**精尽 Spring 面试题**

以下面试题，基于网络整理，和自己编辑。具体参考的文章，会在文末给出所有的链接。

如果胖友有自己的疑问，欢迎在星球提问，我们一起整理吊吊的 Spring 面试题的大保健。

而题目的难度，艿艿尽量按照从容易到困难的顺序，逐步下去。

注意 1 ，这个面试题，暂时不包括 Spring MVC 部分。主要以 Spring IoC、Spring AOP、Spring Transaction 为主。

注意 2 ，T T Spring 怎么会有这么多问题，艿艿自己面试很多都不太问，跟背书一样。所以整理的过程，真的是痛苦万分。

Spring 是一个很庞大的技术体系，可以说包容一切，所以本文我们按照下面的顺序，罗列各种面试题：

* Spring **整体**相关的面试
* Spring **IoC** 相关的面试题
* Spring **AOP** 相关的面试题
* Spring **Transaction** 相关的面试题
* Spring **Data Access** 相关的面试题

**Spring 整体**

**什么是 Spring Framework？**

Spring 是一个开源应用框架，旨在降低应用程序开发的复杂度。

* 它是轻量级、松散耦合的。

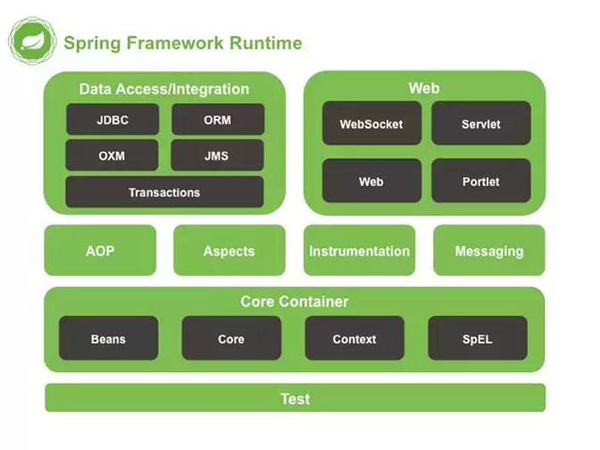
它的轻量级主要是相对于 EJB 。随着 Spring 的体系越来越庞大，大家被 Spring 的配置搞懵逼了，所以后来出了 Spring Boot 。

* 它具有分层体系结构，允许用户选择组件，同时还为 J2EE 应用程序开发提供了一个有凝聚力的框架。
* 它可以集成其他框架，如 Spring MVC、Hibernate、MyBatis 等，所以又称为框架的框架( 粘合剂、脚手架 )。

**Spring Framework 中有多少个模块，它们分别是什么？**

如下是一张比较早期版本的 Spring Framework 的模块图：

艿艿：因为它的配色比较好看，哈哈哈哈。所以，没自己画一个最新的。



我们按照一个一个分块来说。

**Spring 核心容器**

对应图中，Core Container 。

该层基本上是 Spring Framework 的核心。它包含以下模块：

* Spring Core
* Spring Bean

核心容器提供 Spring 框架的基本功能。核心容器的主要组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转 （IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。

* Spring Context

Spring 上下文是一个配置文件，向 Spring 框架提供上下文信息。Spring 上下文包括企业服务，例如 JNDI、EJB、电子邮件、国际化、事件机制、校验和调度功能。

* SpEL (Spring Expression Language)

Spring 表达式语言全称为 “Spring Expression Language”，缩写为 “SpEL” ，类似于 Struts2 中使用的 OGNL 表达式语言，能在运行时构建复杂表达式、存取对象图属性、对象方法调用等等，并且能与 Spring 功能完美整合，如能用来配置 Bean 定义。

或者说，这块就是 Spring IoC 。

**数据访问**

对应图中，Data Access 。

该层提供与数据库交互的支持。它包含以下模块：

* JDBC (Java DataBase Connectivity)

Spring 对 JDBC 的封装模块，提供了对关系数据库的访问。

* ORM (Object Relational Mapping)

Spring ORM 模块，提供了对 hibernate5 和 JPA 的集成。

* + hibernate5 是一个 ORM 框架。
  + JPA 是一个 Java 持久化 API 。
* OXM (Object XML Mappers)

Spring 提供了一套类似 ORM 的映射机制，用来将 Java 对象和 XML 文件进行映射。这就是 Spring 的对象 XML 映射功能，有时候也成为 XML 的序列化和反序列化。

用的比较少，胖友了解下即可。

* Transaction

Spring 简单而强大的事务管理功能，包括声明式事务和编程式事务。

**Web**

该层提供了创建 Web 应用程序的支持。它包含以下模块：

* WebMVC

MVC 框架是一个全功能的构建 Web 应用程序的 MVC 实现。通过策略接口，MVC 框架变成为高度可配置的，MVC 容纳了大量视图技术，其中包括 JSP、Velocity、Tiles、iText 和 POI。

* WebFlux

基于 Reactive 库的响应式的 Web 开发框架

不了解的胖友，可以看看 [《使用 Spring 5 的 WebFlux 开发反应式 Web 应用》](https://www.ibm.com/developerworks/cn/java/spring5-webflux-reactive/index.html)

* WebSocket

Spring 4.0 的一个最大更新是增加了对 Websocket 的支持。

Websocket 提供了一个在 Web 应用中实现高效、双向通讯，需考虑客户端(浏览器)和服务端之间高频和低延时消息交换的机制。

一般的应用场景有：在线交易、网页聊天、游戏、协作、数据可视化等。

* Portlet 已经废弃

**AOP**

该层支持面向切面编程。它包含以下模块：

* AOP

通过配置管理特性，Spring AOP 模块直接将面向方面的编程功能集成到了 Spring 框架中。所以，可以很容易地使 Spring 框架管理的任何对象支持 AOP。

Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP，不用依赖 EJB 组件，就可以将声明性事务管理集成到应用程序中。

* Aspects

该模块为与 AspectJ 的集成提供支持。

* Instrumentation

该层为类检测和类加载器实现提供支持。

用的比较少，胖友了解下即可。

**其它**

* JMS (Java Messaging Service)

提供了一个 JMS 集成框架，简化了 JMS API 的使用。

可能有胖友不太了解 JMS ，可以看看 [《JMS(Java消息服务)入门教程》](https://www.cnblogs.com/chenpi/p/5559349.html) 。

* Test

该模块为使用 JUnit 和 TestNG 进行测试提供支持。

* Messaging

该模块为 STOMP 提供支持。它还支持注解编程模型，该模型用于从 WebSocket 客户端路由和处理 STOMP 消息。

**使用 Spring 框架能带来哪些好处？**

下面列举了一些使用 Spring 框架带来的主要好处：

艿艿：注意，下文中加粗的内容，这是本句话的关键。

* **DI** ：[**Dependency Injection(DI)**](http://howtodoinjava.com/2013/03/19/inversion-of-control-ioc-and-dependency-injection-di-patterns-in-spring-framework-and-related-interview-questions/) 方法，使得构造器和 JavaBean、properties 文件中的依赖关系一目了然。
* **轻量级**：与 EJB 容器相比较，IoC 容器更加趋向于**轻量级**。这样一来 IoC 容器在有限的内存和 CPU 资源的情况下，进行应用程序的开发和发布就变得十分有利。
* **面向切面编程(AOP)**： Spring 支持面向**切面编程**，同时把应用的业务逻辑与系统的服务分离开来。
* **集成主流框架**：Spring 并没有闭门造车，Spring **集成**了已有的技术栈，比如 ORM 框架、Logging 日期框架、J2EE、Quartz 和 JDK Timer ，以及其他视图技术。
* 模块化：Spring 框架是按照**模块**的形式来组织的。由包和类的命名，就可以看出其所属的模块，开发者仅仅需要选用他们需要的模块即可。
* **便捷的测试**：要 [测试一项用Spring开发的应用程序](http://howtodoinjava.com/2013/04/19/how-to-unit-test-spring-security-authentication-with-junit/) 十分简单，因为**测试**相关的环境代码都已经囊括在框架中了。更加简单的是，利用 JavaBean 形式的 POJO 类，可以很方便的利用依赖注入来写入测试数据。
* **Web 框架**：Spring 的 **Web 框架**亦是一个精心设计的 Web MVC 框架，为开发者们在 Web 框架的选择上提供了一个除了主流框架比如 Struts 、过度设计的、不流行 Web 框架的以外的有力选项。
* **事务管理**：Spring 提供了一个便捷的**事务管理**接口，适用于小型的本地事物处理（比如在单 DB 的环境下）和复杂的共同事物处理（比如利用 JTA 的复杂 DB 环境）。
* **异常处理**：Spring 提供一个方便的 API ，将特定技术的异常(由JDBC, Hibernate, 或 JDO 抛出)转化为一致的、Unchecked 异常。

当然，Spring 代码优点的同时，一定会带来相应的缺点：

* 每个框架都有的问题，调试阶段不直观，后期的 bug 对应阶段，不容易判断问题所在。要花一定的时间去理解它。
* 把很多 JavaEE 的东西封装了，在满足快速开发高质量程序的同时，隐藏了实现细节。

这就导致，很多工程师，离开了 Spring 之后，就不知道怎么工作了。从 Java 工程师，变成了 Spring 工程师。对于有追求的我们，还是需要知其然，知其所以然。

**Spring 框架中都用到了哪些设计模式？**

Spring 框架中使用到了大量的设计模式，下面列举了比较有代表性的：

* 代理模式 — 在 AOP 和 remoting 中被用的比较多。
* 单例模式 — 在 Spring 配置文件中定义的 Bean 默认为单例模式。
* 模板方法 — 用来解决代码重复的问题。比如 [RestTemplate](http://howtodoinjava.com/2015/02/20/spring-restful-client-resttemplate-example/)、JmsTemplate、JdbcTemplate 。
* 前端控制器 — Spring提供了 DispatcherServlet 来对请求进行分发。
* 视图帮助(View Helper) — Spring 提供了一系列的 JSP 标签，高效宏来辅助将分散的代码整合在视图里。
* 依赖注入 — 贯穿于 BeanFactory / ApplicationContext 接口的核心理念。
* 工厂模式 — BeanFactory 用来创建对象的实例。

当然，感兴趣的胖友，觉得不过瘾，可以看看艿艿基友知秋写的几篇文章：

* [《Spring 框架中的设计模式(一)》](http://www.iocoder.cn/Spring/DesignPattern-1)
* [《Spring 框架中的设计模式(二)》](http://www.iocoder.cn/Spring/DesignPattern-2)
* [《Spring 框架中的设计模式(三)》](http://www.iocoder.cn/Spring/DesignPattern-3)
* [《Spring 框架中的设计模式(四)》](http://www.iocoder.cn/Spring/DesignPattern-4)
* [《Spring 框架中的设计模式(五)》](http://www.iocoder.cn/Spring/DesignPattern-5)

**Spring IoC**

下面，我们会将分成 IoC 和 Bean 两部分来分享 Spring 容器的内容。

* IoC ，侧重在于容器。
* Bean ，侧重在于被容器管理的 Bean 。

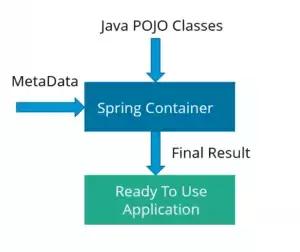
**什么是 Spring IoC 容器？**

注意，正确的拼写是 IoC 。

Spring 框架的核心是 Spring IoC 容器。容器创建 Bean 对象，将它们装配在一起，配置它们并管理它们的完整生命周期。

* Spring 容器使用**依赖注入**来管理组成应用程序的 Bean 对象。
* 容器通过读取提供的**配置元数据** Bean Definition 来接收对象进行实例化，配置和组装的指令。
* 该配置元数据 Bean Definition 可以通过 XML，Java 注解或 Java Config 代码**提供**。

艿艿，注意上面三段段话的**加粗**部分的内容。



**什么是依赖注入？**

在依赖注入中，你不必主动、手动创建对象，但必须描述如何创建它们。

* 你不是直接在代码中将组件和服务连接在一起，而是描述配置文件中哪些组件需要哪些服务。
* 然后，再由 IoC 容器将它们装配在一起。

另外，依赖注入的英文缩写是 Dependency Injection ，简称 DI 。

**IoC 和 DI 有什么区别？**

艿艿的吐槽，最怕这种概念题。下面引用知乎上的一个讨论：[《IoC 和DI 有什么区别？》](https://www.zhihu.com/question/25392984)

IoC 是个更宽泛的概念，DI 是更具体的。引用郑烨的一篇博客，引用郑烨的一篇博客，[我眼中的Spring](https://link.zhihu.com/?target=http%3A//dreamhead.blogbus.com/logs/125334.html)

**Dependency Injection**  
原来，它叫 IoC 。  
Martin Flower 发话了，是个框架都有 IoC ，这不足以新生容器反转的“如何定位插件的具体实现”，于是，它有了个新名字，Dependency Injection 。

其实，它就是一种将调用者与被调用者分离的思想，Uncle Bob 管它叫DIP（Dependency Inversion Principle），并把它归入OO设计原则。  
同 Spring 相比，它更早进入我的大脑。一切都是那么朦胧，直至 Spring 出现。

慢慢的，我知道了它还分为三种：

* Interface Injection（type 1）
* Setter Injection（type 2）
* Constructor Injection（type 3）。

Martin Flower那篇为它更名的大作让我心目关于它的一切趋于完整。

在 Spring 中，它是一切的基础。Spring 的种种优势随之而来。  
于我而言，它为我带来更多的是思维方式的转变，恐怕以后我再也无法写出那种一大块的全功能程序了。

**可以通过多少种方式完成依赖注入？**

通常，依赖注入可以通过**三种**方式完成，即：

上面一个问题的三种方式的英文，下面是三种方式的中文。

* 接口注入
* 构造函数注入
* setter 注入

目前，在 Spring Framework 中，仅使用构造函数和 setter 注入这**两种**方式。

那么这两种方式各有什么优缺点呢？胖友可以简单阅读 [《Spring两种依赖注入方式的比较》](https://my.oschina.net/itblog/blog/203746)，不用太较真。综述来说：

| **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 |

* 实际场景下，setting 注入使用的更多。

**Spring 中有多少种 IoC 容器？**

Spring 提供了两种( 不是“个” ) IoC 容器，分别是 BeanFactory、ApplicationContext 。

**BeanFactory**

BeanFactory 在 spring-beans 项目提供。

BeanFactory ，就像一个包含 Bean 集合的工厂类。它会在客户端要求时实例化 Bean 对象。

**ApplicationContext**

ApplicationContext 在 spring-context 项目提供。

ApplicationContext 接口扩展了 BeanFactory 接口，它在 BeanFactory 基础上提供了一些额外的功能。内置如下功能：

* MessageSource ：管理 message ，实现国际化等功能。
* ApplicationEventPublisher ：事件发布。
* ResourcePatternResolver ：多资源加载。
* EnvironmentCapable ：系统 Environment（profile + Properties）相关。
* Lifecycle ：管理生命周期。
* Closable ：关闭，释放资源
* InitializingBean：自定义初始化。
* BeanNameAware：设置 beanName 的 Aware 接口。

