContext 主要是注解开发获取ioc中的bean实例

**（2） FactoryBean 是spirng提供的工厂bean的一个接口**

FactoryBean 接口提供三个方法，用来创建对象，  
FactoryBean 具体返回的对象是由getObject 方法决定的。

\*/

public interface FactoryBean<T> {

*//创建的具体bean对象的类型*

@Nullable

T getObject() throws Exception;

*//工厂bean 具体创建具体对象是由此getObject()方法来返回的*

@Nullable

Class<?> getObjectType();

*//是否单例*

default boolean isSingleton() {

return true;

}

}

创建一个FactoryBean 用来生产User对象

@Component

public class FactoryBeanTest implements FactoryBean<User> {

*//创建的具体bean对象的类型*

@Override

public Class<?> getObjectType() {

return User.class;

}

//是否单例

@Override

public boolean isSingleton() {

return true;

}

*//工厂bean 具体创建具体对象是由此getObject()方法来返回的*

@Override

public User getObject() throws Exception {

return new User();

}

}

**Junit测试**

@RunWith(SpringRunner.class)

@SpringBootTest(classes = {FactoryBeanTest.class})

@WebAppConfiguration

public class SpringBootDemoApplicationTests {

@Autowired

private ApplicationContext applicationContext;

@Test

public void tesst() {

FactoryBeanTest bean1 = applicationContext.getBean(FactoryBeanTest.class);

try {

User object = bean1.getObject();

System.out.println(object==object);

System.out.println(object);

} catch (Exception e) {

e.printStackTrace();

}

}

}

**结果**

true

User [id=null, name=null, age=0]

**简单的总结：**

BeanFactory是个bean 工厂，是一个工厂类(接口)， 它负责生产和管理bean的一个工厂

是ioc 容器最底层的接口，是个ioc容器，是spring用来管理和装配普通bean的ioc容器（这些bean成为普通bean）。

FactoryBean是个bean，在IOC容器的基础上给Bean的实现加上了一个简单工厂模式和装饰模式，是一个可以生产对象和装饰对象的工厂bean，由spring管理后，生产的对象是由getObject()方法决定的（从容器中获取到的对象不是

“ FactoryBeanTest ” 对象）。

**高频面试题：Spring 如何解决循环依赖？**

在关于Spring的面试中，我们经常会被问到一个问题：Spring是如何解决循环依赖的问题的。

这个问题算是关于Spring的一个高频面试题，因为如果不刻意研读，相信即使读过源码，面试者也不一定能够一下子思考出个中奥秘。

本文主要针对这个问题，从源码的角度对其实现原理进行讲解。

**循环依赖的简单例子**

比如几个Bean之间的互相引用：

甚至自己“循环”依赖自己：

**原型(Prototype)的场景是不支持循环依赖的**

先说明前提：原型(Prototype)的场景是不支持循环依赖的. 单例的场景才能存在循环依赖

原型(Prototype)的场景通常会走到[**AbstractBeanFactory**](http://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247493383&idx=2&sn=883c40b743d496e48dc2f99a083ec5e4&chksm=eb506231dc27eb272437a3dfd2b9a0ae91414c69a83c7e3fe22362cdef909e39122929af9884&scene=21#wechat_redirect)类中下面的判断，抛出异常。

if (isPrototypeCurrentlyInCreation(beanName)) {

throw new BeanCurrentlyInCreationException(beanName);

}

原因很好理解，创建新的A时，发现要注入原型字段B，又创建新的B发现要注入原型字段A...

这就套娃了, Spring就先抛出了BeanCurrentlyInCreationException

什么是原型(Prototype)的场景？

通过如下方式，可以将该类的bean设置为原型模式

@Service

@Scope("prototype")

public class MyReportExporter extends AbstractReportExporter{

...

}

在Spring中，@Service默认都是单例的。用了私有全局变量，若不想影响下次请求，就需要用到原型模式，即@Scope(“prototype”)

所谓单例，就是Spring的IOC机制只创建该类的一个实例，每次请求，都会用这同一个实例进行处理，因此若存在全局变量，本次请求的值肯定会影响下一次请求时该变量的值。  
原型模式，指的是每次调用时，会重新创建该类的一个实例，比较类似于我们自己自己new的对象实例。

**具体例子：循环依赖的代码片段**

我们先看看当时出问题的代码片段：

@Service

publicclass TestService1 {

@Autowired

private TestService2 testService2;

@Async

public void test1() {

}

}

@Service

publicclass TestService2 {

@Autowired

private TestService1 testService1;

public void test2() {

}

}

这两段代码中定义了两个Service类：**TestService1**和**TestService2**，在TestService1中注入了TestService2的实例，同时在TestService2中注入了TestService1的实例，这里构成了**循环依赖**。

只不过，这不是普通的循环依赖，因为TestService1的test1方法上加了一个**@Async**注解。

大家猜猜程序启动后运行结果会怎样？

org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'testService1': Bean with name 'testService1' has been injected into other beans [testService2] in its raw version as part of a circular reference, but has eventually been wrapped. This means that said other beans do not use the final version of the bean. This is often the result of over-eager type matching - consider using 'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.

报错了。。。原因是出现了循环依赖。

**「不科学呀，spring不是号称能解决循环依赖问题吗，怎么还会出现？」**

如果把上面的代码稍微调整一下：

@Service

publicclass TestService1 {

@Autowired

private TestService2 testService2;

public void test1() {

}

}

把TestService1的test1方法上的**@Async**注解去掉，**TestService1**和**TestService2**都需要注入对方的实例，同样构成了循环依赖。

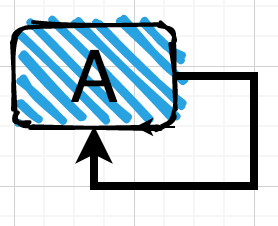
但是重新启动项目，发现它能够正常运行。这又是为什么？

带着这两个问题，让我们一起开始spring循环依赖的探秘之旅。

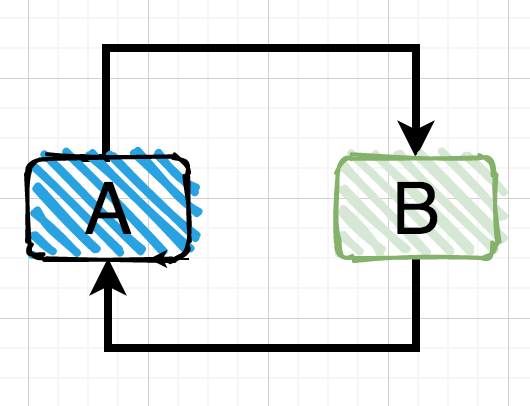
**什么是循环依赖？**

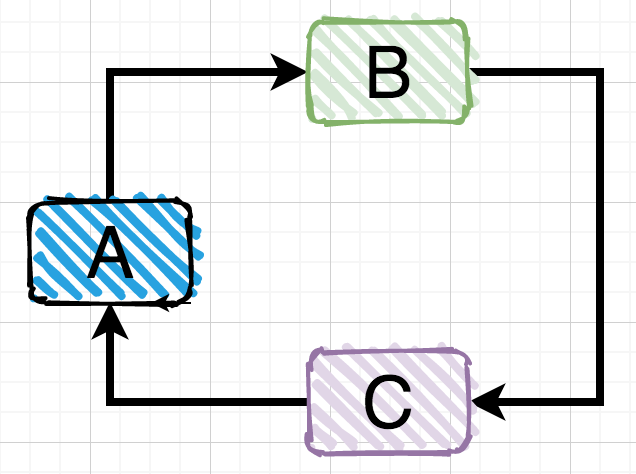
循环依赖：说白是一个或多个对象实例之间存在直接或间接的依赖关系，这种依赖关系构成了构成一个环形调用。

第一种情况：自己依赖自己的直接依赖

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYWIxTWlhY3FHOGJXQWd3Vm1nUjJUeFJkZkdROXJYcGplYlJQUU1ZcDZUV3BjQ1AySlNuWWFqbncvNjQw?x-oss-process=image/format,png)

第二种情况：两个对象之间的直接依赖

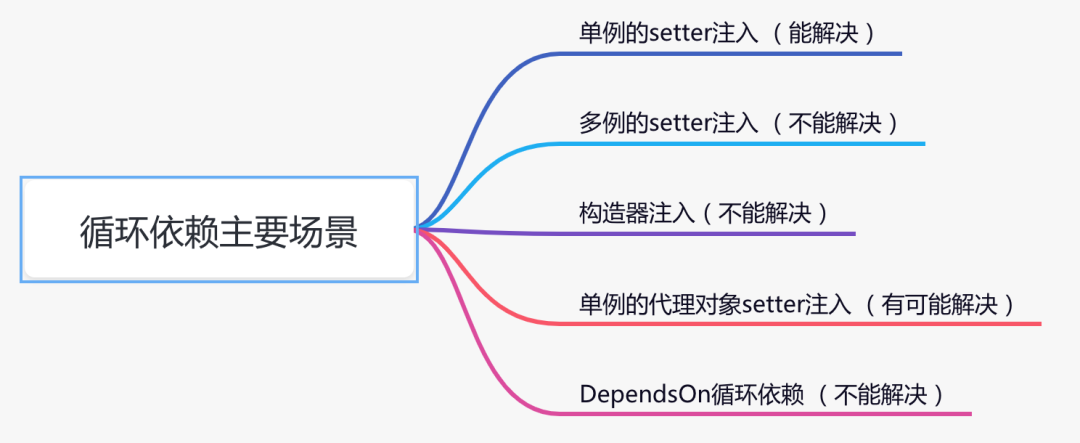
[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYXZheHJoUTYzU0huaWFxOEdSaDRmVWlibENpYWNLV1ViV01zemQ5bENvelpib3hhc01waWI0TmJ4elEvNjQw?x-oss-process=image/format,png)

第三种情况：多个对象之间的间接依赖[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYWxJenVjeXdaaWFEOUg1emljcnlEb3l3WmV2eEp4R0EwSnZzcndZN3RGaEY3SEExRW43WTBDZ1dBLzY0MA?x-oss-process=image/format,png)

前面两种情况的直接循环依赖比较直观，非常好识别，但是第三种间接循环依赖的情况有时候因为业务代码调用层级很深，不容易识别出来。

**循环依赖的N种场景**

spring中出现循环依赖主要有以下场景：

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYVBJSnl2a2s4YjdsNzRsRVVWQjZiMVRrZmZYMkp2ZTlkb0ppYTYxSVhoWTR5T2ZNeGliWE9pYzBBUS82NDA?x-oss-process=image/format,png)

**场景1：单例的setter注入**

这种注入方式应该是spring用的最多的，代码如下：

@Service

publicclass TestService1 {

@Autowired

private TestService2 testService2;

public void test1() {

}

}

@Service

publicclass TestService2 {

@Autowired

private TestService1 testService1;

public void test2() {

}

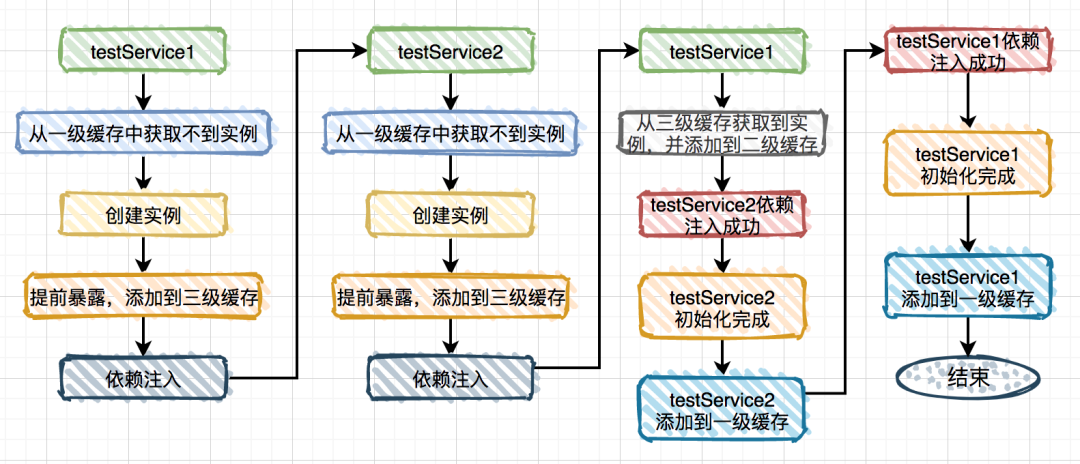
}

这是一个经典的循环依赖，但是它能正常运行，得益于spring的内部机制，让我们根本无法感知它有问题，因为spring默默帮我们解决了。

spring内部有三级缓存：

* singletonObjects 一级缓存，用于保存实例化、注入、初始化完成的bean实例
* earlySingletonObjects 二级缓存，用于保存实例化完成的bean实例
* singletonFactories 三级缓存，用于保存bean创建工厂，以便于后面扩展有机会创建代理对象。

下面用一张图告诉你，spring是如何解决循环依赖的：

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYW1zV2RaUVg2Y0k2SlFQYTNQN2ljb2VwaWJEWG9GRWhDNzFqdWliREFyS2JsRU1jb0JUUkxSUE1DZy82NDA?x-oss-process=image/format,png)

​

细心的朋友可能会发现在这种场景中**第二级缓存**作用不大。

**那么问题来了，为什么要用第二级缓存呢？**

试想一下，如果出现以下这种情况，我们要如何处理？

@Service

publicclass TestService1 {

@Autowired

private TestService2 testService2;

@Autowired

private TestService3 testService3;

public void test1() {

}

}

@Service

publicclass TestService2 {

@Autowired

private TestService1 testService1;

public void test2() {

}

}

@Service

publicclass TestService3 {

@Autowired

private TestService1 testService1;

public void test3() {

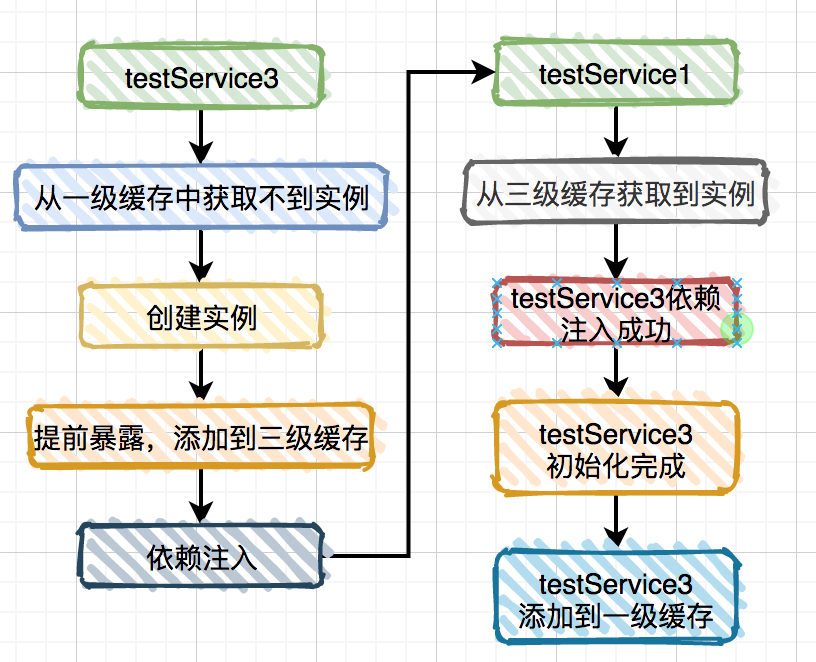
}

}

TestService1依赖于TestService2和TestService3，而TestService2依赖于TestService1，同时TestService3也依赖于TestService1。

按照上图的流程可以把TestService1注入到TestService2，并且TestService1的实例是从第三级缓存中获取的。

假设不用第二级缓存，TestService1注入到TestService3的流程如图：

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYWlhZ3I1eU9tYUpZbk5yU3JnM0lJc2hLSHNYN2Z0RXBsQ2hTZmNZMUNKR1ljYWFIQ0pYVkNRblEvNjQw?x-oss-process=image/format,png)

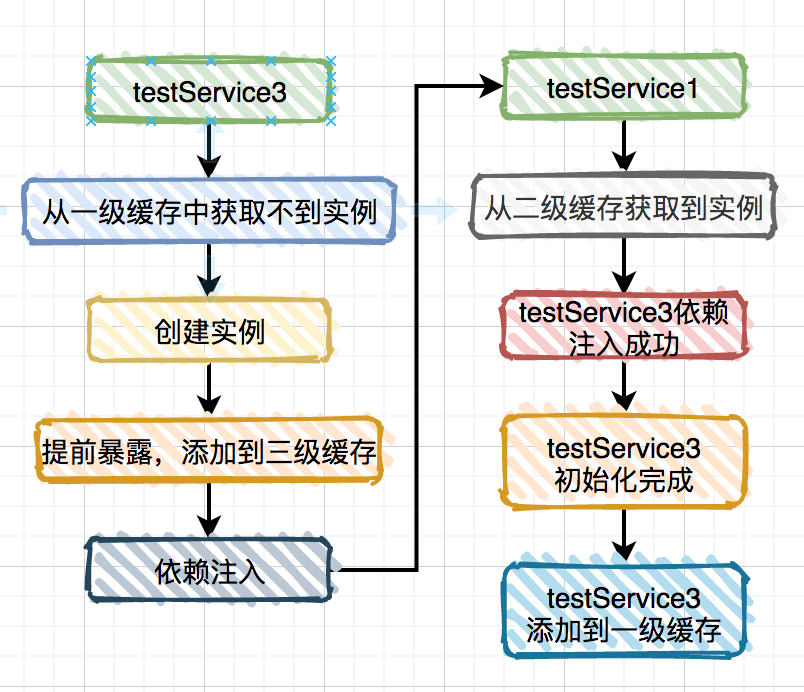
​

TestService1注入到TestService3又需要从第三级缓存中获取实例，而第三级缓存里保存的并非真正的实例对象，而是**ObjectFactory**对象。

说白了，两次从三级缓存中获取都是**ObjectFactory**对象，而通过它创建的实例对象每次可能都不一样的。

这样不是有问题？

为了解决这个问题，spring引入的第二级缓存。前一个图其实TestService1对象的实例已经被添加到第二级缓存中了，而在TestService1注入到TestService3时，只用从第二级缓存中获取该对象即可。

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYWRNWEZzYWd3aWNyRTJra0g4anN5QUR6UFdJMnV6WmNrVGxmV1lLd09kQzBFZ0RIQzVaRU5pYjRRLzY0MA?x-oss-process=image/format,png)

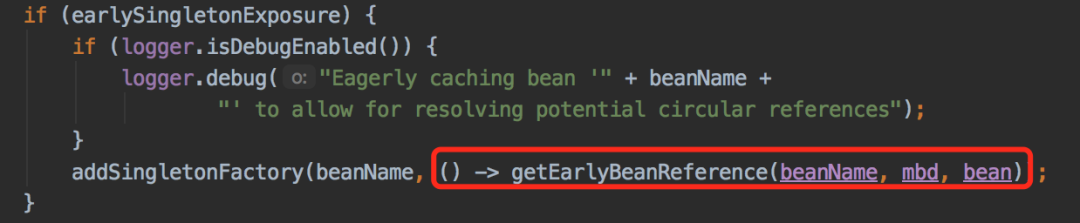
​

还有个问题，第三级缓存中为什么要添加**ObjectFactory**对象，直接保存实例对象不行吗？

答：不行，因为假如你想对添加到三级缓存中的实例对象进行增强，直接用实例对象是行不通的。

针对这种场景spring是怎么做的呢？

答案就在**AbstractAutowireCapableBeanFactory**类**doCreateBean**方法的这段代码中：

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYVQ4VHJFc2xEa1NmSVFjalNaS1VGZWVSQmR6OU90Tk96Qk5wZmRLNllMWk5sV3V2b0YwS2FGQS82NDA?x-oss-process=image/format,png)它定义了一个匿名内部类，通过**getEarlyBeanReference**方法获取代理对象，其实底层是通过**AbstractAutoProxyCreator**类的**getEarlyBeanReference**生成代理对象。

**场景二多例的setter注入**

这种注入方法偶然会有，特别是在多线程的场景下，具体代码如下：

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

@Service

publicclass TestService1 {

@Autowired

private TestService2 testService2;

public void test1() {

}

}

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

@Service

publicclass TestService2 {

@Autowired

private TestService1 testService1;

public void test2() {

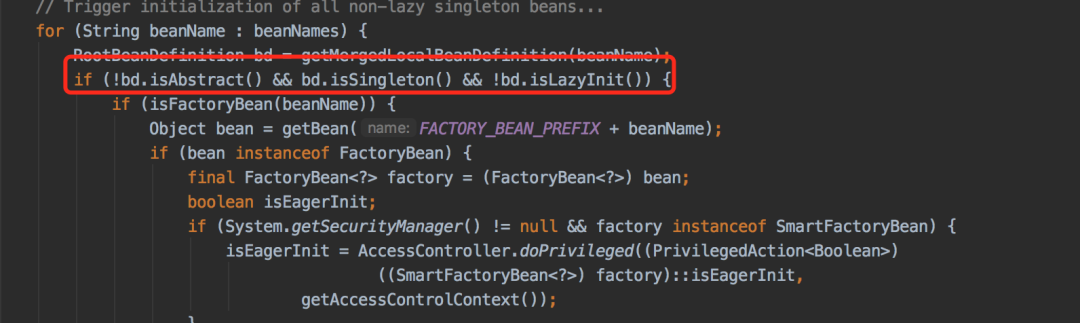
}

}

很多人说这种情况spring容器启动会报错，其实是不对的，我非常负责任的告诉你程序能够正常启动。

**为什么呢？**

其实在**AbstractApplicationContext**类的**refresh**方法中告诉了我们答案，它会调用**finishBeanFactoryInitialization**方法，该方法的作用是为了spring容器启动的时候提前初始化一些bean。该方法的内部又调用了**preInstantiateSingletons**方法

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYVRRQkNnMzZRUVhCaWJhTzdYM2tYS2liQ085UU9lMjhpY0k4OXFueWMwU3pxcDJQbklzNVdDNHk3dy82NDA?x-oss-process=image/format,png)标红的地方明显能够看出：非抽象、单例 并且非懒加载的类才能被提前初始bean。

而多例即**SCOPE\_PROTOTYPE**类型的类，非单例，不会被提前初始化bean，所以程序能够正常启动。

如何让他提前初始化bean呢？

只需要再定义一个单例的类，在它里面注入TestService1

@Service

publicclass TestService3 {

@Autowired

private TestService1 testService1;

}

重新启动程序，执行结果：

Requested bean is currently in creation: Is there an unresolvable circular reference?

果然出现了循环依赖。

注意：这种循环依赖问题是无法解决的，因为它没有用缓存，每次都会生成一个新对象。

**场景三：构造器注入**

这种注入方式现在其实用的已经非常少了，但是我们还是有必要了解一下，看看如下代码：

@Service

publicclass TestService1 {

public TestService1(TestService2 testService2) {

}

}

@Service

publicclass TestService2 {

public TestService2(TestService1 testService1) {

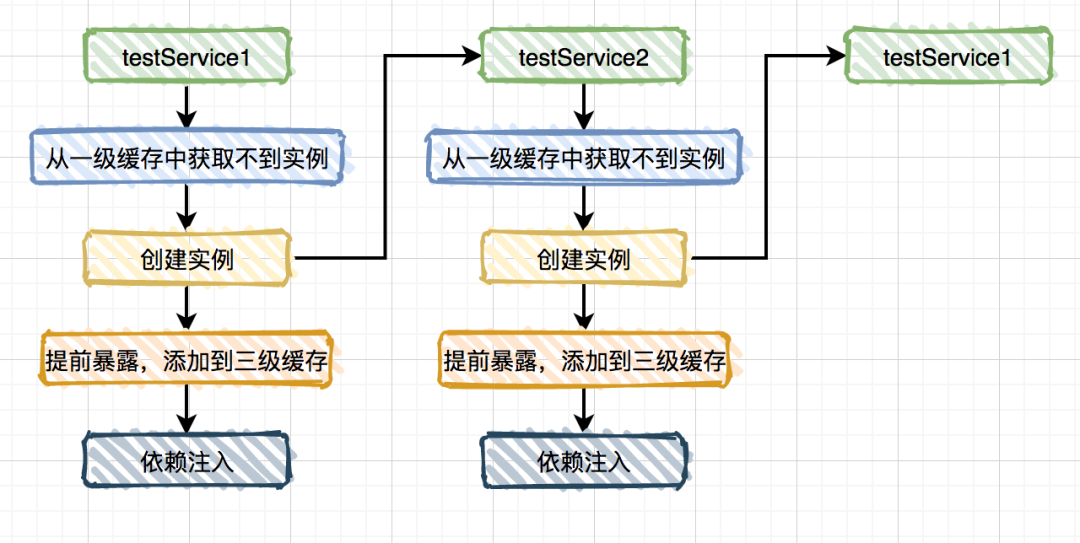
}

}

运行结果：

Requested bean is currently in creation: Is there an unresolvable circular reference?