另外，ApplicationContext 会自动初始化非懒加载的 Bean 对象们。

详细的内容，感兴趣的胖友，可以看看 [《【死磕 Spring】—— ApplicationContext 相关接口架构分析》](http://svip.iocoder.cn/Spring/ApplicationContext/) 一文。源码之前无秘密。简单总结下 BeanFactory 与 ApplicationContext 两者的差异：

艿艿：可能很多胖友没看过源码，所以会比较难。

| **BeanFactory** | **ApplicationContext** |
| --- | --- |
| 它使用懒加载 | 它使用即时加载 |
| 它使用语法显式提供资源对象 | 它自己创建和管理资源对象 |
| 不支持国际化 | 支持国际化 |
| 不支持基于依赖的注解 | 支持基于依赖的注解 |

另外，BeanFactory 也被称为**低级**容器，而 ApplicationContext 被称为**高级**容器。

**请介绍下常用的 BeanFactory 容器？**

BeanFactory 最常用的是 XmlBeanFactory 。它可以根据 XML 文件中定义的内容，创建相应的 Bean。

**请介绍下常用的 ApplicationContext 容器？**

以下是三种较常见的 ApplicationContext 实现方式：

* 1、ClassPathXmlApplicationContext ：从 ClassPath 的 XML 配置文件中读取上下文，并生成上下文定义。应用程序上下文从程序环境变量中取得。示例代码如下：

|  |
| --- |
| ApplicationContext context = new ClassPathXmlApplicationContext(“bean.xml”); |

* 2、FileSystemXmlApplicationContext ：由文件系统中的XML配置文件读取上下文。示例代码如下：

|  |
| --- |
| ApplicationContext context = new FileSystemXmlApplicationContext(“bean.xml”); |

* 3、XmlWebApplicationContext ：由 Web 应用的XML文件读取上下文。例如我们在 Spring MVC 使用的情况。

当然，目前我们更多的是使用 Spring Boot 为主，所以使用的是第四种 ApplicationContext 容器，ConfigServletWebServerApplicationContext 。

**列举一些 IoC 的一些好处？**

* 它将最小化应用程序中的代码量。
* 它以最小的影响和最少的侵入机制促进松耦合。
* 它支持即时的实例化和延迟加载 Bean 对象。
* 它将使您的应用程序易于测试，因为它不需要单元测试用例中的任何单例或 JNDI 查找机制。

**简述 Spring IoC 的实现机制？**

简单来说，Spring 中的 IoC 的实现原理，就是**工厂模式**加**反射机制**。代码如下：

|  |
| --- |
| 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[] args) {  Fruit f = Factory.getInstance("io.github.dunwu.spring.Apple");  if(f != null){  f.eat();  }  }   } |

* Fruit 接口，有 Apple 和 Orange 两个实现类。
* Factory 工厂，通过反射机制，创建 className 对应的 Fruit 对象。
* Client 通过 Factory 工厂，获得对应的 Fruit 对象。
* 😈 实际情况下，Spring IoC 比这个复杂很多很多，例如单例 Bean 对象，Bean 的属性注入，相互依赖的 Bean 的处理，以及等等。

在基友 [《面试问烂的 Spring IoC 过程》](http://www.iocoder.cn/Fight/Interview-poorly-asked-Spring-IOC-process-1/) 的文章中，把 Spring IoC 相关的内容，讲的非常不错。

**Spring 框架中有哪些不同类型的事件？**

Spring 的 ApplicationContext 提供了支持事件和代码中监听器的功能。

我们可以创建 Bean 用来监听在 ApplicationContext 中发布的事件。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。示例代码如下：

|  |
| --- |
| public class AllApplicationEventListener implements ApplicationListener<ApplicationEvent> {     @Override   public void onApplicationEvent(ApplicationEvent applicationEvent) {   // process event   }   } |

Spring 提供了以下五种标准的事件：

1. 上下文更新事件（ContextRefreshedEvent）：该事件会在ApplicationContext 被初始化或者更新时发布。也可以在调用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 We b应用中，当一个HTTP 请求（request）结束触发该事件。

除了上面介绍的事件以外，还可以通过扩展 ApplicationEvent 类来开发**自定义**的事件。

① 示例自定义的事件的类，代码如下：

|  |
| --- |
| public class CustomApplicationEvent extends ApplicationEvent{    public CustomApplicationEvent(Object source, final String msg) {   super(source);  }   } |

② 为了监听这个事件，还需要创建一个监听器。示例代码如下：

|  |
| --- |
| public class CustomEventListener implements ApplicationListener<CustomApplicationEvent> {   @Override   public void onApplicationEvent(CustomApplicationEvent applicationEvent) {   // handle event   }   } |

③ 之后通过 ApplicationContext 接口的 #publishEvent(Object event) 方法，来发布自定义事件。示例代码如下：

|  |
| --- |
| // 创建 CustomApplicationEvent 事件 CustomApplicationEvent customEvent = new CustomApplicationEvent(applicationContext, "Test message"); // 发布事件 applicationContext.publishEvent(customEvent); |

**Spring Bean**

**什么是 Spring Bean ？**

* Bean 由 Spring IoC 容器实例化，配置，装配和管理。
* Bean 是基于用户提供给 IoC 容器的配置元数据 Bean Definition 创建。

这个问题，胖友可以在回过头看 [「什么是 Spring IoC 容器？」](http://yudaoyuanma:ydym1024@svip.iocoder.cn/Spring/Interview/) 问题，相互对照。

**Spring 有哪些配置方式**

单纯从 Spring Framework 提供的方式，一共有三种：

* 1、XML 配置文件。

Bean 所需的依赖项和服务在 XML 格式的配置文件中指定。这些配置文件通常包含许多 bean 定义和特定于应用程序的配置选项。它们通常以 bean 标签开头。例如：

|  |
| --- |
| <bean id="studentBean" class="org.edureka.firstSpring.StudentBean">  <property name="name" value="Edureka"></property> </bean> |

* 2、注解配置。

您可以通过在相关的类，方法或字段声明上使用注解，将 Bean 配置为组件类本身，而不是使用 XML 来描述 Bean 装配。默认情况下，Spring 容器中未打开注解装配。因此，您需要在使用它之前在 Spring 配置文件中启用它。例如：

|  |
| --- |
| <beans> <context:annotation-config/> <!-- bean definitions go here --> </beans> |

* 3、Java Config 配置。

Spring 的 Java 配置是通过使用 @Bean 和 @Configuration 来实现。

* + @Bean 注解扮演与 <bean /> 元素相同的角色。
  + @Configuration 类允许通过简单地调用同一个类中的其他 @Bean 方法来定义 Bean 间依赖关系。
  + 例如：

|  |
| --- |
| @Configuration public class StudentConfig {    @Bean  public StudentBean myStudent() {  return new StudentBean();  }   } |

* + - 是不是很熟悉 😈

目前主要使用 **Java Config** 配置为主。当然，三种配置方式是可以混合使用的。例如说：

* Dubbo 服务的配置，艿艿喜欢使用 XML 。
* Spring MVC 请求的配置，艿艿喜欢使用 @RequestMapping 注解。
* Spring MVC 拦截器的配置，艿艿喜欢 Java Config 配置。

另外，现在已经是 Spring Boot 的天下，所以更加是 **Java Config** 配置为主。

**Spring 支持几种 Bean Scope ？**

艿艿，这个是一个比较小众的题目，简单了解即可。

Spring Bean 支持 5 种 Scope ，分别如下：

* Singleton - 每个 Spring IoC 容器仅有一个单 Bean 实例。**默认**
* Prototype - 每次请求都会产生一个新的实例。
* Request - 每一次 HTTP 请求都会产生一个新的 Bean 实例，并且该 Bean 仅在当前 HTTP 请求内有效。
* Session - 每一个的 Session 都会产生一个新的 Bean 实例，同时该 Bean 仅在当前 HTTP Session 内有效。
* Application - 每一个 Web Application 都会产生一个新的 Bean ，同时该 Bean 仅在当前 Web Application 内有效。

另外，网络上很多文章说有 Global-session 级别，它是 Portlet 模块独有，目前已经废弃，在 Spring5 中是找不到的。

仅当用户使用支持 Web 的 ApplicationContext 时，**最后三个才可用**。

再补充一点，开发者是可以**自定义** Bean Scope ，具体可参见 [《Spring（10）—— Bean 作用范围（二）—— 自定义 Scope》](https://blog.csdn.net/elim168/article/details/75581670) 。

不错呢，还是那句话，这个题目简单了解下即可，实际常用的只有 Singleton 和 Prototype 两种级别，甚至说，只有 Singleton 级别。😈

**Spring Bean 在容器的生命周期是什么样的？**

艿艿说：这是一个比较高级的 Spring 的面试题，非常常见，并且答对比较加分。当然，如果实际真正弄懂，需要对 Spring Bean 的源码，有比较好的理解，所以 [《精尽 Spring 源码》](http://svip.iocoder.cn/categories/Spring/) 系列，该读还是读吧。

艿艿：要注意下面每段话，艿艿进行加粗的地方。

Spring Bean 的**初始化**流程如下：

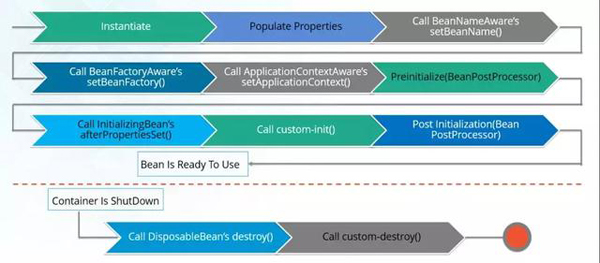
* 实例化 Bean 对象
  + Spring 容器根据配置中的 Bean Definition(定义)中**实例化** Bean 对象。

Bean Definition 可以通过 XML，Java 注解或 Java Config 代码提供。

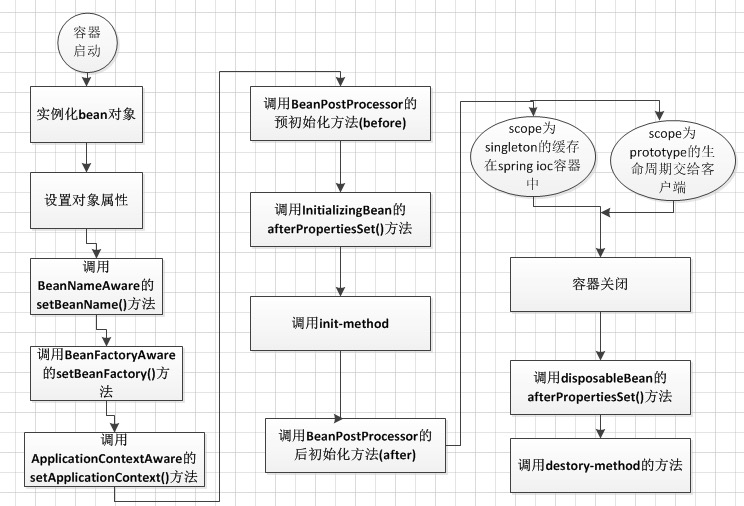
* + Spring 使用依赖注入**填充**所有属性，如 Bean 中所定义的配置。
* Aware 相关的属性，注入到 Bean 对象
  + 如果 Bean 实现 **BeanNameAware** 接口，则工厂通过传递 Bean 的 beanName 来调用 #setBeanName(String name) 方法。
  + 如果 Bean 实现 **BeanFactoryAware** 接口，工厂通过传递自身的实例来调用 #setBeanFactory(BeanFactory beanFactory) 方法。
* 调用相应的方法，进一步初始化 Bean 对象
  + 如果存在与 Bean 关联的任何 **BeanPostProcessor** 们，则调用 #preProcessBeforeInitialization(Object bean, String beanName) 方法。
  + 如果 Bean 实现 **InitializingBean** 接口，则会调用 #afterPropertiesSet() 方法。
  + 如果为 Bean 指定了 **init** 方法（例如 <bean /> 的 init-method 属性），那么将调用该方法。
  + 如果存在与 Bean 关联的任何 **BeanPostProcessor** 们，则将调用 #postProcessAfterInitialization(Object bean, String beanName) 方法。

Spring Bean 的**销毁**流程如下：

* 如果 Bean 实现 **DisposableBean** 接口，当 spring 容器关闭时，会调用 #destroy() 方法。
* 如果为 bean 指定了 **destroy** 方法（例如 <bean /> 的 destroy-method 属性），那么将调用该方法。

整体如下图：

无意中，艿艿又翻到一张有趣的整体图，如下图：



**什么是 Spring 的内部 bean？**

只有将 Bean **仅**用作另一个 Bean 的属性时，才能将 Bean 声明为内部 Bean。

* 为了定义 Bean，Spring 提供基于 XML 的配置元数据在 <property>或 <constructor-arg> 中提供了 <bean>元素的使用。
* 内部 Bean 总是**匿名**的，并且它们总是作为**原型 Prototype** 。

例如，假设我们有一个 Student 类，其中引用了 Person 类。这里我们将只创建一个 Person 类实例并在 Student 中使用它。示例代码如下：

|  |
| --- |
| // Student.java  public class Student {   private Person person;    // ... Setters and Getters }  // Person.java  public class Person {   private String name;  private String address;    // ... Setters and Getters } |

|  |
| --- |
| <!-- bean.xml -->  <bean id=“StudentBean" class="com.edureka.Student">  <property name="person">  <!--This is inner bean -->  <bean class="com.edureka.Person">  <property name="name" value=“Scott"></property>  <property name="address" value=“Bangalore"></property>  </bean>  </property> </bean> |

**什么是 Spring 装配？**

当 Bean 在 Spring 容器中组合在一起时，它被称为**装配**或 **Bean 装配**。Spring 容器需要知道需要什么 Bean 以及容器应该如何使用依赖注入来将 Bean 绑定在一起，同时装配 Bean 。

装配，和上文提到的 DI 依赖注入，实际是一个东西。

**自动装配有哪些方式？**

Spring 容器能够自动装配 Bean 。也就是说，可以通过检查 BeanFactory 的内容让 Spring 自动解析 Bean 的协作者。

自动装配的不同模式：

* no - 这是默认设置，表示没有自动装配。应使用显式 Bean 引用进行装配。
* byName - 它根据 Bean 的名称注入对象依赖项。它匹配并装配其属性与 XML 文件中由相同名称定义的 Bean 。
* 【最常用】**byType** - 它根据类型注入对象依赖项。如果属性的类型与 XML 文件中的一个 Bean 类型匹配，则匹配并装配属性。
* 构造函数 - 它通过调用类的构造函数来注入依赖项。它有大量的参数。
* autodetect - 首先容器尝试通过构造函数使用 autowire 装配，如果不能，则尝试通过 byType 自动装配。

**自动装配有什么局限？**

艿艿：这个题目，了解下即可，也不是很准确。

* 覆盖的可能性 - 您始终可以使用 <constructor-arg> 和 <property> 设置指定依赖项，这将覆盖自动装配。
* 基本元数据类型 - 简单属性（如原数据类型，字符串和类）无法自动装配。