出现了循环依赖，为什么呢？

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYXQwZW4wUEhCTDZXQXFIS3JieFUzOG54WnVaQ0RXeFZURTlpYXpjS1lBNDlGVHNlQ1lJbER4YkEvNjQw?x-oss-process=image/format,png)

从图中的流程看出构造器注入没能添加到三级缓存，也没有使用缓存，所以也无法解决循环依赖问题。

**场景四：单例的代理对象setter注入**

这种注入方式其实也比较常用，比如平时使用：**@Async**注解的场景，会通过**AOP**自动生成代理对象。

我那位同事的问题也是这种情况。

@Service

publicclass TestService1 {

@Autowired

private TestService2 testService2;

@Async

public void test1() {

}

}

@Service

publicclass TestService2 {

@Autowired

private TestService1 testService1;

public void test2() {

}

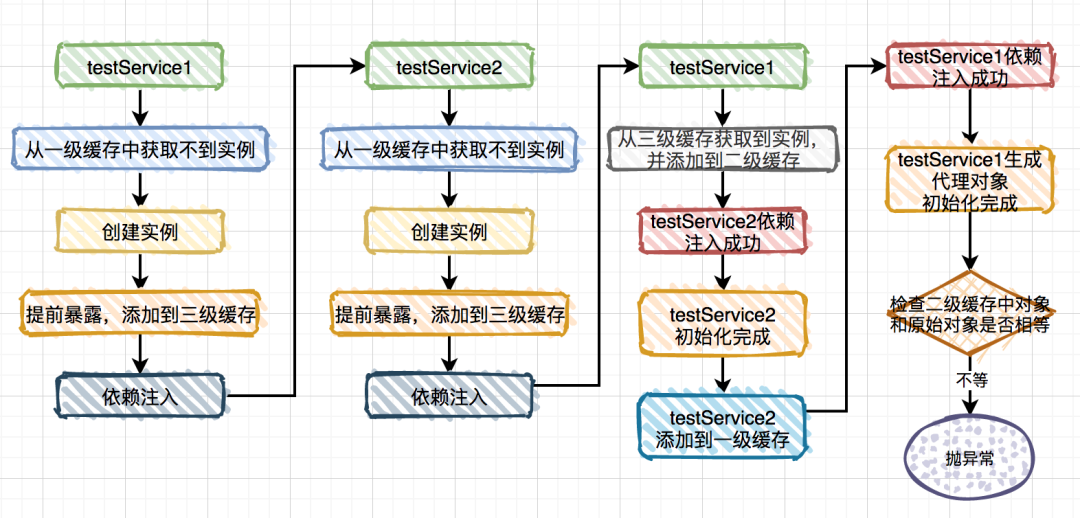
}

从前面得知程序启动会报错，出现了循环依赖：

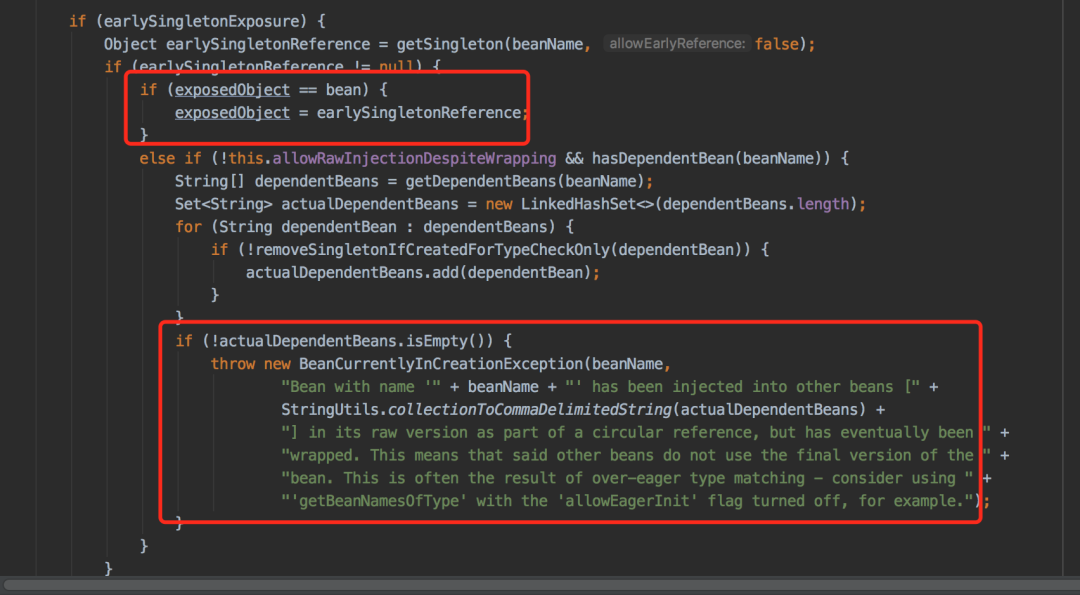
org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'testService1': Bean with name 'testService1' has been injected into other beans [testService2] in its raw version as part of a circular reference, but has eventually been wrapped. This means that said other beans do not use the final version of the bean. This is often the result of over-eager type matching - consider using 'getBeanNamesOfType' with the 'allowEagerInit' flag turned off, for example.

为什么会循环依赖呢？

答案就在下面这张图中：

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERkdMN0lJRUJKVktmQUg2cm14WTR4MmZwQmlhN3pxYXBXMVE4WkY2MDE2NjFJcTJOSXdIWWpyUnY2NFRHbnVRWnk4eUYzaEIwOFJMTWcvNjQw?x-oss-process=image/format,png)

说白了，bean初始化完成之后，后面还有一步去检查：第二级缓存 和 原始对象 是否相等。由于它对前面流程来说无关紧要，所以前面的流程图中省略了，但是在这里是关键点，我们重点说说：

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYWxqaWMzQWFueTJpYWdCUVdSUmtvelVoejMzWG9KN1dyT3loUWdkY3FqY0hFZWZpYzlXY2puZTZGQS82NDA?x-oss-process=image/format,png)

那位同事的问题正好是走到这段代码，发现第二级缓存 和 原始对象不相等，所以抛出了循环依赖的异常。

如果这时候把TestService1改个名字，改成：TestService6，其他的都不变。

@Service

publicclass TestService6 {

@Autowired

private TestService2 testService2;

@Async

public void test1() {

}

}

再重新启动一下程序，神奇般的好了。

what？ 这又是为什么？

这就要从spring的bean加载顺序说起了，默认情况下，spring是按照文件完整路径递归查找的，按路径+文件名排序，排在前面的先加载。所以TestService1比TestService2先加载，而改了文件名称之后，TestService2比TestService6先加载。

为什么TestService2比TestService6先加载就没问题呢？

答案在下面这张图中：

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERkdMN0lJRUJKVktmQUg2cm14WTR4MnJKUnRYQkFUb3dkeHJCak5NV2pRam9TaWNCOEo2MmI0WW1qdlIweVNtQjRUR0NEVWljZHdqYkZRLzY0MA?x-oss-process=image/format,png)

这种情况testService6中其实第二级缓存是空的，不需要跟原始对象判断，所以不会抛出循环依赖。

**场景5：DependsOn循环依赖**

还有一种有些特殊的场景，比如我们需要在实例化Bean A之前，先实例化Bean B，这个时候就可以使用**@DependsOn**注解。

@DependsOn(value = "testService2")

@Service

publicclass TestService1 {

@Autowired

private TestService2 testService2;

public void test1() {

}

}

@DependsOn(value = "testService1")

@Service

publicclass TestService2 {

@Autowired

private TestService1 testService1;

public void test2() {

}

}

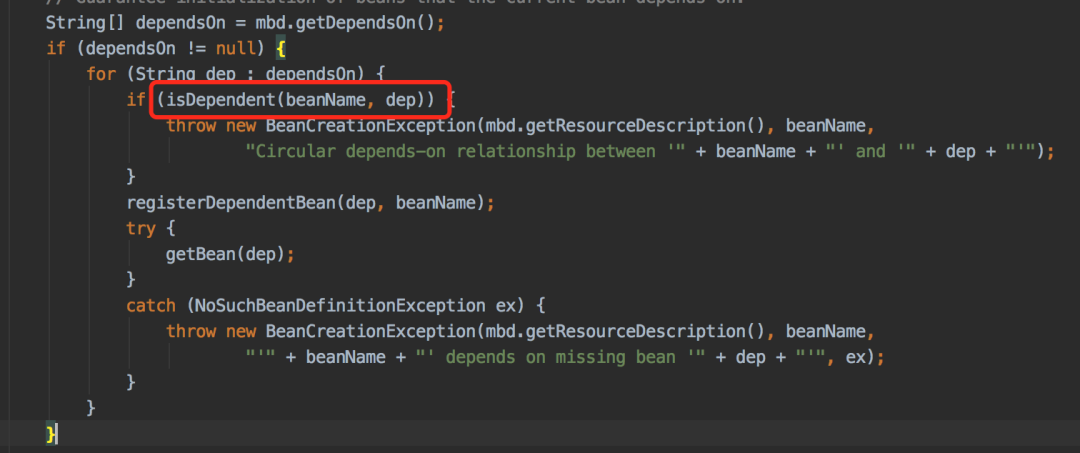
程序启动之后，执行结果：

Circular depends-on relationship between 'testService2' and 'testService1'

这个例子中本来如果TestService1和TestService2都没有加**@DependsOn**注解是没问题的，反而加了这个注解会出现循环依赖问题。

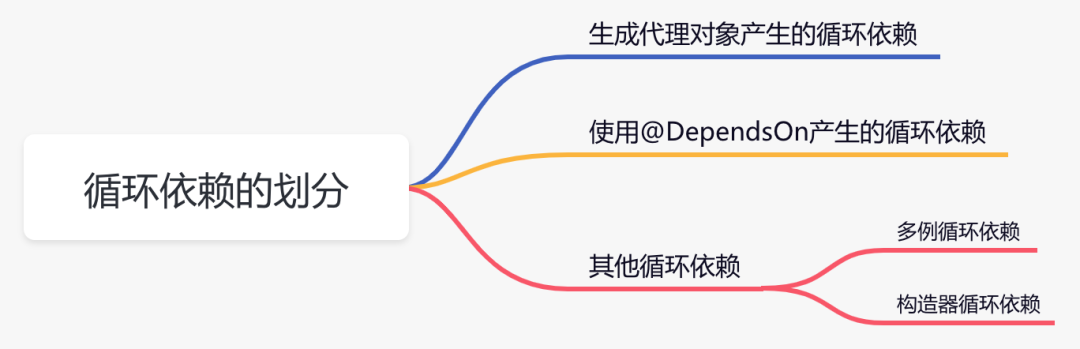
这又是为什么？

答案在**AbstractBeanFactory**类的**doGetBean**方法的这段代码中：

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYXRqSmpTeE1rb2gyUnlWQThLRkpSaWFlZHY0aGliam9pY0ttZnlTZGlhRXM3NGRBcG9CaWI3VDRDZGZRLzY0MA?x-oss-process=image/format,png)它会检查dependsOn的实例有没有循环依赖，如果有循环依赖则抛异常。

**总体策略：出现循环依赖如何解决？**

项目中如果出现循环依赖问题，说明是spring默认无法解决的循环依赖，要看项目的打印日志，属于哪种循环依赖。目前包含下面几种情况：

[](https://imgconvert.csdnimg.cn/aHR0cHM6Ly9tbWJpei5xcGljLmNuL21tYml6X3BuZy91TDM3MTI4MW9ERjI0dDNGanJCOFhlYng3V2drQXhkYXNjemxxblJHUmxFM3Ewd1p6WlRlMk9ZZXpUVjI2SU1USks4VzdkZFZpY2VhS0xMak1NMFY5OEEvNjQw?x-oss-process=image/format,png)

**生成代理对象产生的循环依赖 的解决方案：**

这类循环依赖问题解决方法很多，主要有：

1. 使用**@Lazy**注解，延迟加载
2. 使用**@DependsOn**注解，指定加载先后关系
3. 修改文件名称，改变循环依赖类的加载顺序

**使用@DependsOn产生的循环依赖 的解决方案：**

这类循环依赖问题要找到**@DependsOn**注解循环依赖的地方，迫使它不循环依赖就可以解决问题。

**多例循环依赖 的解决方案：**

这类循环依赖问题可以通过把bean改成单例的解决。

**构造器循环依赖 的解决方案：**

这类循环依赖问题可以通过使用**@Lazy**注解解决

**回答提要：**

按照上面的方式回答， 起码120分。

但是答案太复杂， 如果上面的答案，记不住，就用下面的答案吧，至少也是100分。

**问题： Spring是怎么解决循环依赖的？**

首先，Spring 解决循环依赖有两个前提条件：

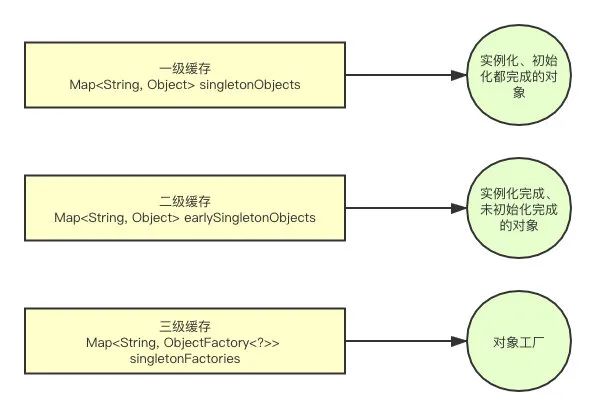
1. 不全是构造器方式的循环依赖
2. 必须是单例

基于上面的问题，我们知道Bean的生命周期，本质上解决循环依赖的问题就是三级缓存，通过三级缓存提前拿到未初始化的对象。

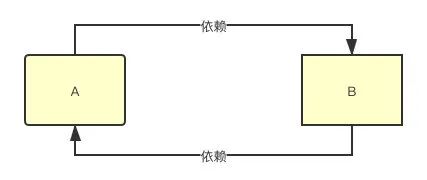
第一级缓存：用来保存实例化、初始化都完成的对象

第二级缓存：用来保存实例化完成，但是未初始化完成的对象

第三级缓存：用来保存一个对象工厂，提供一个匿名内部类，用于创建二级缓存中的对象

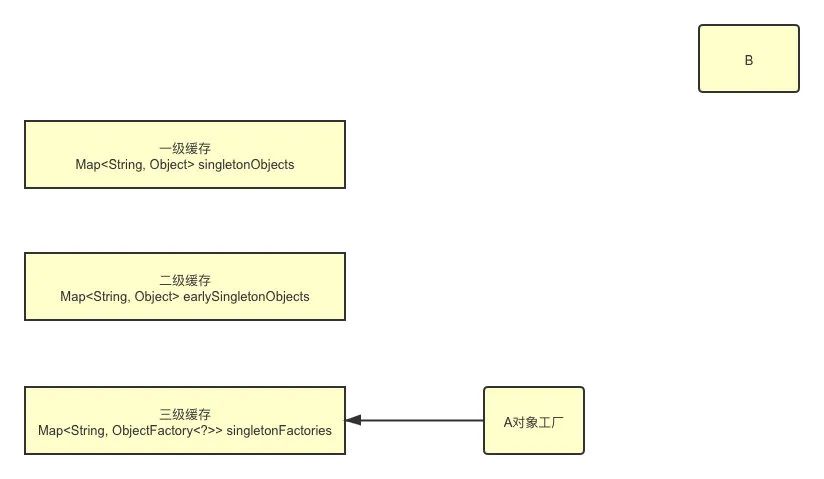
[](https://img-blog.csdnimg.cn/img_convert/284af8d0e1eb22235df36e03742d937d.png)

假设一个简单的循环依赖场景，A、B互相依赖。

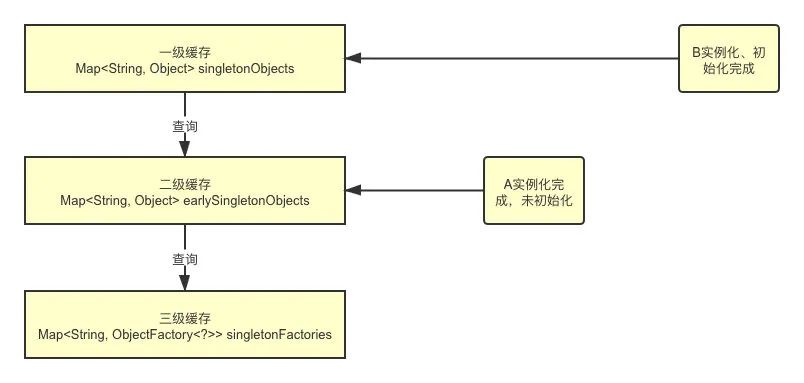
[](https://img-blog.csdnimg.cn/img_convert/1d93a76bea9c09330a248b431fa377df.png)

A对象的创建过程：

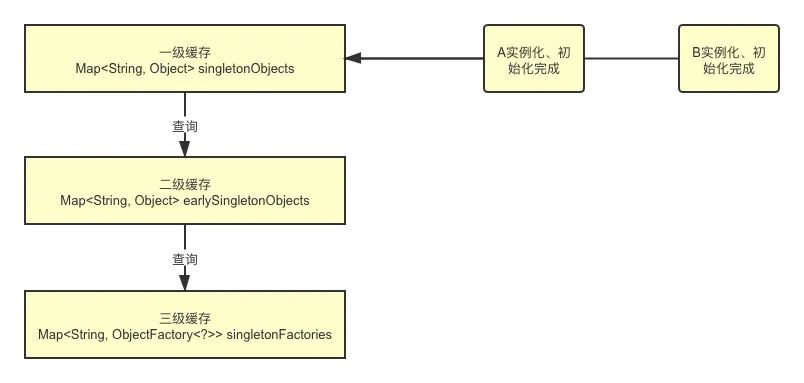
1. 创建对象A，实例化的时候把A对象工厂放入三级缓存

[](https://img-blog.csdnimg.cn/img_convert/c6449f69e0c43e4d99363bf10480f943.png)

1. A注入属性时，发现依赖B，转而去实例化B
2. 同样创建对象B，注入属性时发现依赖A，一次从一级到三级缓存查询A，从三级缓存通过对象工厂拿到A，把A放入二级缓存，同时删除三级缓存中的A，此时，B已经实例化并且初始化完成，把B放入一级缓存。

[](https://img-blog.csdnimg.cn/img_convert/bbc4051fe3fc4715faaf9f4cdcebc0cf.png)

1. 接着继续创建A，顺利从一级缓存拿到实例化且初始化完成的B对象，A对象创建也完成，删除二级缓存中的A，同时把A放入一级缓存
2. 最后，一级缓存中保存着实例化、初始化都完成的A、B对象

[](https://img-blog.csdnimg.cn/img_convert/aa0eab11db07cf276a821aa62ce442d3.png)

因此，由于把实例化和初始化的流程分开了，所以如果都是用构造器的话，就没法分离这个操作，所以都是构造器的话就无法解决循环依赖的问题了。

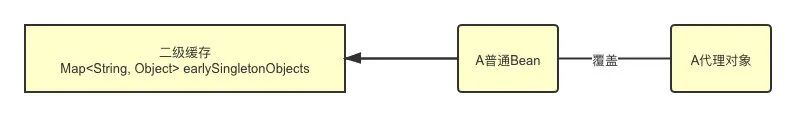
**问题:为什么要三级缓存？二级不行吗？**

不可以，主要是为了生成代理对象。

因为三级缓存中放的是生成具体对象的匿名内部类，他可以生成代理对象，也可以是普通的实例对象。

使用三级缓存主要是为了保证不管什么时候使用的都是一个对象。

假设只有二级缓存的情况，往二级缓存中放的显示一个普通的Bean对象，**BeanPostProcessor**去生成代理对象之后，覆盖掉二级缓存中的普通Bean对象，那么多线程环境下可能取到的对象就不一致了。

[](https://img-blog.csdnimg.cn/img_convert/b86c0d9f9e38b3d54c896b2edccd5140.png)

**Spring注解（8题目）**

**什么是基于Java的Spring注解配置? 给一些注解的例子**

基于Java的配置，允许你在少量的Java注解的帮助下，进行你的大部分Spring配置而非通过XML文件。

以@Configuration 注解为例，它用来标记类可以当做一个bean的定义，被Spring IOC容器使用。

另一个例子是@Bean注解，它表示此方法将要返回一个对象，作为一个bean注册进Spring应用上下文。

@Configuration

public class StudentConfig {

@Bean

public StudentBean myStudent() {

return new StudentBean();

}

}

**怎样开启注解装配？**

注解装配在默认情况下是不开启的，为了使用注解装配，我们必须在Spring配置文件中配置 **<context:annotation-config/>**元素。

**@Component, @Controller, @Repository, @Service 有何区别？**

@Component：这将 java 类标记为 bean。它是任何 Spring 管理组件的通用构造型。spring 的组件扫描机制现在可以将其拾取并将其拉入应用程序环境中。

@Controller：这将一个类标记为 Spring Web MVC 控制器。标有它的 Bean 会自动导入到 IoC 容器中。

@Service：此注解是组件注解的特化。它不会对 @Component 注解提供任何其他行为。您可以在服务层类中使用 @Service 而不是 @Component，因为它以更好的方式指定了意图。

@Repository：这个注解是具有类似用途和功能的 @Component 注解的特化。它为 DAO 提供了额外的好处。它将 DAO 导入 IoC 容器，并使未经检查的异常有资格转换为 Spring DataAccessException。

**@Required 注解有什么作用**

这个注解表明bean的属性必须在配置的时候设置，通过一个bean定义的显式的属性值或通过自动装配，若@Required注解的bean属性未被设置，容器将抛出BeanInitializationException。示例：

public class Employee {

private String name;

@Required

public void setName(String name){

this.name=name;

}

public string getName(){

return name;

}

}

**@Autowired 注解有什么作用**

@Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。@Autowired 注解提供了更细粒度的控制，包括在何处以及如何完成自动装配。它的用法和@Required一样，修饰setter方法、构造器、属性或者具有任意名称和/或多个参数的PN方法。

public class Employee {

private String name;

@Autowired

public void setName(String name) {

this.name=name;

}

public string getName(){

return name;

}

}

**@Autowired和@Resource之间的区别**

@Autowired可用于：构造函数、成员变量、Setter方法

@Autowired和@Resource之间的区别

* @Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。
* @Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。

**@Qualifier 注解有什么作用**

当您创建多个相同类型的 bean 并希望仅使用属性装配其中一个 bean 时，您可以使用@Qualifier 注解和 @Autowired 通过指定应该装配哪个确切的 bean 来消除歧义。

**@RequestMapping 注解有什么用？**

@RequestMapping 注解用于将特定 HTTP 请求方法映射到将处理相应请求的控制器中的特定类/方法。此注释可应用于两个级别：

* 类级别：映射请求的 URL
* 方法级别：映射 URL 以及 HTTP 请求方法

**Spring数据访问（14）**

**解释对象/关系映射集成模块**

Spring 通过提供ORM模块，支持我们在直接JDBC之上使用一个对象/关系映射映射(ORM)工具，Spring 支持集成主流的ORM框架，如Hiberate，JDO和 iBATIS，JPA，TopLink，JDO，OJB 。Spring的事务管理同样支持以上所有ORM框架及JDBC。