这种，严格来说，也不能称为局限。因为可以通过配置文件来解决。

* 令人困惑的性质 - 总是喜欢使用明确的装配，因为自动装配不太精确。

**解释什么叫延迟加载？**

默认情况下，容器启动之后会将所有作用域为**单例**的 Bean 都创建好，但是有的业务场景我们并不需要它提前都创建好。此时，我们可以在Bean 中设置 lzay-init = "true" 。

* 这样，当容器启动之后，作用域为单例的 Bean ，就不在创建。
* 而是在获得该 Bean 时，才真正在创建加载。

**Spring 框架中的单例 Bean 是线程安全的么？**

Spring 框架并没有对[单例](http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/) Bean 进行任何多线程的封装处理。

* 关于单例 Bean 的[线程安全](http://howtodoinjava.com/2014/06/02/what-is-thread-safety/)和并发问题，需要开发者自行去搞定。
* 并且，单例的线程安全问题，也不是 Spring 应该去关心的。Spring 应该做的是，提供根据配置，创建单例 Bean 或多例 Bean 的功能。

当然，但实际上，大部分的 Spring Bean 并没有可变的状态(比如Serview 类和 DAO 类)，所以在某种程度上说 Spring 的单例 Bean 是线程安全的。

如果你的 Bean 有多种状态的话，就需要自行保证线程安全。最浅显的解决办法，就是将多态 Bean 的作用域( Scope )由 Singleton 变更为 Prototype 。

**Spring Bean 怎么解决循环依赖的问题？**

艿艿说：能回答出这个问题的，一般是比较厉害的。

这是个比较复杂的问题，有能力的胖友，建议看下 [《【死磕 Spring】—— IoC 之加载 Bean：创建 Bean（五）之循环依赖处理》](http://svip.iocoder.cn/Spring/IoC-get-Bean-createBean-5/)

感觉，不通过源码，很难解释清楚这个问题。如果看不懂的胖友，可以在认真看完，在星球里，我们一起多交流下。好玩的。

**Spring 注解**

这块内容，实际写在 [「Spring Bean」](http://yudaoyuanma:ydym1024@svip.iocoder.cn/Spring/Interview/) 中比较合适，考虑到后续的问题，都是关于注解的，所以单独起一个大的章节。

**什么是基于注解的容器配置？**

不使用 XML 来描述 Bean 装配，开发人员通过在相关的类，方法或字段声明上使用**注解**将配置移动到组件类本身。它可以作为 XML 设置的替代方案。例如：

Spring 的 Java 配置是通过使用 @Bean 和 @Configuration 来实现。

* @Bean 注解，扮演与 <bean /> 元素相同的角色。
* @Configuration 注解的类，允许通过简单地调用同一个类中的其他 @Bean 方法来定义 Bean 间依赖关系。

示例如下：

|  |
| --- |
| @Configuration public class StudentConfig {   @Bean  public StudentBean myStudent() {  return new StudentBean();  }   } |

**如何在 Spring 中启动注解装配？**

默认情况下，Spring 容器中未打开注解装配。因此，要使用基于注解装配，我们必须通过配置 <context：annotation-config /> 元素在 Spring 配置文件中启用它。

当然，如果胖友是使用 Spring Boot ，默认情况下已经开启。

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

* @Component ：它将 Java 类标记为 Bean 。它是任何 Spring 管理组件的**通用**构造型。
* @Controller ：它将一个类标记为 Spring Web MVC **控制器**。
* @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 注解有什么用？**

@Required 注解，应用于 Bean 属性 setter 方法。

* 此注解仅指示必须在配置时使用 Bean 定义中的显式属性值或使用自动装配填充受影响的 Bean 属性。
* 如果尚未填充受影响的 Bean 属性，则容器将抛出 BeanInitializationException 异常。

示例代码如下：

|  |
| --- |
| public class Employee {   private String name;    @Required  public void setName(String name){  this.name=name;  }    public string getName(){  return name;  }   } |

* T T 貌似平时很少用这个注解噢。

**@Autowired 注解有什么用？**

@Autowired 注解，可以更准确地控制应该在何处以及如何进行自动装配。

* 此注解用于在 setter 方法，构造函数，具有任意名称或多个参数的属性或方法上自动装配 Bean。
* 默认情况下，它是类型驱动的注入。

示例代码如下：

|  |
| --- |
| public class EmpAccount {    @Autowired  private Employee emp;   } |

**@Qualifier 注解有什么用？**

当你创建多个**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 ID 应该装配哪个**确切的** Bean 来消除歧义。

例如，应用中有两个类型为 Employee 的 Bean ID 为 "emp1" 和 "emp2" ，此处，我们希望 EmployeeAccount Bean 注入 "emp1" 对应的 Bean 对象。代码如下：

|  |
| --- |
| public class EmployeeAccount {   @Autowired  @Qualifier(emp1)  private Employee emp;  } |

**Spring AOP**

Spring AOP 的面试题中，大多数都是概念题，主要是对切面的理解。概念点主要有：

* AOP
* Aspect
* JoinPoint
* PointCut
* Advice
* Target
* AOP Proxy
* Weaving
* 在阅读完 [「Spring AOP」](http://yudaoyuanma:ydym1024@svip.iocoder.cn/Spring/Interview/) 的面试题后，在回过头思考下这些概念点，到底理解了多少。注意，不是背，理解！

非常推荐阅读如下两篇文章：

* [《彻底征服 Spring AOP 之理论篇》](https://segmentfault.com/a/1190000007469968)
* [《彻底征服 Spring AOP 之实战篇》](https://segmentfault.com/a/1190000007469982)

**什么是 AOP ？**

AOP(Aspect-Oriented Programming)，即**面向切面编程**, 它与 OOP( Object-Oriented Programming, 面向对象编程) 相辅相成， 提供了与 OOP 不同的抽象软件结构的视角。

* 在 OOP 中，以类( Class )作为基本单元
* 在 AOP 中，以**切面( Aspect )**作为基本单元。

**什么是 Aspect ？**

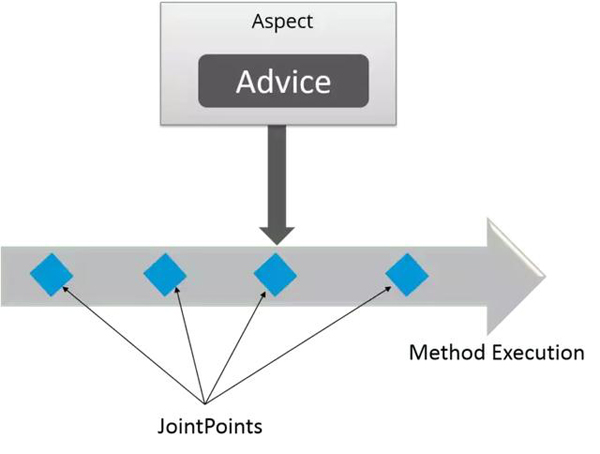
Aspect 由 **PointCut** 和 **Advice** 组成。

* 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:

1. 如何通过 PointCut 和 Advice 定位到特定的 **JoinPoint** 上。
2. 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**



**什么是 JoinPoint ?**

JoinPoint ，**切点**，程序运行中的一些时间点, 例如：

* 一个方法的执行。
* 或者是一个异常的处理。

在 Spring AOP 中，JoinPoint 总是方法的执行点。

**什么是 PointCut ？**

PointCut ，**匹配** JoinPoint 的谓词(a predicate that matches join points)。

简单来说，PointCut 是匹配 JoinPoint 的条件。

* Advice 是和特定的 PointCut 关联的，并且在 PointCut 相匹配的 JoinPoint 中执行。即 Advice => PointCut => JoinPoint 。
* 在 Spring 中, 所有的方法都可以认为是 JoinPoint ，但是我们并不希望在所有的方法上都添加 Advice 。**而 PointCut 的作用**，就是提供一组规则(使用 AspectJ PointCut expression language 来描述) 来匹配 JoinPoint ，给满足规则的 JoinPoint 添加 Advice 。

😈 是不是觉得有点绕，实际场景下，其实也不会弄的这么清楚~~

**关于 JoinPoint 和 PointCut 的区别**

JoinPoint 和 PointCut 本质上就是**两个不同纬度上**的东西。

* 在 Spring AOP 中，所有的方法执行都是 JoinPoint 。而 PointCut 是一个描述信息，它修饰的是 JoinPoint ，通过 PointCut ，我们就可以确定哪些 JoinPoint 可以被织入 Advice 。
* Advice 是在 JoinPoint 上执行的，而 PointCut 规定了哪些 JoinPoint 可以执行哪些 Advice 。

或者，我们在换一种说法：

1. 首先，Advice 通过 PointCut 查询需要被织入的 JoinPoint 。
2. 然后，Advice 在查询到 JoinPoint 上执行逻辑。

**什么是 Advice ？**

Advice ，**通知**。

* 特定 JoinPoint 处的 Aspect 所采取的动作称为 Advice 。
* Spring AOP 使用一个 Advice 作为拦截器，在 JoinPoint “周围”维护一系列的**拦截器**。

**有哪些类型的 Advice？**

* Before - 这些类型的 Advice 在 JoinPoint 方法之前执行，并使用 @Before 注解标记进行配置。
* After Returning - 这些类型的 Advice 在连接点方法正常执行后执行，并使用 @AfterReturning 注解标记进行配置。
* After Throwing - 这些类型的 Advice 仅在 JoinPoint 方法通过抛出异常退出并使用 @AfterThrowing 注解标记配置时执行。
* After Finally - 这些类型的 Advice 在连接点方法之后执行，无论方法退出是正常还是异常返回，并使用 @After 注解标记进行配置。
* Around - 这些类型的 Advice 在连接点之前和之后执行，并使用 @Around 注解标记进行配置。

😈 看起来，是不是和拦截器的执行时间，有几分相似。实际上，用于拦截效果的各种实现，大体都是类似的。

**什么是 Target ？**

Target ，织入 Advice 的**目标对象**。目标对象也被称为 **Advised Object** 。

* 因为 Spring AOP 使用运行时代理的方式来实现 Aspect ，因此 Advised Object 总是一个代理对象(Proxied Object) 。
* **注意, Advised Object 指的不是原来的对象，而是织入 Advice 后所产生的代理对象**。
* Advice + Target Object = Advised Object = Proxy 。

**AOP 有哪些实现方式？**

实现 AOP 的技术，主要分为两大类：

* ① **静态代理** - 指使用 AOP 框架提供的命令进行编译，从而在编译阶段就可生成 AOP 代理类，因此也称为编译时增强。
  + 编译时编织（特殊编译器实现）
  + 类加载时编织（特殊的类加载器实现）。

例如，SkyWalking 基于 Java Agent 机制，配置上 ByteBuddy 库，实现类加载时编织时增强，从而实现链路追踪的透明埋点。

感兴趣的胖友，可以看看 [《SkyWalking 源码分析之 JavaAgent 工具 ByteBuddy 的应用》](http://www.kailing.pub/article/index/arcid/178.html) 。

* ② **动态代理** - 在运行时在内存中“临时”生成 AOP 动态代理类，因此也被称为运行时增强。目前 Spring 中使用了两种动态代理库：
  + JDK 动态代理
  + CGLIB

那么 Spring 什么时候使用 JDK 动态代理，什么时候使用 CGLIB 呢？

|  |
| --- |
| // From 《Spring 源码深度解析》P172 // Spring AOP 部分使用 JDK 动态代理或者 CGLIB 来为目标对象创建代理。（建议尽量使用 JDK 的动态代理） // 如果被代理的目标对象实现了至少一个接口，则会使用 JDK 动态代理。所有该目标类型实现的接口都讲被代理。 // 若该目标对象没有实现任何接口，则创建一个 CGLIB 代理。 // 如果你希望强制使用 CGLIB 代理，（例如希望代理目标对象的所有方法，而不只是实现自接口的方法）那也可以。但是需要考虑以下两个方法： // 1> 无法通知(advise) Final 方法，因为它们不能被覆盖。 // 2> 你需要将 CGLIB 二进制发型包放在 classpath 下面。 // 为什么 Spring 默认使用 JDK 的动态代理呢？笔者猜测原因如下： // 1> 使用 JDK 原生支持，减少三方依赖 // 2> JDK8 开始后，JDK 代理的性能差距 CGLIB 的性能不会太多。可参见：https://www.cnblogs.com/haiq/p/4304615.html |

* 实际上，Spring AOP 的代码量不大，与其在窗户外面不清不楚，不如捅破它！感兴趣的胖友，可以撸一撸 [《精尽 Spring 源码分析 —— AOP 源码简单导读》](http://svip.iocoder.cn/Spring/aop-simple-intro/) 。

或者，我们来换一个解答答案：

Spring AOP 中的动态代理主要有两种方式，

* JDK 动态代理

JDK 动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是 InvocationHandler 接口和 Proxy 类。

* CGLIB 动态代理

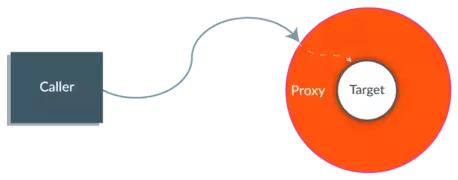
如果目标类没有实现接口，那么 Spring AOP 会选择使用 CGLIB 来动态代理目标类。当然，Spring 也支持配置，**强制**使用 CGLIB 动态代理。  
CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB 是通过继承的方式做的动态代理，因此如果某个类被标记为 final ，那么它是无法使用 CGLIB 做动态代理的。

**Spring AOP and AspectJ AOP 有什么区别？**

* 代理方式不同
  + Spring AOP 基于动态代理方式实现。
  + AspectJ AOP 基于静态代理方式实现。
* PointCut 支持力度不同
  + Spring AOP **仅**支持方法级别的 PointCut 。
  + AspectJ AOP 提供了完全的 AOP 支持，它还支持属性级别的 PointCut 。

**什么是编织（Weaving）？**

Weaving ，**编织**。

* 为了创建一个 Advice 对象而链接一个 Aspect 和其它应用类型或对象，称为编织（Weaving）。
* 在 Spring AOP 中，编织在运行时执行，即动态代理。请参考下图：

**Spring 如何使用 AOP 切面？**

在 Spring AOP 中，有两种方式配置 AOP 切面：

* 基于 **XML** 方式的切面实现。
* 基于 **注解** 方式的切面实现。

目前，主流喜欢使用 **注解** 方式。胖友可以看看 [《彻底征服 Spring AOP 之实战篇》](https://segmentfault.com/a/1190000007469982) 。

**Spring Transaction**

非常推荐阅读如下文章：

* [《可能是最漂亮的 Spring 事务管理详解》](https://mp.weixin.qq.com/s?__biz=MzUzMTA2NTU2Ng==&mid=2247484702&idx=1&sn=c04261d63929db09ff6df7cadc7cca21&chksm=fa497aafcd3ef3b94082da7bca841b5b7b528eb2a52dbc4eb647b97be63a9a1cf38a9e71bf90&token=165108535&lang=zh_CN#rd)