**在Spring框架中如何更有效地使用JDBC？**

使用Spring JDBC 框架，资源管理和错误处理的代价都会被减轻。所以开发者只需写statements 和 queries从数据存取数据，JDBC也可以在Spring框架提供的模板类的帮助下更有效地被使用，这个模板叫JdbcTemplate

**解释JDBC抽象和DAO模块**

通过使用JDBC抽象和DAO模块，保证数据库代码的简洁，并能避免数据库资源错误关闭导致的问题，它在各种不同的数据库的错误信息之上，提供了一个统一的异常访问层。它还利用Spring的AOP 模块给Spring应用中的对象提供事务管理服务。

**spring DAO 有什么用？**

Spring DAO（数据访问对象） 使得 JDBC，Hibernate 或 JDO 这样的数据访问技术更容易以一种统一的方式工作。这使得用户容易在持久性技术之间切换。它还允许您在编写代码时，无需考虑捕获每种技术不同的异常。

**spring JDBC API 中存在哪些类？**

JdbcTemplate

SimpleJdbcTemplate

NamedParameterJdbcTemplate

SimpleJdbcInsert

SimpleJdbcCall

**JdbcTemplate是什么**

JdbcTemplate 类提供了很多便利的方法解决诸如把数据库数据转变成基本数据类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据错误处理。

**使用Spring通过什么方式访问Hibernate？使用 Spring 访问 Hibernate 的方法有哪些？**

在Spring中有两种方式访问Hibernate：

* 使用 Hibernate 模板和回调进行控制反转
* 扩展 HibernateDAOSupport 并应用 AOP 拦截器节点

**如何通过HibernateDaoSupport将Spring和Hibernate结合起来？**

用Spring的 SessionFactory 调用 LocalSessionFactory。集成过程分三步：

* 配置the Hibernate SessionFactory
* 继承HibernateDaoSupport实现一个DAO
* 在AOP支持的事务中装配

**Spring支持的事务管理类型， spring 事务实现方式有哪些？**

Spring支持两种类型的事务管理：

**编程式事务管理**：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。

**声明式事务管理**：这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

**Spring事务的实现方式和实现原理**

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过binlog或者redo log实现的。

**说一下Spring的事务传播行为**

spring事务的传播行为说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为。

① PROPAGATION\_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

② PROPAGATION\_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。

③ PROPAGATION\_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ PROPAGATION\_REQUIRES\_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ PROPAGATION\_NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

⑥ PROPAGATION\_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ PROPAGATION\_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。

**说一下 spring 的事务隔离？**

spring 有五大隔离级别，默认值为 ISOLATION\_DEFAULT（使用数据库的设置），其他四个隔离级别和数据库的隔离级别一致：

1. ISOLATION\_DEFAULT：用底层数据库的设置隔离级别，数据库设置的是什么我就用什么；
2. ISOLATION\_READ\_UNCOMMITTED：未提交读，最低隔离级别、事务未提交前，就可被其他事务读取（会出现幻读、脏读、不可重复读）；
3. ISOLATION\_READ\_COMMITTED：提交读，一个事务提交后才能被其他事务读取到（会造成幻读、不可重复读），SQL server 的默认级别；
4. ISOLATION\_REPEATABLE\_READ：可重复读，保证多次读取同一个数据时，其值都和事务开始时候的内容是一致，禁止读取到别的事务未提交的数据（会造成幻读），MySQL 的默认级别；
5. ISOLATION\_SERIALIZABLE：序列化，代价最高最可靠的隔离级别，该隔离级别能防止脏读、不可重复读、幻读。

**脏读** ：表示一个事务能够读取另一个事务中还未提交的数据。比如，某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。

**不可重复读** ：是指在一个事务内，多次读同一数据。

**幻读** ：指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了。

**Spring框架的事务管理有哪些优点？**

* 为不同的事务API 如 JTA，JDBC，Hibernate，JPA 和JDO，提供一个不变的编程模式。
* 为编程式事务管理提供了一套简单的API而不是一些复杂的事务API
* 支持声明式事务管理。
* 和Spring各种数据访问抽象层很好得集成。

**你更倾向用那种事务管理类型？**

大多数Spring框架的用户选择声明式事务管理，因为它对应用代码的影响最小，因此更符合一个无侵入的轻量级容器的思想。声明式事务管理要优于编程式事务管理，虽然比编程式事务管理（这种方式允许你通过代码控制事务）少了一点灵活性。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

**Spring面向切面编程(AOP)（13）**

**什么是AOP**

OOP(Object-Oriented Programming)面向对象编程，允许开发者定义纵向的关系，但并适用于定义横向的关系，导致了大量代码的重复，而不利于各个模块的重用。

AOP(Aspect-Oriented Programming)，一般称为面向切面编程，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理等。

**Spring AOP and AspectJ AOP 有什么区别？AOP 有哪些实现方式？**

AOP实现的关键在于 代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

（1）AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。

（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

**JDK动态代理和CGLIB动态代理的区别**

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

* JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例, 生成目标类的代理对象。
* 如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

InvocationHandler 的 invoke(Object proxy,Method method,Object[] args)：proxy是最终生成的代理实例; method 是被代理目标实例的某个具体方法; args 是被代理目标实例某个方法的具体入参, 在方法反射调用时使用。

**如何理解 Spring 中的代理？**

将 Advice 应用于目标对象后创建的对象称为代理。在客户端对象的情况下，目标对象和代理对象是相同的。

Advice + Target Object = Proxy

**解释一下Spring AOP里面的几个名词**

（1）切面（Aspect）：切面是通知和切点的结合。通知和切点共同定义了切面的全部内容。 在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @AspectJ 注解来实现。

（2）连接点（Join point）：指方法，在Spring AOP中，一个连接点 总是 代表一个方法的执行。 应用可能有数以千计的时机应用通知。这些时机被称为连接点。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

（3）通知（Advice）：在AOP术语中，切面的工作被称为通知。

（4）切入点（Pointcut）：切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。

（5）引入（Introduction）：引入允许我们向现有类添加新方法或属性。

（6）目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。它通常是一个代理对象。也有人把它叫做 被通知（adviced） 对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。

（7）织入（Weaving）：织入是把切面应用到目标对象并创建新的代理对象的过程。在目标对象的生命周期里有多少个点可以进行织入：

* 编译期：切面在目标类编译时被织入。AspectJ的织入编译器是以这种方式织入切面的。
* 类加载期：切面在目标类加载到JVM时被织入。需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ5的加载时织入就支持以这种方式织入切面。
* 运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。SpringAOP就是以这种方式织入切面。

**Spring在运行时通知对象**

通过在代理类中包裹切面，Spring在运行期把切面织入到Spring管理的bean中。代理封装了目标类，并拦截被通知方法的调用，再把调用转发给真正的目标bean。当代理拦截到方法调用时，在调用目标bean方法之前，会执行切面逻辑。

直到应用需要被代理的bean时，Spring才创建代理对象。如果使用的是ApplicationContext的话，在ApplicationContext从BeanFactory中加载所有bean的时候，Spring才会创建被代理的对象。因为Spring运行时才创建代理对象，所以我们不需要特殊的编译器来织入SpringAOP的切面。

**Spring只支持方法级别的连接点**

因为Spring基于动态代理，所以Spring只支持方法连接点。Spring缺少对字段连接点的支持，而且它不支持构造器连接点。方法之外的连接点拦截功能，我们可以利用Aspect来补充。

**在Spring AOP 中，关注点和横切关注的区别是什么？在 spring aop 中 concern 和 cross-cutting concern 的不同之处**

关注点（concern）是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。

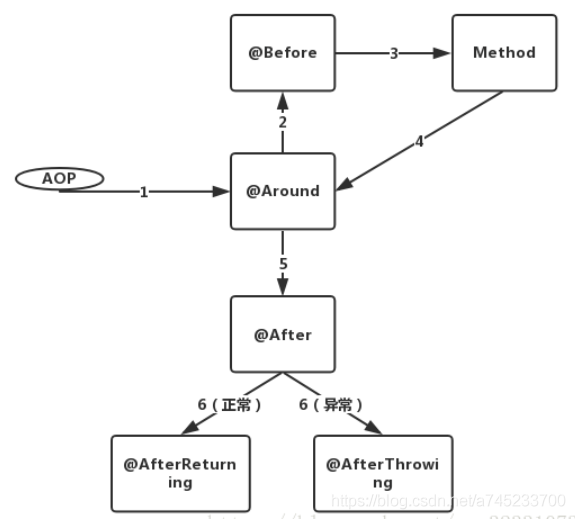
横切关注点（cross-cutting concern）是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

**Spring通知有哪些类型？**

在AOP术语中，切面的工作被称为通知，实际上是程序执行时要通过SpringAOP框架触发的代码段。

Spring切面可以应用5种类型的通知：

1. 前置通知（Before）：在目标方法被调用之前调用通知功能；
2. 后置通知（After）：在目标方法完成之后调用通知，此时不会关心方法的输出是什么；
3. 返回通知（After-returning ）：在目标方法成功执行之后调用通知；
4. 异常通知（After-throwing）：在目标方法抛出异常后调用通知；
5. 环绕通知（Around）：通知包裹了被通知的方法，在被通知的方法调用之前和调用之后执行自定义的行为。

[](https://img-blog.csdnimg.cn/20201207005911882.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L2E3NDUyMzM3MDA=,size_16,color_FFFFFF,t_70)

同一个aspect，不同advice的执行顺序：

①没有异常情况下的执行顺序：

around before advice  
before advice  
target method 执行  
around after advice  
after advice  
afterReturning

②有异常情况下的执行顺序：

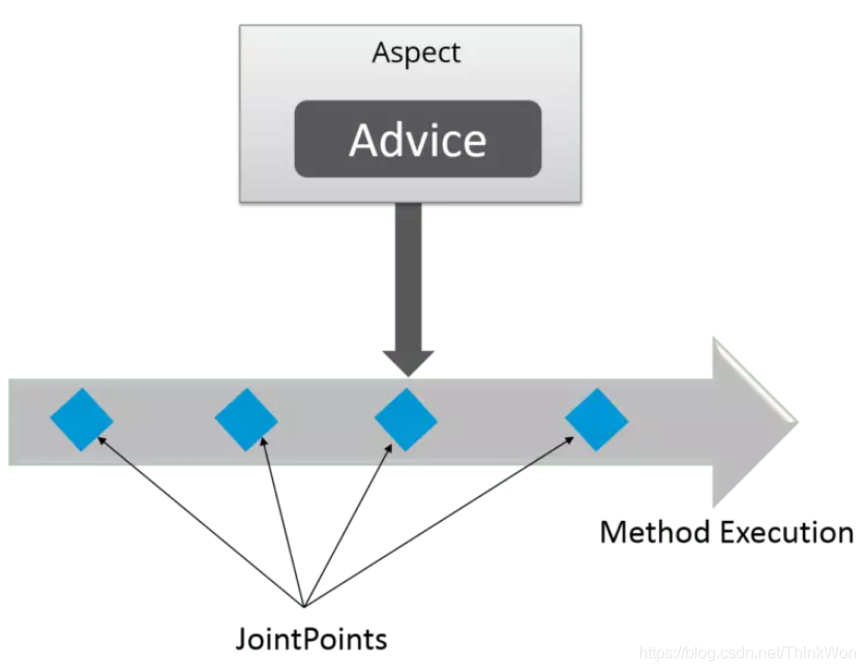
around before advice  
before advice  
target method 执行  
around after advice  
after advice  
afterThrowing:异常发生  
java.lang.RuntimeException: 异常发生

**什么是切面 Aspect？**

aspect 由 pointcount 和 advice 组成，切面是通知和切点的结合。 它既包含了横切逻辑的定义, 也包括了连接点的定义. Spring AOP 就是负责实施切面的框架, 它将切面所定义的横切逻辑编织到切面所指定的连接点中.  
AOP 的工作重心在于如何将增强编织目标对象的连接点上, 这里包含两个工作:

* 如何通过 pointcut 和 advice 定位到特定的 joinpoint 上
* 如何在 advice 中编写切面代码.

可以简单地认为, 使用 @Aspect 注解的类就是切面.

[](https://img-blog.csdnimg.cn/2020021212264438.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

**解释基于XML Schema方式的切面实现**

在这种情况下，切面由常规类以及基于XML的配置实现。

**解释基于注解的切面实现**

在这种情况下(基于@AspectJ的实现)，涉及到的切面声明的风格与带有java5标注的普通java类一致。

**有几种不同类型的自动代理？**

BeanNameAutoProxyCreator

DefaultAdvisorAutoProxyCreator

Metadata autoproxying

**Spring MVC面试题 专题部分**

**什么是Spring MVC？简单介绍下你对Spring MVC的理解？**

Spring MVC是一个基于Java的实现了MVC设计模式的请求驱动类型的轻量级Web框架，通过把模型-视图-控制器分离，将web层进行职责解耦，把复杂的web应用分成逻辑清晰的几部分，简化开发，减少出错，方便组内开发人员之间的配合。

**Spring MVC的优点**

（1）可以支持各种视图技术,而不仅仅局限于JSP；

（2）与Spring框架集成（如IoC容器、AOP等）；

（3）清晰的角色分配：前端控制器(dispatcherServlet) , 请求到处理器映射（handlerMapping), 处理器适配器（HandlerAdapter), 视图解析器（ViewResolver）。

（4） 支持各种请求资源的映射策略。

**核心组件**

**Spring MVC的主要组件？**

（1）前端控制器 DispatcherServlet（不需要程序员开发）

作用：接收请求、响应结果，相当于转发器，有了DispatcherServlet 就减少了其它组件之间的耦合度。

（2）处理器映射器HandlerMapping（不需要程序员开发）

作用：根据请求的URL来查找Handler

（3）处理器适配器HandlerAdapter

注意：在编写Handler的时候要按照HandlerAdapter要求的规则去编写，这样适配器HandlerAdapter才可以正确的去执行Handler。

（4）处理器Handler（需要程序员开发）

（5）视图解析器 ViewResolver（不需要程序员开发）

作用：进行视图的解析，根据视图逻辑名解析成真正的视图（view）

（6）视图View（需要程序员开发jsp）

View是一个接口， 它的实现类支持不同的视图类型（jsp，freemarker，pdf等等）

**什么是DispatcherServlet**

Spring的MVC框架是围绕DispatcherServlet来设计的，它用来处理所有的HTTP请求和响应。

**什么是Spring MVC框架的控制器？**

控制器提供一个访问应用程序的行为，此行为通常通过服务接口实现。控制器解析用户输入并将其转换为一个由视图呈现给用户的模型。Spring用一个非常抽象的方式实现了一个控制层，允许用户创建多种用途的控制器。

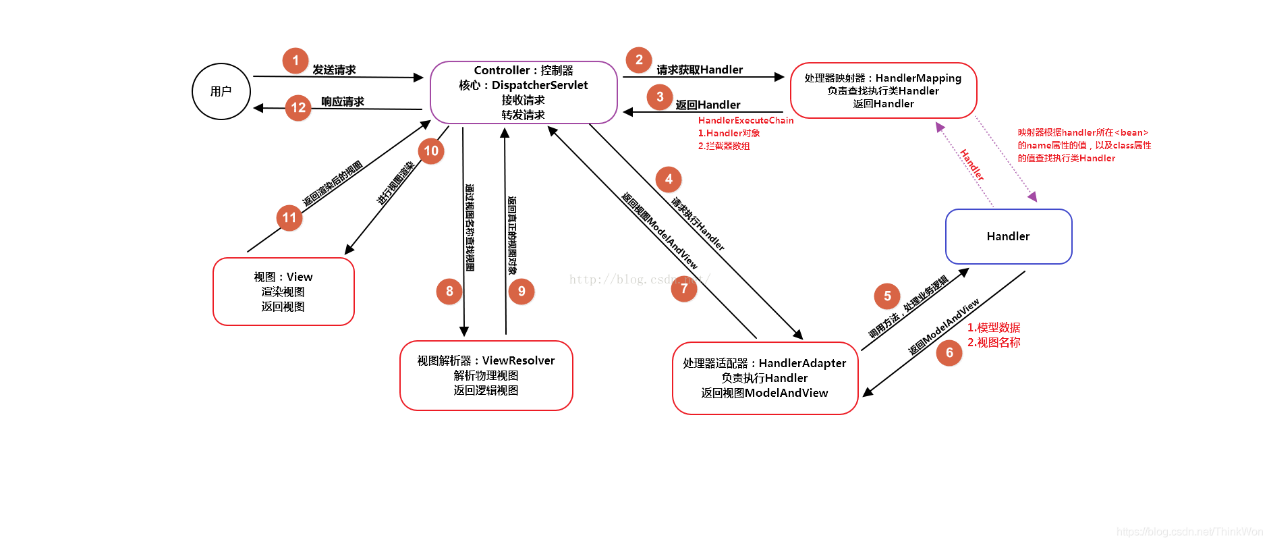
**Spring MVC的控制器是不是单例模式,如果是,有什么问题,怎么解决？**

答：是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

**工作原理**

**请描述Spring MVC的工作流程？描述一下 DispatcherServlet 的工作流程？**

（1）用户发送请求至前端控制器DispatcherServlet；  
（2） DispatcherServlet收到请求后，调用HandlerMapping处理器映射器，请求获取Handle；  
（3）处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet；  
（4）DispatcherServlet 调用 HandlerAdapter处理器适配器；  
（5）HandlerAdapter 经过适配调用 具体处理器(Handler，也叫后端控制器)；  
（6）Handler执行完成返回ModelAndView；  
（7）HandlerAdapter将Handler执行结果ModelAndView返回给DispatcherServlet；  
（8）DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行解析；  
（9）ViewResolver解析后返回具体View；  
（10）DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）  
（11）DispatcherServlet响应用户。

[](https://img-blog.csdnimg.cn/20200208211439106.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

**MVC框架**

**MVC是什么？MVC设计模式的好处有哪些**

mvc是一种设计模式（设计模式就是日常开发中编写代码的一种好的方法和经验的总结）。模型（model）-视图（view）-控制器（controller），三层架构的设计模式。用于实现前端页面的展现与后端业务数据处理的分离。

mvc设计模式的好处

1.分层设计，实现了业务系统各个组件之间的解耦，有利于业务系统的可扩展性，可维护性。

2.有利于系统的并行开发，提升开发效率。

**常用注解**

**注解原理是什么**

注解本质是一个继承了Annotation的特殊接口，其具体实现类是Java运行时生成的动态代理类。我们通过反射获取注解时，返回的是Java运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用AnnotationInvocationHandler的invoke方法。该方法会从memberValues这个Map中索引出对应的值。而memberValues的来源是Java常量池。

**Spring MVC常用的注解有哪些？**

@RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。

@RequestBody：注解实现接收http请求的json数据，将json转换为java对象。

@ResponseBody：注解实现将conreoller方法返回对象转化为json对象响应给客户。

**SpingMvc中的控制器的注解一般用哪个,有没有别的注解可以替代？**

答：一般用@Controller注解,也可以使用@RestController,@RestController注解相当于@ResponseBody ＋ @Controller,表示是表现层,除此之外，一般不用别的注解代替。

**@Controller注解的作用**

在Spring MVC 中，控制器Controller 负责处理由DispatcherServlet 分发的请求，它把用户请求的数据经过业务处理层处理之后封装成一个Model ，然后再把该Model 返回给对应的View 进行展示。在Spring MVC 中提供了一个非常简便的定义Controller 的方法，你无需继承特定的类或实现特定的接口，只需使用@Controller 标记一个类是Controller ，然后使用@RequestMapping 和@RequestParam 等一些注解用以定义URL 请求和Controller 方法之间的映射，这样的Controller 就能被外界访问到。此外Controller 不会直接依赖于HttpServletRequest 和HttpServletResponse 等HttpServlet 对象，它们可以通过Controller 的方法参数灵活的获取到。

@Controller 用于标记在一个类上，使用它标记的类就是一个Spring MVC Controller 对象。分发处理器将会扫描使用了该注解的类的方法，并检测该方法是否使用了@RequestMapping 注解。@Controller 只是定义了一个控制器类，而使用@RequestMapping 注解的方法才是真正处理请求的处理器。单单使用@Controller 标记在一个类上还不能真正意义上的说它就是Spring MVC 的一个控制器类，因为这个时候Spring 还不认识它。那么要如何做Spring 才能认识它呢？这个时候就需要我们把这个控制器类交给Spring 来管理。有两种方式：

* 在Spring MVC 的配置文件中定义MyController 的bean 对象。
* 在Spring MVC 的配置文件中告诉Spring 该到哪里去找标记为@Controller 的Controller 控制器。

**@RequestMapping注解的作用**

RequestMapping是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。

RequestMapping注解有六个属性，下面我们把她分成三类进行说明（下面有相应示例）。

**value， method**

value： 指定请求的实际地址，指定的地址可以是URI Template 模式（后面将会说明）；

method： 指定请求的method类型， GET、POST、PUT、DELETE等；

**consumes，produces**

consumes： 指定处理请求的提交内容类型（Content-Type），例如application/json, text/html;

produces: 指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回；

**params，headers**

params： 指定request中必须包含某些参数值是，才让该方法处理。

headers： 指定request中必须包含某些指定的header值，才能让该方法处理请求。

**@ResponseBody注解的作用**

作用： 该注解用于将Controller的方法返回的对象，通过适当的HttpMessageConverter转换为指定格式后，写入到Response对象的body数据区。

使用时机：返回的数据不是html标签的页面，而是其他某种格式的数据时（如json、xml等）使用；

**@PathVariable和@RequestParam的区别**

请求路径上有个id的变量值，可以通过@PathVariable来获取 @RequestMapping(value = “/page/{id}”, method = RequestMethod.GET)

@RequestParam用来获得静态的URL请求入参 spring注解时action里用到。

**其他**

**Spring MVC与Struts2区别**

相同点

都是基于mvc的表现层框架，都用于web项目的开发。

不同点

1.前端控制器不一样。Spring MVC的前端控制器是servlet：DispatcherServlet。struts2的前端控制器是filter：StrutsPreparedAndExcutorFilter。

2.请求参数的接收方式不一样。Spring MVC是使用方法的形参接收请求的参数，基于方法的开发，线程安全，可以设计为单例或者多例的开发，推荐使用单例模式的开发（执行效率更高），默认就是单例开发模式。struts2是通过类的成员变量接收请求的参数，是基于类的开发，线程不安全，只能设计为多例的开发。

3.Struts采用值栈存储请求和响应的数据，通过OGNL存取数据，Spring MVC通过参数解析器是将request请求内容解析，并给方法形参赋值，将数据和视图封装成ModelAndView对象，最后又将ModelAndView中的模型数据通过reques域传输到页面。Jsp视图解析器默认使用jstl。

4.与spring整合不一样。Spring MVC是spring框架的一部分，不需要整合。在企业项目中，Spring MVC使用更多一些。

**Spring MVC怎么样设定重定向和转发的？**

（1）转发：在返回值前面加"forward:"，譬如"forward:user.do?name=method4"

（2）重定向：在返回值前面加"redirect:"，譬如"redirect:[http://www.baidu.com](http://www.baidu.com/)"

**Spring MVC怎么和AJAX相互调用的？**

通过Jackson框架就可以把Java里面的对象直接转化成Js可以识别的Json对象。具体步骤如下 ：

（1）加入Jackson.jar

（2）在配置文件中配置json的映射

（3）在接受Ajax方法里面可以直接返回Object,List等,但方法前面要加上@ResponseBody注解。

**如何解决POST请求中文乱码问题，GET的又如何处理呢？**

（1）解决post请求乱码问题：

在web.xml中配置一个CharacterEncodingFilter过滤器，设置成utf-8；

<filter>

<filter-name>CharacterEncodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>utf-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>CharacterEncodingFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