**什么是事务？**

事务就是对一系列的数据库操作（比如插入多条数据）进行统一的提交或回滚操作，如果插入成功，那么一起成功，如果中间有一条出现异常，那么回滚之前的所有操作。

这样可以防止出现脏数据，防止数据库数据出现问题。

**事务的特性指的是？**

指的是 **ACID** ，如下图所示：



1. **原子性** Atomicity ：一个事务（transaction）中的所有操作，或者全部完成，或者全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被恢复（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。即，事务不可分割、不可约简。
2. **一致性** Consistency ：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设[约束](https://zh.wikipedia.org/wiki/%E6%95%B0%E6%8D%AE%E5%AE%8C%E6%95%B4%E6%80%A7)、[触发器](https://zh.wikipedia.org/wiki/%E8%A7%A6%E5%8F%91%E5%99%A8_(%E6%95%B0%E6%8D%AE%E5%BA%93))、[级联回滚](https://zh.wikipedia.org/w/index.php?title=%E7%BA%A7%E8%81%94%E5%9B%9E%E6%BB%9A&action=edit&redlink=1)等。
3. **隔离性** Isolation ：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
4. **持久性** Durability ：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

**列举 Spring 支持的事务管理类型？**

目前 Spring 提供两种类型的事务管理：

* **声明式**事务：通过使用注解或基于 XML 的配置事务，从而事务管理与业务代码分离。
* **编程式**事务：通过编码的方式实现事务管理，需要在代码中显式的调用事务的获得、提交、回滚。它为您提供极大的灵活性，但维护起来非常困难。

实际场景下，我们一般使用 Spring Boot + 注解的**声明式**事务。具体的示例，胖友可以看看 [《Spring Boot 事务注解详解》](https://www.jianshu.com/p/cddeca2c9245) 。

另外，也推荐看看 [《Spring 事务管理 － 编程式事务、声明式事务》](https://blog.csdn.net/xktxoo/article/details/77919508) 一文。

**Spring 事务如何和不同的数据持久层框架做集成？**

① 首先，我们先明确下，这里数据持久层框架，指的是 Spring JDBC、Hibernate、Spring JPA、MyBatis 等等。

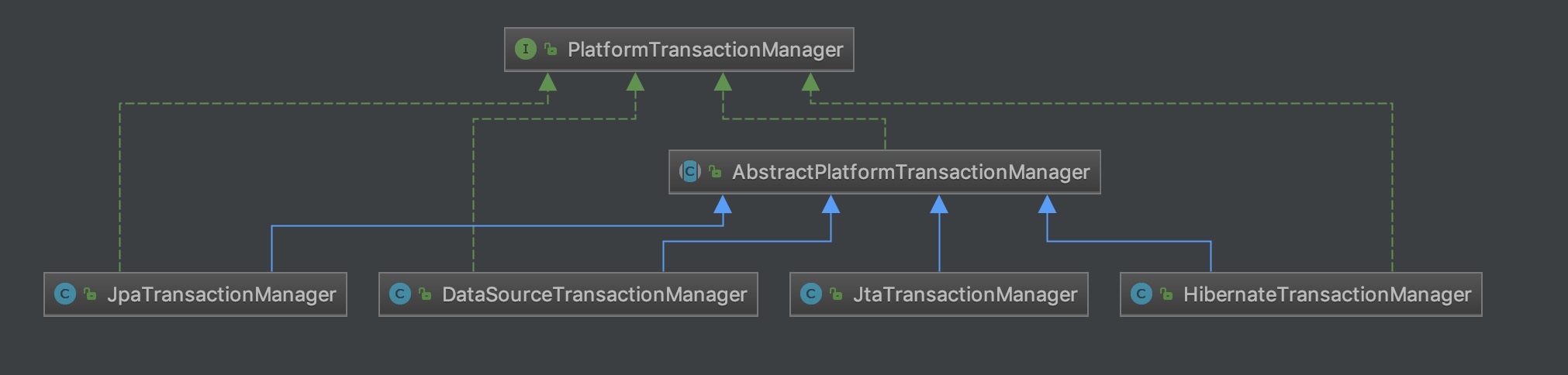
② 然后，Spring 事务的管理，是通过 org.springframework.transaction.PlatformTransactionManager 进行管理，定义如下：

|  |
| --- |
| // PlatformTransactionManager.java  public interface PlatformTransactionManager {   // 根据事务定义 TransactionDefinition ，获得 TransactionStatus 。   TransactionStatus getTransaction(@Nullable TransactionDefinition definition) throws TransactionException;   // 根据情况，提交事务  void commit(TransactionStatus status) throws TransactionException;    // 根据情况，回滚事务  void rollback(TransactionStatus status) throws TransactionException;   } |

* PlatformTransactionManager 是负责事务管理的接口，一共有三个接口方法，分别负责事务的获得、提交、回滚。
* #getTransaction(TransactionDefinition definition) 方法，根据事务定义 TransactionDefinition ，获得 TransactionStatus 。
  + 为什么不是创建事务呢？因为如果当前如果已经有事务，则不会进行创建，一般来说会跟当前线程进行绑定。如果不存在事务，则进行创建。
  + 为什么返回的是 TransactionStatus 对象？在 TransactionStatus 中，不仅仅包含事务属性，还包含事务的其它信息，例如是否只读、是否为新创建的事务等等。😈 下面，也会详细解析 TransactionStatus 。
  + 事务 TransactionDefinition 是什么？😈 下面，也会详细解析 TransactionStatus 。
* #commit(TransactionStatus status) 方法，根据 TransactionStatus 情况，提交事务。
  + 为什么根据 TransactionStatus 情况，进行提交？例如说，带@Transactional 注解的的 A 方法，会调用 @Transactional 注解的的 B 方法。
    - 在 B 方法结束调用后，会执行 PlatformTransactionManager#commit(TransactionStatus status) 方法，此处事务**是不能**、**也不会**提交的。
    - 而是在 A 方法结束调用后，执行 PlatformTransactionManager#commit(TransactionStatus status) 方法，提交事务。
* #rollback(TransactionStatus status) 方法，根据 TransactionStatus 情况，回滚事务。
  + 为什么根据 TransactionStatus 情况，进行回滚？原因同 #commit(TransactionStatus status) 方法。

③ 再之后，PlatformTransactionManager 有**抽象子**类 org.springframework.transaction.support.AbstractPlatformTransactionManager ，基于 [模板方法模式](https://blog.csdn.net/carson_ho/article/details/54910518) ，实现事务整体逻辑的骨架，而抽象 #doCommit(DefaultTransactionStatus status)、#doRollback(DefaultTransactionStatus status) 等等方法，交由子类类来实现。

前方高能，即将进入关键的 ④ 步骤。

④ 最后，不同的数据持久层框架，会有其对应的 PlatformTransactionManager 实现类，如下图所示：

* 所有的实现类，都基于 AbstractPlatformTransactionManager 这个骨架类。
* HibernateTransactionManager ，和 Hibernate5 的事务管理做集成。
* DataSourceTransactionManager ，和 JDBC 的事务管理做集成。所以，它也适用于 MyBatis、Spring JDBC 等等。
* JpaTransactionManager ，和 JPA 的事务管理做集成。

如下，是一个比较常见的 XML 方式来配置的事务管理器，使用的是 DataSourceTransactionManager 。代码如下：

|  |
| --- |
| <!-- 事务管理器 --> <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  <!-- 数据源 -->  <property name="dataSource" ref="dataSource" /> </bean> |

* 正如上文所说，它适用于 MyBatis、Spring JDBC 等等。

😈 是不是很有趣，更多详细的解析，可见如下几篇文章：

* [《精尽 Spring 源码分析 —— Transaction 源码简单导读》](http://svip.iocoder.cn/categories/Spring/)
* [《精尽 MyBatis 源码分析 —— 事务模块》](http://svip.iocoder.cn/MyBatis/transaction-package/)
* [《精尽 MyBatis 源码解析 —— Spring 集成（四）之事务》](http://svip.iocoder.cn/MyBatis/Spring-Integration-4/)

**为什么在 Spring 事务中不能切换数据源？**

做过 Spring 多数据源的胖友，都会有个惨痛的经历，为什么在开启事务的 Service 层的方法中，无法切换数据源呢？因为，在 Spring 的事务管理中，**所使用的数据库连接会和当前线程所绑定**，即使我们设置了另外一个数据源，使用的还是当前的数据源连接。

另外，多个数据源且需要事务的场景，本身会带来**多事务一致性**的问题，暂时没有特别好的解决方案。

所以一般一个应用，推荐除非了读写分离所带来的多数据源，其它情况下，建议只有一个数据源。并且，随着微服务日益身形，一个服务对应一个 DB 是比较常见的架构选择。

**@Transactional 注解有哪些属性？如何使用？**

@Transactional 注解的**属性**如下：

| **属性** | **类型** | **描述** |
| --- | --- | --- |
| value | String | 可选的限定描述符，指定使用的事务管理器 |
| propagation | enum: Propagation | 可选的事务传播行为设置 |
| isolation | enum: Isolation | 可选的事务隔离级别设置 |
| readOnly | boolean | 读写或只读事务，默认读写 |
| timeout | int (in seconds granularity) | 事务超时时间设置 |
| rollbackFor | Class对象数组，必须继承自Throwable | 导致事务回滚的异常类数组 |
| rollbackForClassName | 类名数组，必须继承自Throwable | 导致事务回滚的异常类名字数组 |
| noRollbackFor | Class对象数组，必须继承自Throwable | 不会导致事务回滚的异常类数组 |
| noRollbackForClassName | 类名数组，必须继承自Throwable | 不会导致事务回滚的异常类名字数组 |

* 一般情况下，我们直接使用 @Transactional 的所有属性默认值即可。

具体**用法**如下：

* @Transactional 可以作用于接口、接口方法、类以及类方法上。当作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。
* 虽然 @Transactional 注解可以作用于接口、接口方法、类以及类方法上，但是 Spring 建议不要在接口或者接口方法上使用该注解，因为这只有在使用基于接口的代理时它才会生效。另外， **@Transactional 注解应该只被应用到 public 方法上，这是由 Spring AOP 的本质决定的**。如果你在 protected、private 或者默认可见性的方法上使用 @Transactional 注解，这将被忽略，也不会抛出任何异常。**这一点，非常需要注意**。

下面，我们来简单说下**源码**相关的东西。

@Transactional 注解的属性，会解析成 org.springframework.transaction.TransactionDefinition 对象，即事务定义。TransactionDefinition 代码如下：

|  |
| --- |
| public interface TransactionDefinition {   int getPropagationBehavior(); // 事务的传播行为  int getIsolationLevel(); // 事务的隔离级别  int getTimeout(); // 事务的超时时间  boolean isReadOnly(); // 事务是否只读  @Nullable  String getName(); // 事务的名字  } |

* 可能会胖友有以后，@Transactional 注解的 rollbackFor、rollbackForClassName、noRollbackFor、noRollbackForClassName 属性貌似没体现出来？它们提现在 TransactionDefinition 的实现类 RuleBasedTransactionAttribute 中。
* #getPropagationBehavior() 方法，返回事务的**传播行为**，该值是个枚举，在下面来说。
* #getIsolationLevel() 方法，返回事务的**隔离级别**，该值是个枚举，在下面来说。

**什么是事务的隔离级别？分成哪些隔离级别？**

关于这个问题，涉及的内容会比较多，胖友直接看如下两篇文章：

* [《数据库四大特性以及事务隔离级别》](https://zhuanlan.zhihu.com/p/25419593)
* [《五分钟搞清楚 MySQL 事务隔离级别》](https://www.jianshu.com/p/4e3edbedb9a8)

另外，有一点非常重要，不同数据库对四个隔离级别的支持和实现略有不同。因为我们目前互联网主要使用 MySQL 为主，所以至少要搞懂 MySQL 对隔离级别的支持和实现情况。

在 TransactionDefinition 接口中，定义了“**四种**”的隔离级别枚举。代码如下：

|  |
| --- |
| // TransactionDefinition.java  /\*\*  \* 【Spring 独有】使用后端数据库默认的隔离级别  \*  \* MySQL 默认采用的 REPEATABLE\_READ隔离级别  \* Oracle 默认采用的 READ\_COMMITTED隔离级别  \*/ int ISOLATION\_DEFAULT = -1;  /\*\*  \* 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读  \*/ int ISOLATION\_READ\_UNCOMMITTED = Connection.TRANSACTION\_READ\_UNCOMMITTED;  /\*\*  \* 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生  \*/ int ISOLATION\_READ\_COMMITTED = Connection.TRANSACTION\_READ\_COMMITTED; /\*\*  \* 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。  \*/ int ISOLATION\_REPEATABLE\_READ = Connection.TRANSACTION\_REPEATABLE\_READ; /\*\*  \* 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。  \*  \* 但是这将严重影响程序的性能。通常情况下也不会用到该级别。  \*/ int ISOLATION\_SERIALIZABLE = Connection.TRANSACTION\_SERIALIZABLE; |

**什么是事务的传播级别？分成哪些传播级别？**

事务的**传播行为**，指的是当前带有事务配置的方法，需要怎么处理事务。

* 例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。
* 有一点需要注意，事务的传播级别，并不是数据库事务规范中的名词，**而是 Spring 自身所定义的**。通过事务的传播级别，Spring 才知道如何处理事务，是创建一个新事务呢，还是继续使用当前的事务。

艿艿的自我吐槽：是不是有种背概念背的想哭

在 TransactionDefinition 接口中，定义了**三类七种**传播级别。代码如下：

|  |
| --- |
| // TransactionDefinition.java  // ========== 支持当前事务的情况 ==========   /\*\*  \* 如果当前存在事务，则使用该事务。  \* 如果当前没有事务，则创建一个新的事务。  \*/ int PROPAGATION\_REQUIRED = 0; /\*\*  \* 如果当前存在事务，则使用该事务。  \* 如果当前没有事务，则以非事务的方式继续运行。  \*/ int PROPAGATION\_SUPPORTS = 1; /\*\*  \* 如果当前存在事务，则使用该事务。  \* 如果当前没有事务，则抛出异常。  \*/ int PROPAGATION\_MANDATORY = 2;  // ========== 不支持当前事务的情况 ==========   /\*\*  \* 创建一个新的事务。  \* 如果当前存在事务，则把当前事务挂起。  \*/ int PROPAGATION\_REQUIRES\_NEW = 3; /\*\*  \* 以非事务方式运行。  \* 如果当前存在事务，则把当前事务挂起。  \*/ int PROPAGATION\_NOT\_SUPPORTED = 4; /\*\*  \* 以非事务方式运行。  \* 如果当前存在事务，则抛出异常。  \*/ int PROPAGATION\_NEVER = 5;  // ========== 其他情况 ==========   /\*\*  \* 如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行。  \* 如果当前没有事务，则等价于 {@link TransactionDefinition#PROPAGATION\_REQUIRED}  \*/ int PROPAGATION\_NESTED = 6; |