（2）get请求中文参数出现乱码解决方法有两个：

①修改tomcat配置文件添加编码与工程编码一致，如下：

<ConnectorURIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>

②另外一种方法对参数进行重新编码：

String userName = new String(request.getParamter(“userName”).getBytes(“ISO8859-1”),“utf-8”)

ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码。

**Spring MVC的异常处理？**

答：可以将异常抛给Spring框架，由Spring框架来处理；我们只需要配置简单的异常处理器，在异常处理器中添视图页面即可。

**如果在拦截请求中，我想拦截get方式提交的方法,怎么配置**

答：可以在@RequestMapping注解里面加上method=RequestMethod.GET。

**怎样在方法里面得到Request,或者Session？**

答：直接在方法的形参中声明request,Spring MVC就自动把request对象传入。

**如果想在拦截的方法里面得到从前台传入的参数,怎么得到？**

答：直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样。

**如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象？**

答：直接在方法中声明这个对象,Spring MVC就自动会把属性赋值到这个对象里面。

**Spring MVC中函数的返回值是什么？**

答：返回值可以有很多类型,有String, ModelAndView。ModelAndView类把视图和数据都合并的一起的，但一般用String比较好。

**Spring MVC用什么对象从后台向前台传递数据的？**

答：通过ModelMap对象,可以在这个对象里面调用put方法,把对象加到里面,前台就可以通过el表达式拿到。

**怎么样把ModelMap里面的数据放入Session里面？**

答：可以在类上面加上@SessionAttributes注解,里面包含的字符串就是要放入session里面的key。

**Spring MVC里面拦截器是怎么写的**

有两种写法,一种是实现HandlerInterceptor接口，另外一种是继承适配器类，接着在接口方法当中，实现处理逻辑；然后在Spring MVC的配置文件中配置拦截器即可：

*<!-- 配置Spring MVC的拦截器 -->*

<mvc:interceptors>

*<!-- 配置一个拦截器的Bean就可以了 默认是对所有请求都拦截 -->*

<bean id="myInterceptor" class="com.zwp.action.MyHandlerInterceptor"></bean>

*<!-- 只针对部分请求拦截 -->*

<mvc:interceptor>

<mvc:mapping path="/modelMap.do" />

<bean class="com.zwp.action.MyHandlerInterceptorAdapter" />

</mvc:interceptor>

</mvc:interceptors>

**介绍一下 WebApplicationContext**

WebApplicationContext 继承了ApplicationContext 并增加了一些WEB应用必备的特有功能，它不同于一般的ApplicationContext ，因为它能处理主题，并找到被关联的servlet。

**Tomcat 专题 部分**

**Tomcat是什么？**

Tomcat 服务器Apache软件基金会项目中的一个核心项目，是一个免费的开放源代码的Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问用户不是很多的场合下被普遍使用，是开发和调试JSP 程序的首选。

**Tomcat的缺省端口是多少，怎么修改**

1. 找到Tomcat目录下的conf文件夹
2. 进入conf文件夹里面找到server.xml文件
3. 打开server.xml文件
4. 在server.xml文件里面找到下列信息
5. 把Connector标签的8080端口改成你想要的端口

<Service name="Catalina">

<Connector port="8080" protocol="HTTP/1.1"

connectionTimeout="20000"

redirectPort="8443" />

**tomcat 有哪几种Connector 运行模式(优化)？**

下面，我们先大致了解Tomcat Connector的三种运行模式。

* **BIO：同步并阻塞** 一个线程处理一个请求。缺点：并发量高时，线程数较多，浪费资源。Tomcat7或以下，在Linux系统中默认使用这种方式。

**配制项**：protocol=”HTTP/1.1”

* NIO：同步非阻塞IO

利用Java的异步IO处理，可以通过少量的线程处理大量的请求，可以复用同一个线程处理多个connection(多路复用)。

Tomcat8在Linux系统中默认使用这种方式。

Tomcat7必须修改Connector配置来启动。

**配制项**：protocol=”org.apache.coyote.http11.Http11NioProtocol”

**备注**：我们常用的Jetty，Mina，ZooKeeper等都是基于java nio实现.

* APR：即Apache Portable Runtime，从操作系统层面解决io阻塞问题。**AIO方式，\*\*\*\*异步非阻塞IO**(Java NIO2又叫AIO) 主要与NIO的区别主要是操作系统的底层区别.可以做个比喻:比作快递，NIO就是网购后要自己到官网查下快递是否已经到了(可能是多次)，然后自己去取快递；AIO就是快递员送货上门了(不用关注快递进度)。

**配制项**：protocol=”org.apache.coyote.http11.Http11AprProtocol”

**备注**：需在本地服务器安装APR库。Tomcat7或Tomcat8在Win7或以上的系统中启动默认使用这种方式。Linux如果安装了apr和native，Tomcat直接启动就支持apr。

**Tomcat有几种部署方式？**

**在Tomcat中部署Web应用的方式主要有如下几种：**

1. 利用Tomcat的自动部署。

把web应用拷贝到webapps目录。Tomcat在启动时会加载目录下的应用，并将编译后的结果放入work目录下。

1. 使用Manager App控制台部署。

在tomcat主页点击“Manager App” 进入应用管理控制台，可以指定一个web应用的路径或war文件。

1. 修改conf/server.xml文件部署。

修改conf/server.xml文件，增加Context节点可以部署应用。

1. 增加自定义的Web部署文件。

在conf/Catalina/localhost/ 路径下增加 xyz.xml文件，内容是Context节点，可以部署应用。

**tomcat容器是如何创建servlet类实例？用到了什么原理？**

1. 当容器启动时，会读取在webapps目录下所有的web应用中的web.xml文件，然后对 **xml文件进行解析，并读取servlet注册信息**。然后，将每个应用中注册的servlet类都进行加载，并通过 **反射的方式实例化**。（有时候也是在第一次请求时实例化）
2. 在servlet注册时加上1如果为正数，则在一开始就实例化，如果不写或为负数，则第一次请求实例化。

**Tomcat工作模式**

Tomcat作为servlet容器，有三种工作模式：

* 1、独立的servlet容器，servlet容器是web服务器的一部分；
* 2、进程内的servlet容器，servlet容器是作为web服务器的插件和java容器的实现，web服务器插件在内部地址空间打开一个jvm使得java容器在内部得以运行。反应速度快但伸缩性不足；
* 3、进程外的servlet容器，servlet容器运行于web服务器之外的地址空间，并作为web服务器的插件和java容器实现的结合。反应时间不如进程内但伸缩性和稳定性比进程内优；

进入Tomcat的请求可以根据Tomcat的工作模式分为如下两类：

* Tomcat作为应用程序服务器：请求来自于前端的web服务器，这可能是Apache, IIS, Nginx等；
* Tomcat作为独立服务器：请求来自于web浏览器；

面试时问到Tomcat相关问题的几率并不高，正式因为如此，很多人忽略了对Tomcat相关技能的掌握，下面这一篇文章整理了Tomcat相关的系统架构，介绍了Server、Service、Connector、Container之间的关系，各个模块的功能，可以说把这几个掌握住了，Tomcat相关的面试题你就不会有任何问题了！另外，在面试的时候你还要有意识无意识的往Tomcat这个地方引，就比如说常见的Spring MVC的执行流程，一个URL的完整调用链路，这些相关的题目你是可以往Tomcat处理请求的这个过程去说的！掌握了Tomcat这些技能，面试官一定会佩服你的！

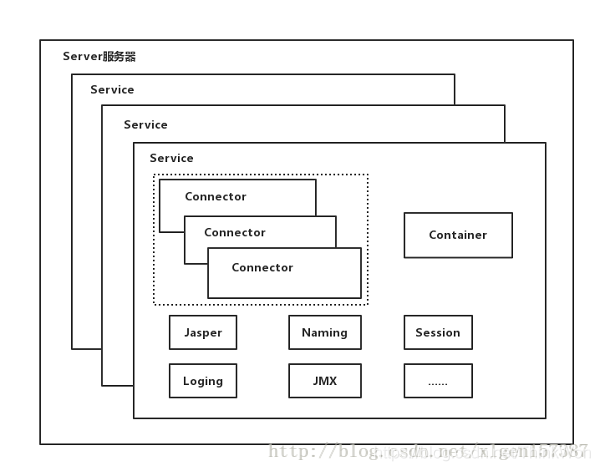
学了本章之后你应该明白的是：

* Server、Service、Connector、Container四大组件之间的关系和联系，以及他们的主要功能点；
* Tomcat执行的整体架构，请求是如何被一步步处理的；
* Engine、Host、Context、Wrapper相关的概念关系；
* Container是如何处理请求的；
* Tomcat用到的相关设计模式；

**Tomcat顶层架构**

俗话说，站在巨人的肩膀上看世界，一般学习的时候也是先总览一下整体，然后逐个部分个个击破，最后形成思路，了解具体细节，Tomcat的结构很复杂，但是 Tomcat 非常的模块化，找到了 Tomcat 最核心的模块，问题才可以游刃而解，了解了 Tomcat 的整体架构对以后深入了解 Tomcat 来说至关重要！

先上一张Tomcat的顶层结构图（图A），如下：

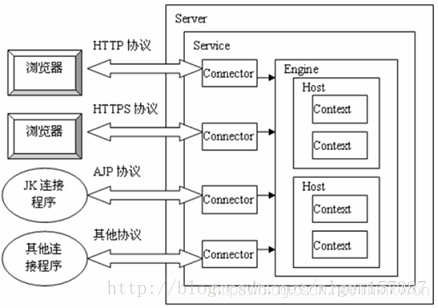
[](https://img-blog.csdnimg.cn/20191021215330153.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

Tomcat中最顶层的容器是Server，代表着整个服务器，从上图中可以看出，一个Server可以包含至少一个Service，即可以包含多个Service，用于具体提供服务。

Service主要包含两个部分：Connector和Container。从上图中可以看出 Tomcat 的心脏就是这两个组件，他们的作用如下：

* Connector用于处理连接相关的事情，并提供Socket与Request请求和Response响应相关的转化;
* Container用于封装和管理Servlet，以及具体处理Request请求；

一个Tomcat中只有一个Server，一个Server可以包含多个Service，一个Service只有一个Container，但是可以有多个Connectors，这是因为一个服务可以有多个连接，如同时提供Http和Https链接，也可以提供向相同协议不同端口的连接，示意图如下（Engine、Host、Context下面会说到）：

[](https://img-blog.csdnimg.cn/20191021215344811.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

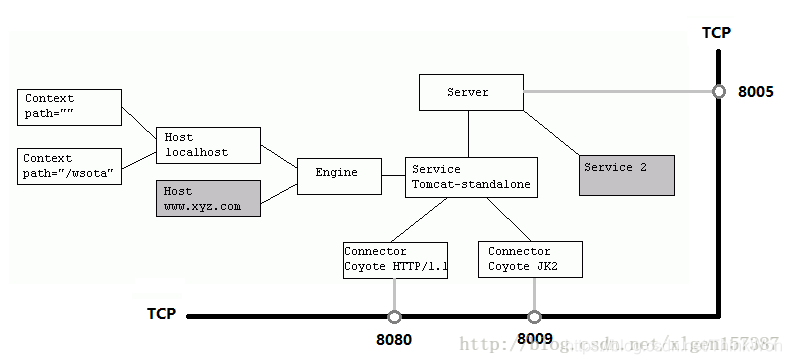
多个 Connector 和一个 Container 就形成了一个 Service，有了 Service 就可以对外提供服务了，但是 Service 还要一个生存的环境，必须要有人能够给她生命、掌握其生死大权，那就非 Server 莫属了！所以整个 Tomcat 的生命周期由 Server 控制。

另外，上述的包含关系或者说是父子关系，都可以在tomcat的conf目录下的server.xml配置文件中看出，下图是删除了注释内容之后的一个完整的server.xml配置文件（Tomcat版本为8.0）

[](https://img-blog.csdnimg.cn/20191021215355649.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

详细的配置文件内容可以到Tomcat官网查看：[Tomcat配置文件](http://tomcat.apache.org/tomcat-8.0-doc/index.html)

上边的配置文件，还可以通过下边的一张结构图更清楚的理解：

[](https://img-blog.csdnimg.cn/2019102121541531.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

Server标签设置的端口号为8005，shutdown=”SHUTDOWN” ，表示在8005端口监听“SHUTDOWN”命令，如果接收到了就会关闭Tomcat。一个Server有一个Service，当然还可以进行配置，一个Service有多个Connector，Service左边的内容都属于Container的，Service下边是Connector。

**Tomcat顶层架构小结**

1. Tomcat中只有一个Server，一个Server可以有多个Service，一个Service可以有多个Connector和一个Container；
2. Server掌管着整个Tomcat的生死大权；
3. Service 是对外提供服务的；
4. Connector用于接受请求并将请求封装成Request和Response来具体处理；
5. Container用于封装和管理Servlet，以及具体处理request请求；

知道了整个Tomcat顶层的分层架构和各个组件之间的关系以及作用，对于绝大多数的开发人员来说Server和Service对我们来说确实很远，而我们开发中绝大部分进行配置的内容是属于Connector和Container的，所以接下来介绍一下Connector和Container。

**Connector和Container的微妙关系**

由上述内容我们大致可以知道一个请求发送到Tomcat之后，首先经过Service然后会交给我们的Connector，Connector用于接收请求并将接收的请求封装为Request和Response来具体处理，Request和Response封装完之后再交由Container进行处理，Container处理完请求之后再返回给Connector，最后在由Connector通过Socket将处理的结果返回给客户端，这样整个请求的就处理完了！

Connector最底层使用的是Socket来进行连接的，Request和Response是按照HTTP协议来封装的，所以Connector同时需要实现TCP/IP协议和HTTP协议！

Tomcat既然需要处理请求，那么肯定需要先接收到这个请求，接收请求这个东西我们首先就需要看一下Connector！

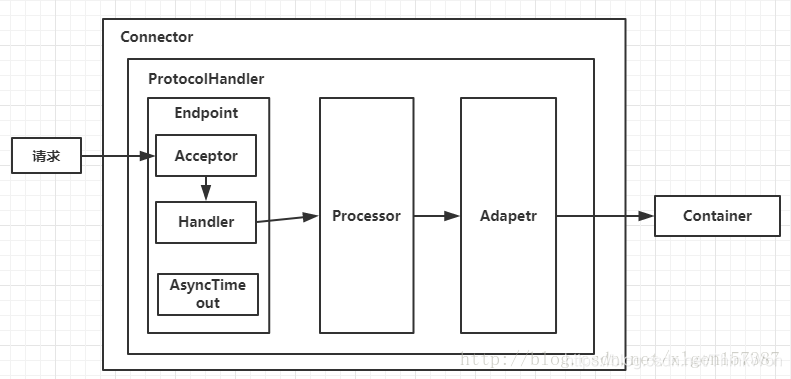
Connector架构分析

Connector用于接受请求并将请求封装成Request和Response，然后交给Container进行处理，Container处理完之后在交给Connector返回给客户端。

因此，我们可以把Connector分为四个方面进行理解：

1. Connector如何接受请求的？
2. 如何将请求封装成Request和Response的？
3. 封装完之后的Request和Response如何交给Container进行处理的？
4. Container处理完之后如何交给Connector并返回给客户端的？

首先看一下Connector的结构图（图B），如下所示：

[](https://img-blog.csdnimg.cn/20191021215430677.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

Connector就是使用ProtocolHandler来处理请求的，不同的ProtocolHandler代表不同的连接类型，比如：Http11Protocol使用的是普通Socket来连接的，Http11NioProtocol使用的是NioSocket来连接的。

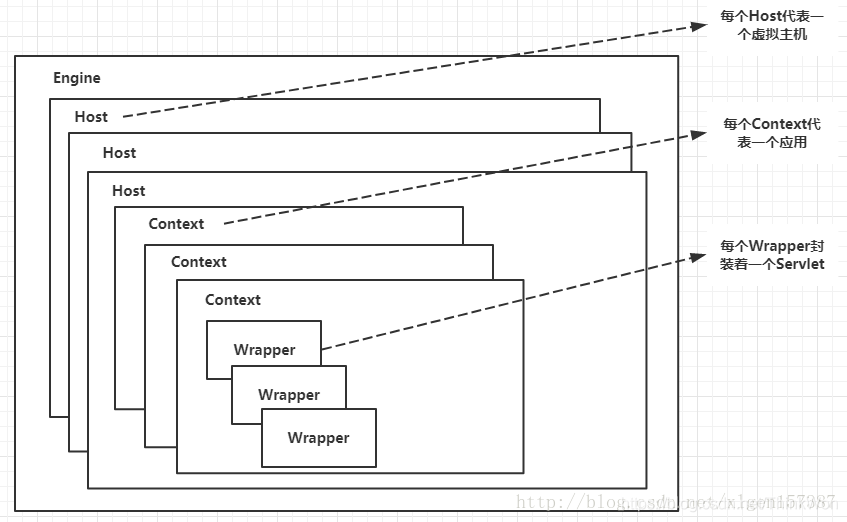
其中ProtocolHandler由包含了三个部件：Endpoint、Processor、Adapter。

1. Endpoint用来处理底层Socket的网络连接，Processor用于将Endpoint接收到的Socket封装成Request，Adapter用于将Request交给Container进行具体的处理。
2. Endpoint由于是处理底层的Socket网络连接，因此Endpoint是用来实现TCP/IP协议的，而Processor用来实现HTTP协议的，Adapter将请求适配到Servlet容器进行具体的处理。
3. Endpoint的抽象实现AbstractEndpoint里面定义的Acceptor和AsyncTimeout两个内部类和一个Handler接口。Acceptor用于监听请求，AsyncTimeout用于检查异步Request的超时，Handler用于处理接收到的Socket，在内部调用Processor进行处理。

至此，我们应该很轻松的回答1，2，3的问题了，但是4还是不知道，那么我们就来看一下Container是如何进行处理的以及处理完之后是如何将处理完的结果返回给Connector的？

**Container架构分析**

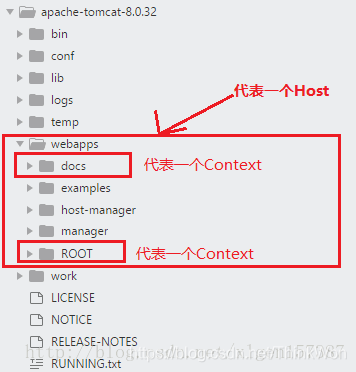
Container用于封装和管理Servlet，以及具体处理Request请求，在Container内部包含了4个子容器，结构图如下（图C）：

[](https://img-blog.csdnimg.cn/20191021215443306.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

4个子容器的作用分别是：

1. Engine：引擎，用来管理多个站点，一个Service最多只能有一个Engine；
2. Host：代表一个站点，也可以叫虚拟主机，通过配置Host就可以添加站点；
3. Context：代表一个应用程序，对应着平时开发的一套程序，或者一个WEB-INF目录以及下面的web.xml文件；
4. Wrapper：每一Wrapper封装着一个Servlet；

下面找一个Tomcat的文件目录对照一下，如下图所示：

[](https://img-blog.csdnimg.cn/20191021215455991.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

Context和Host的区别是Context表示一个应用，我们的Tomcat中默认的配置下webapps下的每一个文件夹目录都是一个Context，其中ROOT目录中存放着主应用，其他目录存放着子应用，而整个webapps就是一个Host站点。

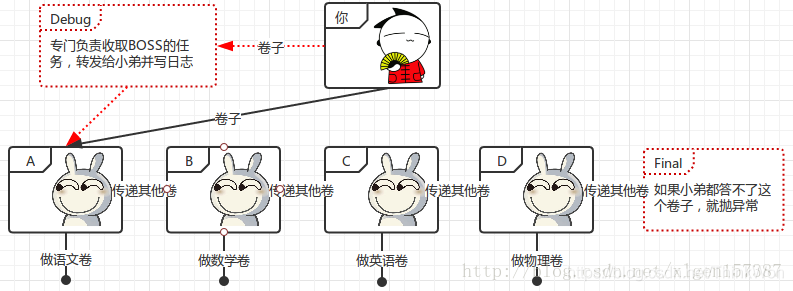
我们访问应用Context的时候，如果是ROOT下的则直接使用域名就可以访问，例如：www.baidu.com，如果是Host（webapps）下的其他应用，则可以使用www.baidu.com/docs进行访问，当然默认指定的根应用（ROOT）是可以进行设定的，只不过Host站点下默认的主应用是ROOT目录下的。

看到这里我们知道Container是什么，但是还是不知道Container是如何进行请求处理的以及处理完之后是如何将处理完的结果返回给Connector的？别急！下边就开始探讨一下Container是如何进行处理的！

**Container如何处理请求的**

Container处理请求是使用Pipeline-Valve管道来处理的！（Valve是阀门之意）

Pipeline-Valve是**责任链模式**，责任链模式是指在一个请求处理的过程中有很多处理者依次对请求进行处理，每个处理者负责做自己相应的处理，处理完之后将处理后的结果返回，再让下一个处理者继续处理。

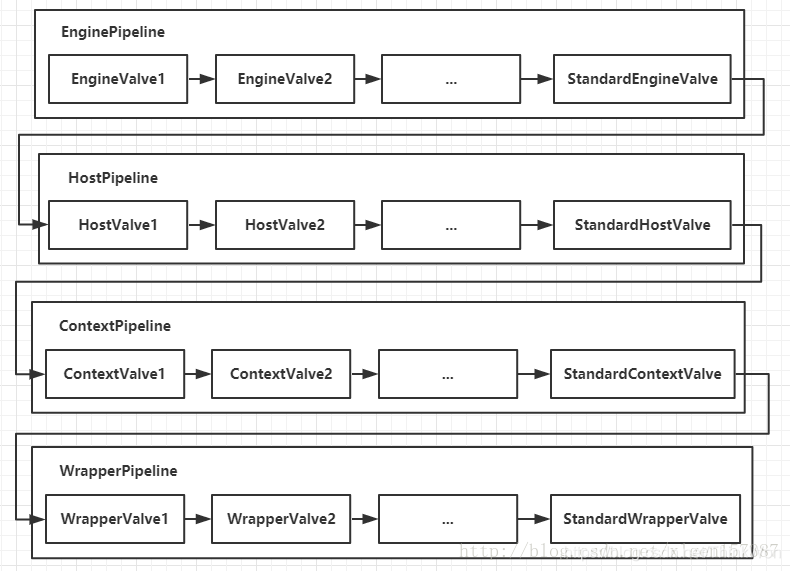
[](https://img-blog.csdnimg.cn/20191021215507725.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

但是！Pipeline-Valve使用的责任链模式和普通的责任链模式有些不同！区别主要有以下两点：

* 每个Pipeline都有特定的Valve，而且是在管道的最后一个执行，这个Valve叫做BaseValve，BaseValve是不可删除的；
* 在上层容器的管道的BaseValve中会调用下层容器的管道。

我们知道Container包含四个子容器，而这四个子容器对应的BaseValve分别在：StandardEngineValve、StandardHostValve、StandardContextValve、StandardWrapperValve。

Pipeline的处理流程图如下（图D）：

[](https://img-blog.csdnimg.cn/20191021215519408.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L1RoaW5rV29u,size_16,color_FFFFFF,t_70)

* Connector在接收到请求后会首先调用最顶层容器的Pipeline来处理，这里的最顶层容器的Pipeline就是EnginePipeline（Engine的管道）；
* 在Engine的管道中依次会执行EngineValve1、EngineValve2等等，最后会执行StandardEngineValve，在StandardEngineValve中会调用Host管道，然后再依次执行Host的HostValve1、HostValve2等，最后在执行StandardHostValve，然后再依次调用Context的管道和Wrapper的管道，最后执行到StandardWrapperValve。
* 当执行到StandardWrapperValve的时候，会在StandardWrapperValve中创建FilterChain，并调用其doFilter方法来处理请求，这个FilterChain包含着我们配置的与请求相匹配的Filter和Servlet，其doFilter方法会依次调用所有的Filter的doFilter方法和Servlet的service方法，这样请求就得到了处理！
* 当所有的Pipeline-Valve都执行完之后，并且处理完了具体的请求，这个时候就可以将返回的结果交给Connector了，Connector在通过Socket的方式将结果返回给客户端。