* 分类之后，其实还是比较好记的。当然，绝大数场景，我们只用 PROPAGATION\_REQUIRED 传播级别。
* 这里需要指出的是，前面的六种事务传播行为是 Spring 从 EJB 中引入的，他们共享相同的概念。而 PROPAGATION\_NESTED 是 Spring 所特有的。
  + 以 PROPAGATION\_NESTED 启动的事务内嵌于外部事务中（如果存在外部事务的话），此时，内嵌事务并不是一个独立的事务，它依赖于外部事务的存在，只有通过外部的事务提交，才能引起内部事务的提交，嵌套的子事务不能单独提交。如果熟悉 JDBC 中的保存点（SavePoint）的概念，那嵌套事务就很容易理解了，其实嵌套的子事务就是保存点的一个应用，一个事务中可以包括多个保存点，每一个嵌套子事务。另外，外部事务的回滚也会导致嵌套子事务的回滚。
  + 😈 当然，虽然上面 PROPAGATION\_NESTED 文字很长，实际我们基本没用过。或者说，去掉基本，我们根本没用过。

**什么是事务的超时属性？**

所谓事务超时，就是指一个事务所允许执行的最长时间，如果超过该时间限制但事务还没有完成，则自动回滚事务。

在 TransactionDefinition 中以 int 的值来表示超时时间，其单位是秒。

当然，这个属性，貌似我们基本也没用过。

**什么是事务的只读属性？**

事务的只读属性是指，对事务性资源进行只读操作或者是读写操作。

* 所谓事务性资源就是指那些被事务管理的资源，比如数据源、JMS 资源，以及自定义的事务性资源等等。
* 如果确定只对事务性资源进行只读操作，那么我们可以将事务标志为只读的，以提高事务处理的性能。感兴趣的胖友，可以看看 [《不使用事务和使用只读事务的区别 》](https://my.oschina.net/uniquejava/blog/80954) 。

在 TransactionDefinition 中以 boolean 类型来表示该事务是否只读。

**什么是事务的回滚规则？**

回滚规则，定义了哪些异常会导致事务回滚而哪些不会。

* 默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚（这一行为与EJB的回滚行为是一致的）。
* 但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

注意，事务的回滚规则，并不是数据库事务规范中的名词，**而是 Spring 自身所定义的**。

**简单介绍 TransactionStatus ？**

艿艿：这个可能不是一个面试题，主要满足下大家的好奇心。

TransactionStatus 接口，记录事务的状态，不仅仅包含事务本身，还包含事务的其它信息。代码如下：

|  |
| --- |
| // TransactionStatus.java  public interface TransactionStatus extends SavepointManager, Flushable {   /\*\*  \* 是否是新创建的事务  \*/  boolean isNewTransaction();   /\*\*  \* 是否有 Savepoint  \*  \* 在 {@link TransactionDefinition#PROPAGATION\_NESTED} 传播级别使用。  \*/  boolean hasSavepoint();   /\*\*  \* 设置为只回滚  \*/  void setRollbackOnly();  /\*\*  \* 是否为只回滚  \*/  boolean isRollbackOnly();   /\*\*  \* 执行 flush 操作  \*/  @Override  void flush();   /\*\*  \* 是否事务已经完成  \*/  boolean isCompleted();  } |

* 为什么没有事务对象呢？在 TransactionStatus 的实现类 DefaultTransactionStatus 中，有个 Object transaction 属性，表示事务对象。
* #isNewTransaction() 方法，表示是否是新创建的事务。有什么用呢？答案结合 [「Spring 事务如何和不同的数据持久层框架做集成？」](http://yudaoyuanma:ydym1024@svip.iocoder.cn/Spring/Interview/) 问题，我们对 #commit(TransactionStatus status) 方法的解释。通过该方法，我们可以判断，当前事务是否当前方法所创建的，只有创建事务的方法，**才能且应该真正的提交事务**。

**使用 Spring 事务有什么优点？**

1. 通过 PlatformTransactionManager ，为不同的数据层持久框架提供统一的 API ，无需关心到底是原生 JDBC、Spring JDBC、JPA、Hibernate 还是 MyBatis 。
2. 通过使用声明式事务，使业务代码和事务管理的逻辑分离，更加清晰。

从倾向上来说，艿艿比较喜欢**注解** + 声明式事务。

**Spring Data Access**

艿艿：这块的问题，感觉面试问的不多，至少我很少问。哈哈哈。就当做下了解，万一问了呢。

**Spring 支持哪些 ORM 框架？**

* Hibernate
* JPA
* MyBatis
* [JDO](https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch13s04.html)
* [OJB](https://db.apache.org/ojb/docu/howtos/howto-use-spring.html)

可能会有胖友说，不是应该还有 Spring JDBC 吗。注意，Spring JDBC 不是 ORM 框架。

**在 Spring 框架中如何更有效地使用 JDBC ？**

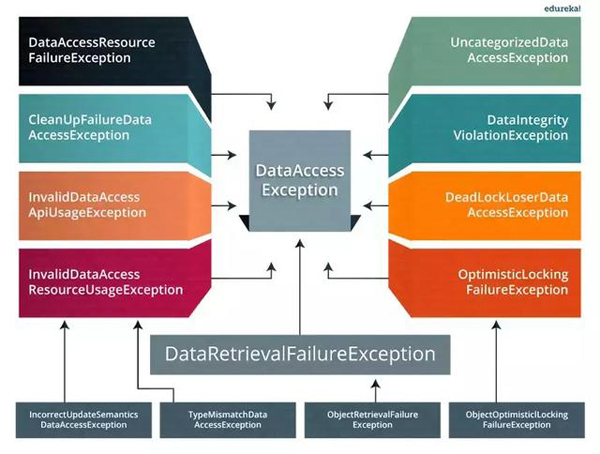
Spring 提供了 Spring JDBC 框架，方便我们使用 JDBC 。

对于开发者，只需要使用 JdbcTemplate 类，它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。

没有使用过的胖友，可以看看 [《Spring JDBC 访问关系型数据库》](https://www.tianmaying.com/tutorial/spring-jdbc-data-accessing) 文章。

**Spring 数据数据访问层有哪些异常？**

通过使用 Spring 数据数据访问层，它统一了各个数据持久层框架的不同异常，统一进行提供 org.springframework.dao.DataAccessException 异常及其子类。如下图所示：



**使用 Spring 访问 Hibernate 的方法有哪些？**

艿艿：这个问题很灵异，因为艿艿已经好久不使用 Hibernate 了，所以答案是直接复制的。

我们可以通过两种方式使用 Spring 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。

艿艿：不过我记得，12 年我用过 Spring JPA 的方式，操作 Hibernate 。具体可参考 [《一起来学 SpringBoot 2.x | 第六篇：整合 Spring Data JPA》](http://www.iocoder.cn/Spring-Boot/battcn/v2-orm-jpa/) 。

当然，我们可以再来看一道 [《JPA 规范与 ORM 框架之间的关系是怎样的呢？》](https://www.cnblogs.com/xiaoheike/p/5150553.html) 。这个问题，我倒是问过面试的候选人，哈哈哈哈。

# 精尽 Spring 学习指南

## 1. 视频

😈 记得艿艿当年学习 Spring ，还是看的马士兵老师的。有握爪的同学么？嘻嘻。

* 布丁狼 [《Java 大神之路（第八季 Spring5）》](https://www.bilibili.com/video/av33077509) 一共有 57P 。声音有点小…
* 尚硅谷 [《Spring4 视频教程》](https://www.bilibili.com/video/av21335209) 一共有 36P 。
* 传智播客 [《Spring3 框架》](https://www.imooc.com/learn/498) 一共有 87P 。
* moocer [《Spring 入门篇》](http://www.imooc.com/learn/196) 一共 7 小时，侧重 Spring IoC 和 AOP 。
* apollo\_0001 [《探秘 Spring AOP》](http://www.imooc.com/learn/869) 一共 2 小时 30 分钟，侧重 Spring AOP 。
* 慕\_神 [《Spring 事务管理》](http://www.imooc.com/learn/478) 一共 2 小时，侧重 Spring Transaction 。

上述的视频，是艿艿翻了下目前大家看的比较多的视频，利益无关哈。

虽然现在 Spring Boot 给我们带来了非常便利的搭建开发环境的方式，但是艿艿的态度，希望大家能对 Spring 本身能够有比较深入的理解。知其然，知其所以然。这是技术的态度，也是让我们能够走的更远的基础。

## 2. 书籍

[《Spring 书单整理》](http://www.iocoder.cn/Books/Netty-Spring-recommended)

* [《Spring 实战（第4版）》](https://item.jd.com/11899370.html) 豆瓣评分：8.3 ，艿艿当年入门 Spring 的书，翻烂了。
* [《Spring 揭秘》](https://union-click.jd.com/jdc?d=kYry5g) 豆瓣评分：9.0 ，zhisheng 学习 Spring 的书，虽然 Spring 版本比较老，但是不妨碍我们理解 Spring 。
* 陈雄华/林开雄/文建国 [《精通Spring 4.x》](https://book.douban.com/subject/26952826/) 豆瓣评分 8.2 分，很厚，799 页。
* [《Spring 源码深度解析》](https://item.jd.com/11311737.html) 豆瓣评分：6.6 ，艿艿当时撸 Spring 源码，翻了好几遍，认为评分低了，还是很值得推荐的。
* [《Spring 技术内幕》](https://item.jd.com/1027598917.html) 豆瓣评分 5.8 ，也是 Spring 源码相关，艿艿当年也买过，没读完~

## 3. 文章

* [《Spring 框架的设计理念与设计模式分析》](https://www.ibm.com/developerworks/cn/java/j-lo-spring-principle/index.html)
* [《超全面 Spring 复习总结笔记》](https://www.jianshu.com/p/15a0f94b4063)

艿艿说：推荐这篇文章的原因不是这篇文章写的多好，而是希望我们，在每次之后，我们要去整理总结，即使是简单的笔记，也值得输出。

生活的节奏很快，学习之后，总会淡忘。通过文字，能让我们更快的回忆起来。

* [《彻底征服 Spring AOP 之理论篇》](https://segmentfault.com/a/1190000007469968)
* [《彻底征服 Spring AOP 之实战篇》](https://segmentfault.com/a/1190000007469982)
* [《可能是最漂亮的 Spring 事务管理详解》](https://mp.weixin.qq.com/s?__biz=MzUzMTA2NTU2Ng==&mid=2247484702&idx=1&sn=c04261d63929db09ff6df7cadc7cca21&chksm=fa497aafcd3ef3b94082da7bca841b5b7b528eb2a52dbc4eb647b97be63a9a1cf38a9e71bf90&token=165108535&lang=zh_CN#rd)
* [《Spring 事务管理 － 编程式事务、声明式事务》](https://blog.csdn.net/xktxoo/article/details/77919508)

如果胖友有看到合适的文章，也可以推荐给我哟。

**精尽 Spring 源码分析 —— 调试环境搭建**

## 1. 依赖工具

* Gradle
* Git
* JDK1.8+
* IntelliJ IDEA

笔者目前使用的系统版本是 macOS Mojave 10.14 。所以，如果胖友是 Windows 环境，胖到一些问题，请在星球给我留言。

懵逼的艿艿：根据现在收到的信息，貌似主要是 Windows 环境会搭建失败。如果胖友真的搭建不起来，建议可以先新建一个项目，搭建一个 Spring Demo 来调试。

😈 总之，我们的目的是，一定一定一定要调试。酱紫，才能更好的阅读 Spring 的代码。

另外，本文参考官方提供的文档 [《import-into-idea》](https://github.com/spring-projects/spring-framework/blob/master/import-into-idea.md) 。

补充说明 1 ：IntelliJ IDEA 请使用 2018 版本，之前有胖友反馈搭建不起来，因为 IDEA 版本过低。

## 2. 源码拉取

从官方仓库 <https://github.com/spring-projects/spring-framework> Fork 出属于自己的仓库。

* 为什么要 Fork ？既然开始阅读、调试源码，我们可能会写一些注释，有了自己的仓库，可以进行自由的提交。😈
* 本文使用的 Spring 版本为 5.1.1.BUILD-SNAPSHOT 。
* 使用 IntelliJ IDEA 从 Fork 出来的仓库拉取代码。因为 Spring 项目比较大，从仓库中拉取代码的时间会比较长。

拉取完成后，Gradle 会开始自动 **Build** 项目。因为 Build 的过程中，会下载非常多的依赖，请耐心等待。

* 😈 不过笔者有点不太确定，Gradle 是否会自动 **Build** 项目，反正我的会。如果此处碰到问题，请给我留言。

## 3. 预编译 spring-oxm 项目

打开 IDEA Terminal ，输入如下命令，预编译 spring-oxm 项目：

|  |
| --- |
| gradle.bat cleanIdea :spring-oxm:compileTestJava |

当看到 BUILD SUCCESSFUL ，说明编译成功。

😈 另外，笔者有点不确定，Gradle 在上面已经自动。

感谢【蝴蝶落于指尖】同学，经过测试，这个是必须的操作。

## 4. 运行示例

在 spring-context 项目中的 src/test/java/example 目录下，已经提供了一些示例。

① 解析 XML 配置文件成对应的 BeanDefinition 们的流程

可调试 org.springframework.beans.factory.xml.XmlBeanDefinitionReaderTests 的 #withFreshInputStream() 和 #withImport() 这两个单元测试。

相比来说，后者比前者多了一个 <import /> 标签的解析。当然，XmlBeanDefinitionReaderTests 类中，其它方法也可以简单调试下。看胖友的兴趣哈。

* #factorySingleton() 方法，单例的 FactoryBean 的单元测试。

② 加载 Bean 的流程

可调试 org.springframework.beans.factory.xml.AbstractBeanFactoryTests 这个单元测试类里的方法。

实际上，AbstractBeanFactoryTests 是一个**抽象**类，所以在运行时，需要选择对应的子类，例如 XmlListableBeanFactoryTests 类。

③ ClassPathXmlApplicationContext 的流程

可调试 org.springframework.context.support.ClassPathXmlApplicationContextTests 这个单元测试类里的方法。例如 #testResourceAndInputStream() 方法。

④ 解析 Properties 配置文件成对应的 BeanDefinition 们的流程

选读，实际使用非常少。主要目的是为了更深入的理解 BeanDefinitionReader 的设计。

**可调试** org.springframework.beans.factory.support.PropertiesBeanDefinitionReaderTests 这个单元测试里的方法。

另外，也推荐阅读下 [《spring beans源码解读之 – BeanDefinition 解析器》](https://www.bbsmax.com/A/KE5QLe8PJL/) 一文。

⑤ 调试 Spring AOP 相关的流程

参见 [《精尽 Spring 源码分析 —— AOP 源码简单导读》](http://svip.iocoder.cn/Spring/aop-simple-intro)

⑥ 调试 Spring Transaction 相关的流程

参见 [《精尽 Spring 源码分析 —— Transaction 源码简单导读》](http://svip.iocoder.cn/Spring/transaction-simple-intro)

⑦ 调试 Spring MVC 相关的流程

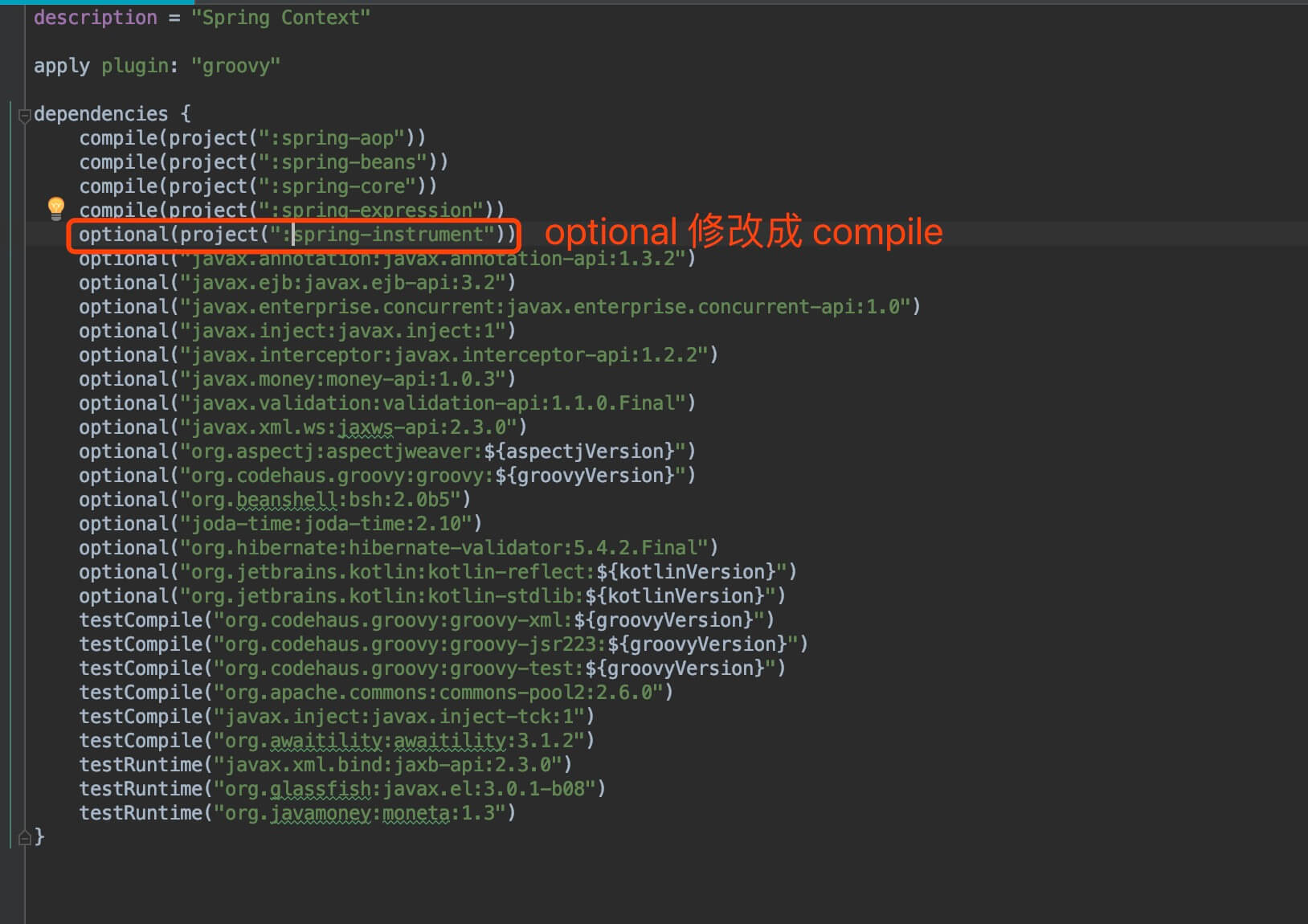
参见 [《精尽 Spring MVC 源码分析 —— 调试环境搭建》](http://svip.iocoder.cn/Spring-MVC/build-debugging-environment)

⑧ TODO 芋艿，补充一些推荐阅读的示例

## 5. 可能碰到的问题

### 5.1 报 InstrumentationSavingAgent 不存在的错误

例如说，在运行 spring-context 项目中的单元测试时，会报 InstrumentationSavingAgent 存在的错误。此时，我们将 spring-context.gradle 修改如下：



修改完成后，Gradle 又会自动 **Build** 项目，下载相关依赖。完成后，再次运行 spring-context 项目中的单元测试，顺利通过。

根据胖友【Tomy\_Rich】测试的说法，凡是报 XXX 不存在的，就和 InstrumentationSavingAgent 的处理方式一样就可以了。

### 5.2 报 ‘io.spring.dependency-management’ 插件不存在

可参考 [《Plugin [id: ‘io.spring.dependency-management’, version: ‘1.0.5.RELEASE’, apply: false] was not found in any of the following sources:》](https://stackoverflow.com/questions/53587243/plugin-id-io-spring-dependency-management-version-1-0-5-release-apply) 一文进行解决。

来自胖友【贾鹤鸣】的提供。

### 5.3 其它

如果胖友你在搭建调试环境的过程中，如果碰到任何问题，可以在星球给我留言。

## 666. 彩蛋

笔者开始更新 Spring 源码解析系列，让我们在 2019 一起**精尽** Spring 。

另外，笔者的好基友小明哥，已经在死磕 Spring 源码，并更新相应的文章。所以对于这个系列，如果小明哥已经写了，会直接进行引用。感谢小明哥的文章授权。美滋滋。

那么，就开始干吧。

重要的友情提示一：

《Spring 源码深度解析》，基于 Spring 3 版本的源码解析。虽然版本有点老，但是内容的流畅性很不错，特别是 Spring IoC 部分。艿艿自己也阅读了一遍，点赞。

# 1.【死磕 Spring】—— IoC 之深入理解 Spring IoC

**本文主要基于 Spring 5.0.6.RELEASE**

摘要: 原创出处 <http://cmsblogs.com/?p=2652> 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老艿艿」略作修改，记录在理解过程中，参考的资料。

在一开始学习 Spring 的时候，我们就接触 IoC 了，作为 Spring 第一个最核心的概念，我们在解读它源码之前势必需要对其有深入的认识，本篇为【死磕 Spring】系列博客的第一篇博文，主要介绍 IoC 基本概念和各个组件。

## 1. IoC 理论

IoC 全称为 Inversion of Control，翻译为 “控制反转”，它还有一个别名为 DI（Dependency Injection）,即依赖注入。

如何理解“控制反转”好呢？理解好它的关键在于我们需要回答如下四个问题：

1. 谁控制谁
2. 控制什么
3. 为何是反转
4. 哪些方面反转了

在回答这四个问题之前，我们先看 IoC 的定义：

**所谓 IoC ，就是由 Spring IoC 容器来负责对象的生命周期和对象之间的关系**

上面这句话是整个 IoC 理论的核心。如何来理解这句话？我们引用一个例子来走阐述（看完该例子上面四个问题也就不是问题了）。

找女朋友，一般情况下我们是如何来找女朋友的呢？首先我们需要根据自己的需求（漂亮、身材好、性格好）找一个妹子，然后到处打听她的兴趣爱好、微信、电话号码，然后各种投其所好送其所要，最后追到手。如下：

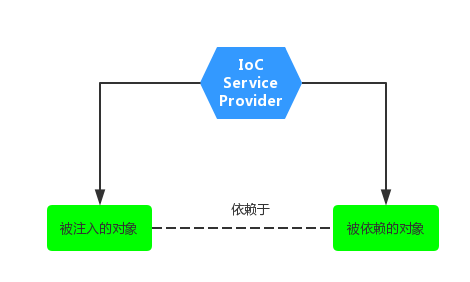
|  |
| --- |
| /\*\*  \* 年轻小伙子  \*/ public class YoungMan {   private BeautifulGirl beautifulGirl;   YoungMan(){   // 可能你比较牛逼，指腹为婚   // beautifulGirl = new BeautifulGirl();   }   public void setBeautifulGirl(BeautifulGirl beautifulGirl) {   this.beautifulGirl = beautifulGirl;   }   public static void main(String[] args){   YoungMan you = new YoungMan();   BeautifulGirl beautifulGirl = new BeautifulGirl("你的各种条件");   beautifulGirl.setxxx("各种投其所好");   // 然后你有女票了   you.setBeautifulGirl(beautifulGirl);   }  } |

这就是我们通常做事的方式，如果我们需要某个对象，一般都是采用这种直接创建的方式(new BeautifulGirl())，这个过程复杂而又繁琐，而且我们必须要面对每个环节，而且使用完成之后我们还要复杂销毁它，这种情况下我们的对象与它所依赖的对象耦合在一起。

其实我们需要思考一个问题？我们每次用到自己依赖的对象真的需要自己去创建吗？我们知道，我们依赖对象其实并不是依赖该对象本身，而是依赖它所提供的服务，只要在我们需要它的时候，它能够及时提供服务即可，至于它是我们主动去创建的还是别人送给我们的，其实并不是那么重要。再说了，相比于自己千辛万苦去创建它还要管理善后而言，直接有人送过来是不是显得更加好呢？

这个给我们送东西的“人” 就是 IoC，在上面的例子中，它就相当于一个婚介公司，作为一个婚介公司它管理着很多男男女女的资料，当我们需要一个女朋友的时候，直接跟婚介公司提出我们的需求，婚介公司则会根据我们的需求提供一个妹子给我们，我们只需要负责谈恋爱，生猴子就行了。你看，这样是不是很简单明了。

诚然，作为婚介公司的 IoC 帮我们省略了找女朋友的繁杂过程，将原来的主动寻找变成了现在的被动接受，更加简洁轻便。你想啊，原来你还得鞍马前后，各种巴结，什么东西都需要自己去亲力亲为，现在好了，直接有人把现成的送过来，多么美妙的事情啊。所以，简单点说，IoC 的理念就是**让别人为你服务**，如下图（摘自Spring揭秘）：



在没有引入 IoC 的时候，被注入的对象直接依赖于被依赖的对象，有了 IoC 后，两者及其他们的关系都是通过 Ioc Service Provider 来统一管理维护的。被注入的对象需要什么，直接跟 IoC Service Provider 打声招呼，后者就会把相应的被依赖对象注入到被注入的对象中，从而达到 IoC Service Provider 为被注入对象服务的目的。**所以 IoC 就是这么简单！原来是需要什么东西自己去拿，现在是需要什么东西让别人（IoC Service Provider）送过来**

现在在看上面那四个问题，答案就显得非常明显了:

1. **谁控制谁**：在传统的开发模式下，我们都是采用直接 new 一个对象的方式来创建对象，也就是说你依赖的对象直接由你自己控制，但是有了 IoC 容器后，则直接由 IoC 容器来控制。所以“谁控制谁”，当然是 IoC 容器控制对象
2. **控制什么**：控制对象。
3. **为何是反转**：没有 IoC 的时候我们都是在自己对象中主动去创建被依赖的对象，这是正转。但是有了 IoC 后，所依赖的对象直接由 IoC 容器创建后注入到被注入的对象中，依赖的对象由原来的主动获取变成被动接受，所以是反转。
4. **哪些方面反转了**：所依赖对象的获取被反转了。

妹子有了，但是如何拥有妹子呢？这也是一门学问。

1. 可能你比较牛逼，刚刚出生的时候就指腹为婚了。
2. 大多数情况我们还是会考虑自己想要什么样的妹子，所以还是需要向婚介公司打招呼的。
3. 还有一种情况就是，你根本就不知道自己想要什么样的妹子，直接跟婚介公司说，我就要一个这样的妹子。

### 1.1 注入形式

所以，IoC Service Provider 为被注入对象提供被依赖对象也有如下几种方式：构造方法注入、stter方法注入、接口注入。

**① 构造器注入**

构造器注入，顾名思义就是被注入的对象通过在其构造方法中声明依赖对象的参数列表，让外部知道它需要哪些依赖对象。

|  |
| --- |
| YoungMan(BeautifulGirl beautifulGirl) {  this.beautifulGirl = beautifulGirl; } |

构造器注入方式比较直观，对象构造完毕后就可以直接使用，这就好比你出生你家里就给你指定了你媳妇。

**② setter 方法注入**

对于 JavaBean 对象而言，我们一般都是通过 getter 和 setter 方法来访问和设置对象的属性。所以，当前对象只需要为其所依赖的对象提供相对应的 setter 方法，就可以通过该方法将相应的依赖对象设置到被注入对象中。如下：

|  |
| --- |
| public class YoungMan {   private BeautifulGirl beautifulGirl;   public void setBeautifulGirl(BeautifulGirl beautifulGirl) {  this.beautifulGirl = beautifulGirl;  }  } |

相比于构造器注入，setter 方式注入会显得比较宽松灵活些，它可以在任何时候进行注入（当然是在使用依赖对象之前），这就好比你可以先把自己想要的妹子想好了，然后再跟婚介公司打招呼，你可以要林志玲款式的，赵丽颖款式的，甚至凤姐哪款的，随意性较强。

**③ 接口方式注入**

接口方式注入显得比较霸道，因为它需要被依赖的对象实现不必要的接口，带有侵入性。一般都不推荐这种方式。

感兴趣的胖友，可以看看 [《依赖注入的三种实现形式 —— 接口注入（Interface Injection）》](http://wiki.jikexueyuan.com/project/spring-ioc/iocordi-1.html#6e5dfcd838f3a79e9129641785cf736f)

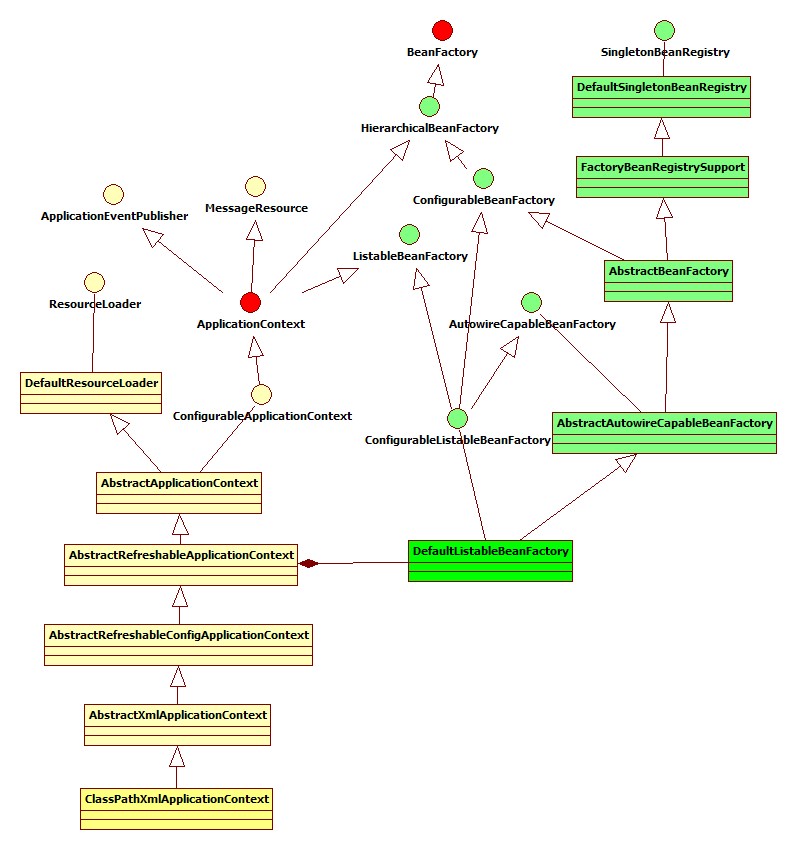
### 1.2 推荐文章

关于 IoC 理论部分，笔者不在阐述，这里推荐几篇博客阅读：

* [《谈谈对 Spring IoC 的理解》](http://www.cnblogs.com/xdp-gacl/p/4249939.html)
* [《Spring 的 IoC 原理[通俗解释一下]》](https://blog.csdn.net/m13666368773/article/details/7802126)
* [《Spring IoC 原理（看完后大家可以自己写一个spring）》](https://blog.csdn.net/it_man/article/details/4402245)

## 2. 各个组件

先看下图（摘自:<http://singleant.iteye.com/blog/1177358）>

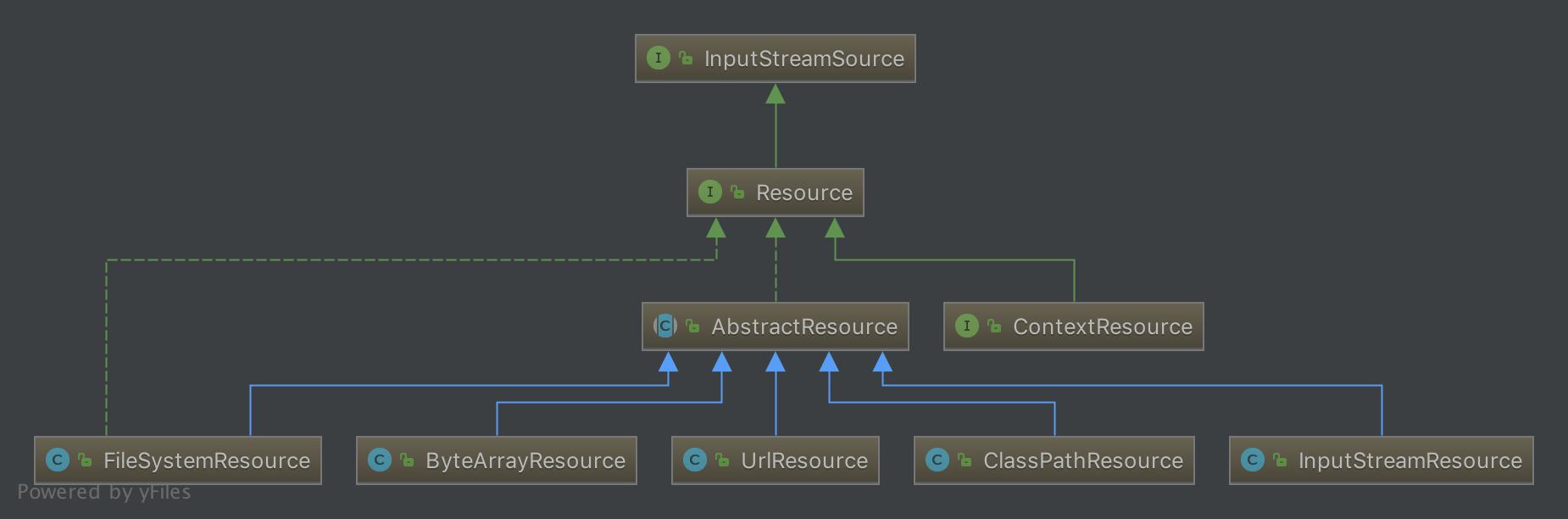


该图为 ClassPathXmlApplicationContext 的类继承体系结构，虽然只有一部分，但是它基本上包含了 IoC 体系中大部分的核心类和接口。

下面我们就针对这个图进行简单的拆分和补充说明。

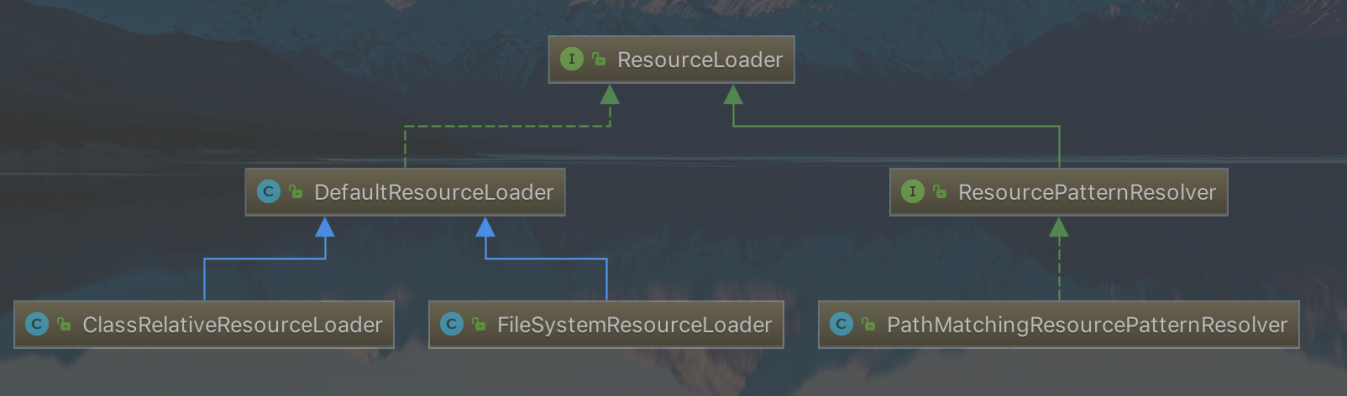
### 2.1 Resource 体系

org.springframework.core.io.Resource，对资源的抽象。它的每一个实现类都代表了一种资源的访问策略，如 ClassPathResource、RLResource、FileSystemResource 等。



#### 2.1.2 ResourceLoader 体系

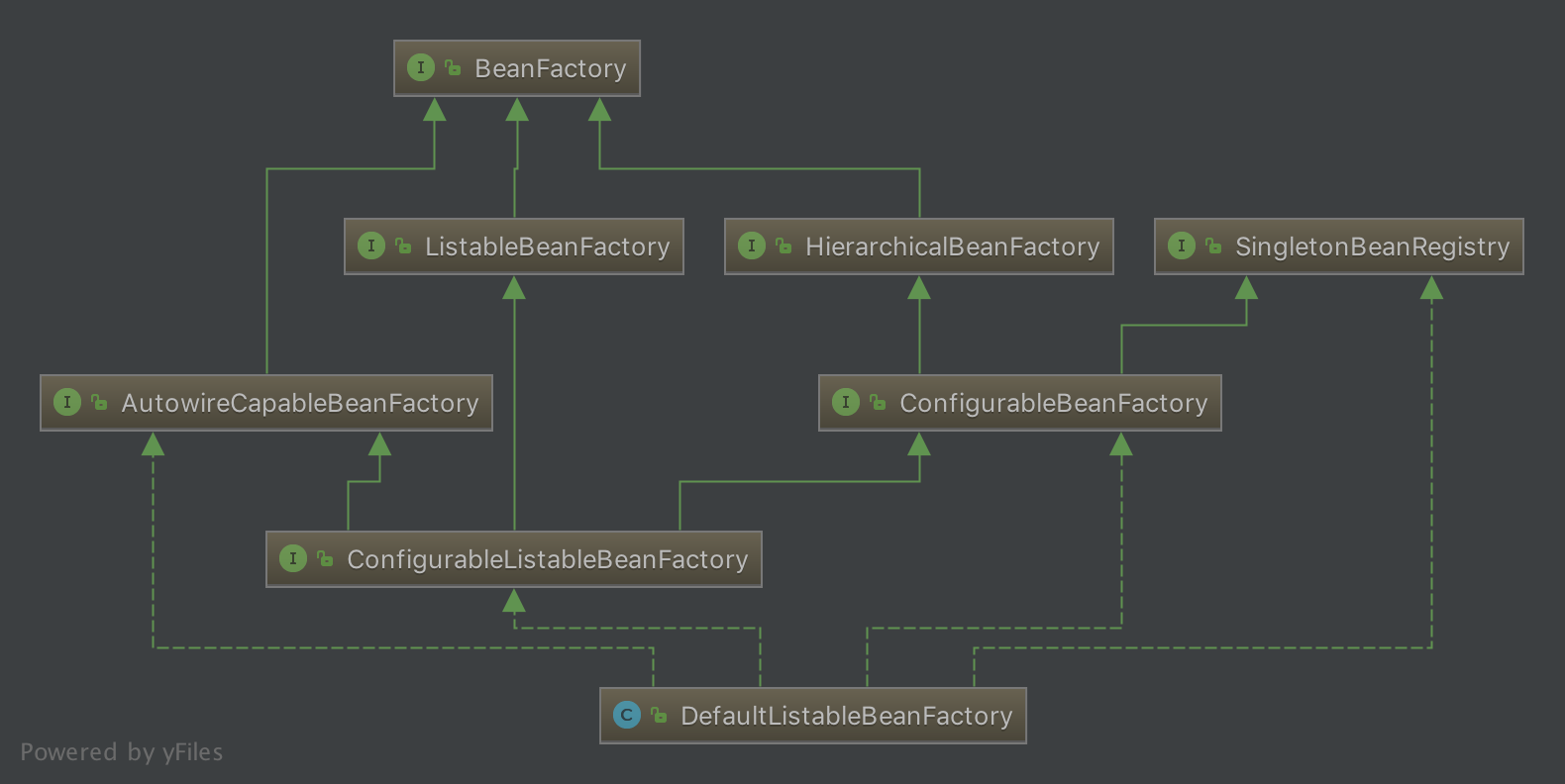
有了资源，就应该有资源加载，Spring 利用 org.springframework.core.io.ResourceLoader 来进行统一资源加载，类图如下：



关于 Resource 和 ResourceLoader 的源码解析，见 [《【死磕 Spring】—— IoC 之 Spring 统一资源加载策略》](http://svip.iocoder.cn/Spring/IoC-load-Resource) 。

### 2.2 BeanFactory 体系

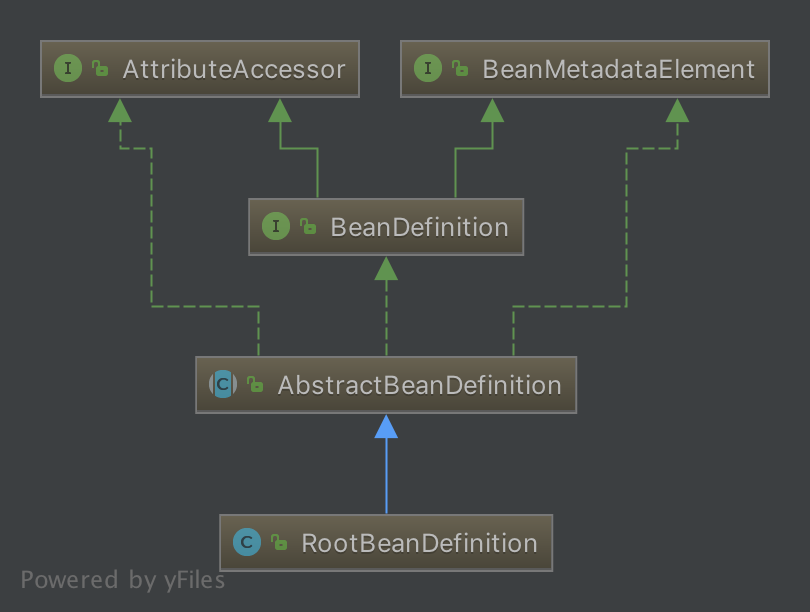
org.springframework.beans.factory.BeanFactory，是一个非常纯粹的 bean 容器，它是 IoC 必备的数据结构，其中 BeanDefinition 是它的基本结构。BeanFactory 内部维护着一个BeanDefinition map ，并可根据 BeanDefinition 的描述进行 bean 的创建和管理。



* BeanFactory 有三个直接子类 ListableBeanFactory、HierarchicalBeanFactory 和 AutowireCapableBeanFactory 。
* DefaultListableBeanFactory 为最终默认实现，它实现了所有接口。

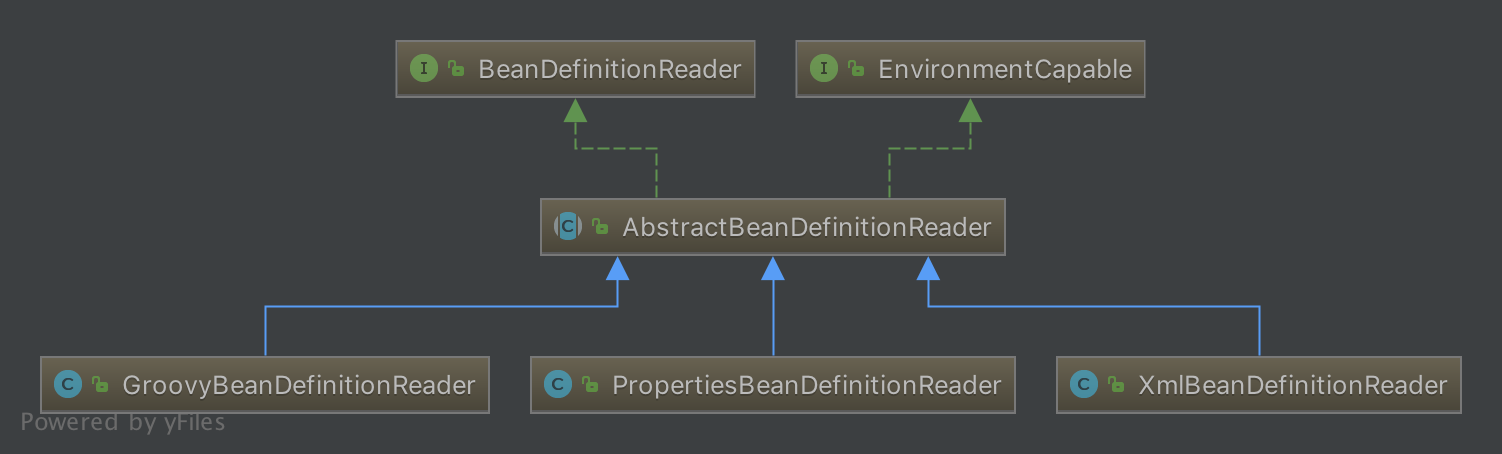
### 2.3 BeanDefinition 体系

org.springframework.beans.factory.config.BeanDefinition ，用来描述 Spring 中的 Bean 对象。



### 2.4 BeanDefinitionReader 体系

org.springframework.beans.factory.support.BeanDefinitionReader 的作用是读取 Spring 的配置文件的内容，并将其转换成 Ioc 容器内部的数据结构 ：BeanDefinition 。



关于 BeanDefinitionReader 的源码解析，见如下文章：

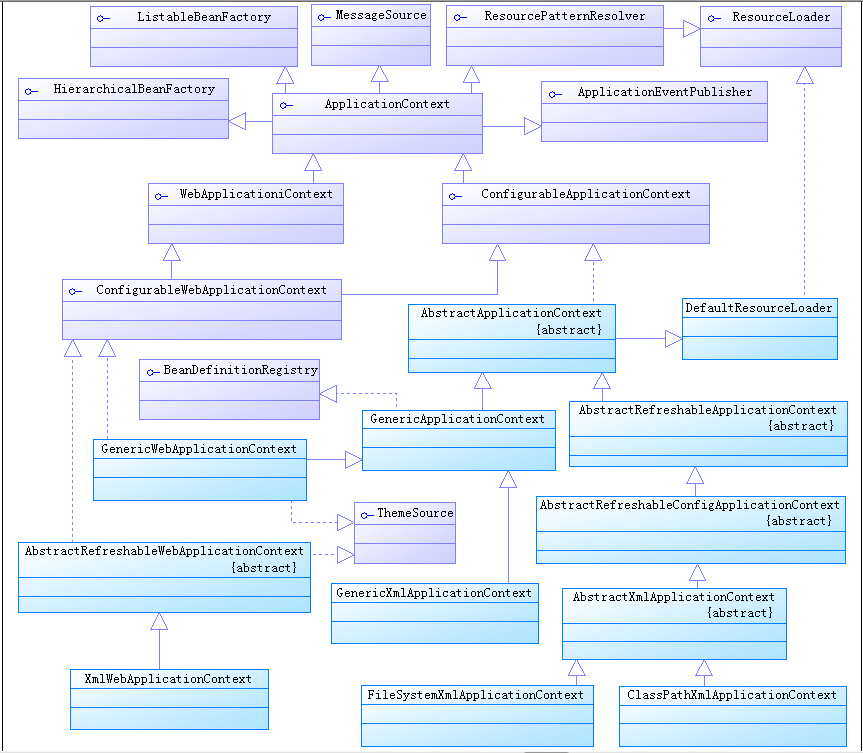
* [《【死磕 Spring】—— IoC 之加载 Definitions》](http://svip.iocoder.cn/Spring/IoC-load-BeanDefinitions)
* [《【死磕 Spring】—— IoC 之获取验证模型》](http://svip.iocoder.cn/Spring/IoC-Validation-Mode-For-Resource)
* [《【死磕 Spring】—— IoC 之获取 Document 对象》](http://svip.iocoder.cn/Spring/IoC-load-Document)
* [《【死磕 Spring】—— IoC 之注册 BeanDefinitions》](http://svip.iocoder.cn/Spring/IoC-register-BeanDefinitions)

### 2.5 ApplicationContext 体系

org.springframework.context.ApplicationContext ，这个就是大名鼎鼎的 Spring 容器，它叫做应用上下文，与我们应用息息相关。它继承 BeanFactory ，所以它是 BeanFactory 的扩展升级版，如果BeanFactory 是屌丝的话，那么 ApplicationContext 则是名副其实的高富帅。由于 ApplicationContext 的结构就决定了它与 BeanFactory 的不同，其主要区别有：

1. 继承 org.springframework.context.MessageSource 接口，提供国际化的标准访问策略。
2. 继承 org.springframework.context.ApplicationEventPublisher 接口，提供强大的**事件**机制。
3. 扩展 ResourceLoader ，可以用来加载多种 Resource ，可以灵活访问不同的资源。
4. 对 Web 应用的支持。

下图来源：<https://blog.csdn.net/yujin753/article/details/47043143>



### 2.6 小结

上面五个体系可以说是 Spring IoC 中最核心的部分，后面博文也是针对这五个部分进行源码分析。其实 IoC 咋一看还是挺简单的，无非就是将配置文件（暂且认为是 xml 文件）进行解析（分析 xml 谁不会啊），然后放到一个 Map 里面就差不多了，初看有道理，其实要面临的问题还是有很多的，下面就劳烦各位看客跟着 LZ 博客来一步一步揭开 Spring IoC 的神秘面纱。

**此系列博文为 LZ 学习、研究 Spring 机制和源码的学习笔记，会涉及参考别人的博文和书籍内容，如有雷同，纯属借鉴，当然 LZ 会标明参考来源。同时由于知识面和能力的问题，文章中难免会出现错误之处，如有，望各位大佬指出，不胜感激**。

另外，通过上面五个体系，我们可以看出，IoC 主要由 spring-beans 和 spring-context 项目，进行实现。

# 【死磕 Spring】—— IoC 之 Spring 统一资源加载策略

**本文主要基于 Spring 5.0.6.RELEASE**

摘要: 原创出处 <http://cmsblogs.com/?p=2656> 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老艿艿」略作修改，记录在理解过程中，参考的资料。

在学 Java SE 的时候，我们学习了一个标准类 java.net.URL，该类在 Java SE 中的定位为统一资源定位器（Uniform Resource Locator），但是我们知道它的实现基本只限于网络形式发布的资源的查找和定位。然而，实际上资源的定义比较广泛，除了网络形式的资源，还有以二进制形式存在的、以文件形式存在的、以字节流形式存在的等等。而且它可以存在于任何场所，比如网络、文件系统、应用程序中。所以 java.net.URL 的局限性迫使 Spring 必须实现自己的资源加载策略，该资源加载策略需要满足如下要求：

1. 职能划分清楚。资源的定义和资源的加载应该要有一个清晰的**界限**；
2. 统一的抽象。统一的资源**定义**和资源加载**策略**。资源加载后要返回统一的抽象给客户端，客户端要对资源进行怎样的处理，应该由抽象资源接口来界定。

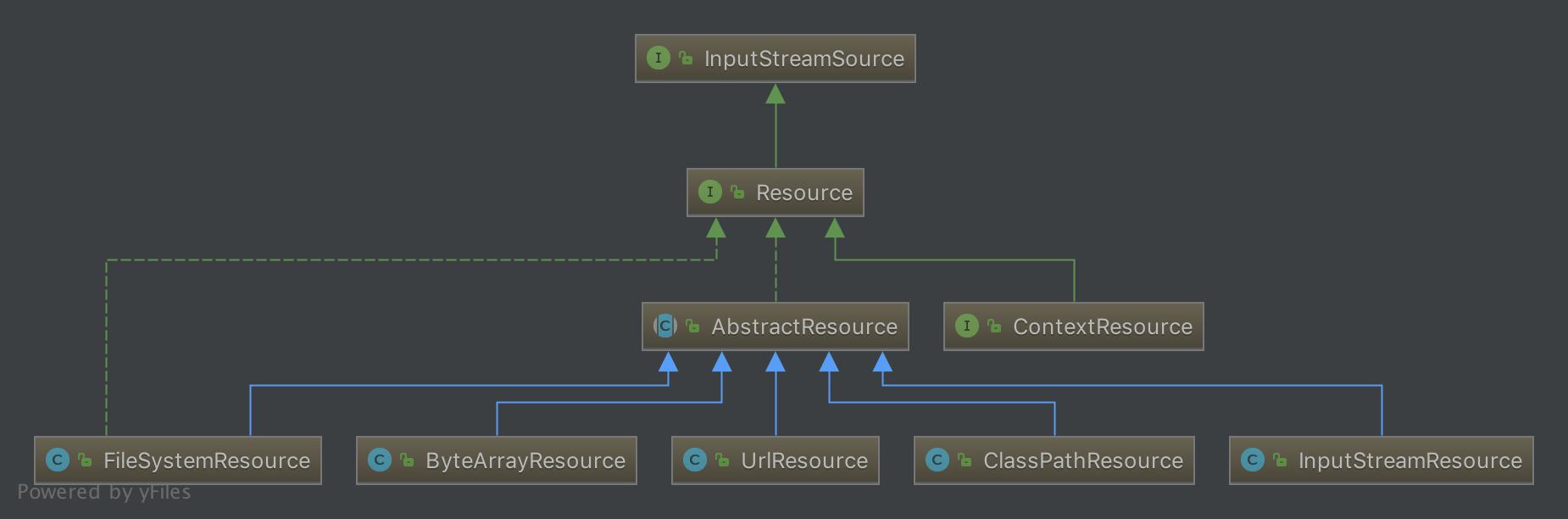
## 1. 统一资源：Resource

org.springframework.core.io.Resource 为 Spring 框架所有资源的抽象和访问接口，它继承 org.springframework.core.io.InputStreamSource接口。作为所有资源的统一抽象，Resource 定义了一些通用的方法，由子类 AbstractResource 提供统一的默认实现。定义如下：

|  |
| --- |
| public interface Resource extends InputStreamSource {   /\*\*  \* 资源是否存在  \*/  boolean exists();   /\*\*  \* 资源是否可读  \*/  default boolean isReadable() {  return true;  }   /\*\*  \* 资源所代表的句柄是否被一个 stream 打开了  \*/  default boolean isOpen() {  return false;  }   /\*\*  \* 是否为 File  \*/  default boolean isFile() {  return false;  }   /\*\*  \* 返回资源的 URL 的句柄  \*/  URL getURL() throws IOException;   /\*\*  \* 返回资源的 URI 的句柄  \*/  URI getURI() throws IOException;   /\*\*  \* 返回资源的 File 的句柄  \*/  File getFile() throws IOException;   /\*\*  \* 返回 ReadableByteChannel  \*/  default ReadableByteChannel readableChannel() throws IOException {  return java.nio.channels.Channels.newChannel(getInputStream());  }   /\*\*  \* 资源内容的长度  \*/  long contentLength() throws IOException;   /\*\*  \* 资源最后的修改时间  \*/  long lastModified() throws IOException;   /\*\*  \* 根据资源的相对路径创建新资源  \*/  Resource createRelative(String relativePath) throws IOException;   /\*\*  \* 资源的文件名  \*/  @Nullable  String getFilename();   /\*\*  \* 资源的描述  \*/  String getDescription();  } |

## 1.1 子类结构

类结构图如下：



从上图可以看到，Resource 根据资源的不同类型提供不同的具体实现，如下：

* FileSystemResource ：对 java.io.File 类型资源的封装，只要是跟 File 打交道的，基本上与 FileSystemResource 也可以打交道。支持文件和 URL 的形式，实现 WritableResource 接口，且从 Spring Framework 5.0 开始，FileSystemResource 使用 NIO2 API进行读/写交互。
* ByteArrayResource ：对字节数组提供的数据的封装。如果通过 InputStream 形式访问该类型的资源，该实现会根据字节数组的数据构造一个相应的 ByteArrayInputStream。
* UrlResource ：对 java.net.URL类型资源的封装。内部委派 URL 进行具体的资源操作。
* ClassPathResource ：class path 类型资源的实现。使用给定的 ClassLoader 或者给定的 Class 来加载资源。
* InputStreamResource ：将给定的 InputStream 作为一种资源的 Resource 的实现类。

## 1.2 AbstractResource

org.springframework.core.io.AbstractResource ，为 Resource 接口的默认**抽象**实现。它实现了 Resource 接口的**大部分的公共实现**，作为 Resource 接口中的重中之重，其定义如下：

|  |
| --- |
| public abstract class AbstractResource implements Resource {   /\*\*  \* 判断文件是否存在，若判断过程产生异常（因为会调用SecurityManager来判断），就关闭对应的流  \*/  @Override  public boolean exists() {  try {  // 基于 File 进行判断  return getFile().exists();  }  catch (IOException ex) {  // Fall back to stream existence: can we open the stream?  // 基于 InputStream 进行判断  try {  InputStream is = getInputStream();  is.close();  return true;  } catch (Throwable isEx) {  return false;  }  }  }   /\*\*  \* 直接返回true，表示可读  \*/  @Override  public boolean isReadable() {  return true;  }   /\*\*  \* 直接返回 false，表示未被打开  \*/  @Override  public boolean isOpen() {  return false;  }   /\*\*  \* 直接返回false，表示不为 File  \*/  @Override  public boolean isFile() {  return false;  }   /\*\*  \* 抛出 FileNotFoundException 异常，交给子类实现  \*/  @Override  public URL getURL() throws IOException {  throw new FileNotFoundException(getDescription() + " cannot be resolved to URL");   }   /\*\*  \* 基于 getURL() 返回的 URL 构建 URI  \*/  @Override  public URI getURI() throws IOException {  URL url = getURL();  try {  return ResourceUtils.toURI(url);  } catch (URISyntaxException ex) {  throw new NestedIOException("Invalid URI [" + url + "]", ex);  }  }   /\*\*  \* 抛出 FileNotFoundException 异常，交给子类实现  \*/  @Override  public File getFile() throws IOException {  throw new FileNotFoundException(getDescription() + " cannot be resolved to absolute file path");  }   /\*\*  \* 根据 getInputStream() 的返回结果构建 ReadableByteChannel  \*/  @Override  public ReadableByteChannel readableChannel() throws IOException {  return Channels.newChannel(getInputStream());  }   /\*\*  \* 获取资源的长度  \*  \* 这个资源内容长度实际就是资源的字节长度，通过全部读取一遍来判断  \*/  @Override  public long contentLength() throws IOException {  InputStream is = getInputStream();  try {  long size = 0;  byte[] buf = new byte[255]; // 每次最多读取 255 字节  int read;  while ((read = is.read(buf)) != -1) {  size += read;  }  return size;  } finally {  try {  is.close();  } catch (IOException ex) {  }  }  }   /\*\*  \* 返回资源最后的修改时间  \*/  @Override  public long lastModified() throws IOException {  long lastModified = getFileForLastModifiedCheck().lastModified();  if (lastModified == 0L) {  throw new FileNotFoundException(getDescription() +  " cannot be resolved in the file system for resolving its last-modified timestamp");  }  return lastModified;  }   protected File getFileForLastModifiedCheck() throws IOException {  return getFile();  }   /\*\*  \* 抛出 FileNotFoundException 异常，交给子类实现  \*/  @Override  public Resource createRelative(String relativePath) throws IOException {  throw new FileNotFoundException("Cannot create a relative resource for " + getDescription());  }   /\*\*  \* 获取资源名称，默认返回 null ，交给子类实现  \*/  @Override  @Nullable  public String getFilename() {  return null;  }   /\*\*  \* 返回资源的描述  \*/  @Override  public String toString() {  return getDescription();  }   @Override  public boolean equals(Object obj) {  return (obj == this ||  (obj instanceof Resource && ((Resource) obj).getDescription().equals(getDescription())));  }   @Override  public int hashCode() {  return getDescription().hashCode();  }  } |

如果我们想要实现自定义的 Resource ，记住不要实现 Resource 接口，而应该继承 AbstractResource 抽象类，然后根据当前的具体资源特性覆盖相应的方法即可。

## 1.3 其他子类

来自艿艿

Resource 的子类，例如 FileSystemResource、ByteArrayResource 等等的代码非常简单。感兴趣的胖友，自己去研究。

## 2. 统一资源定位：ResourceLoader

一开始就说了 Spring 将资源的定义和资源的加载区分开了，Resource 定义了统一的资源，**那资源的加载则由 ResourceLoader 来统一定义**。

org.springframework.core.io.ResourceLoader 为 Spring 资源加载的统一抽象，具体的资源加载则由相应的实现类来完成，所以我们可以将 ResourceLoader 称作为统一资源定位器。其定义如下：

FROM 《Spring 源码深度解析》P16 页

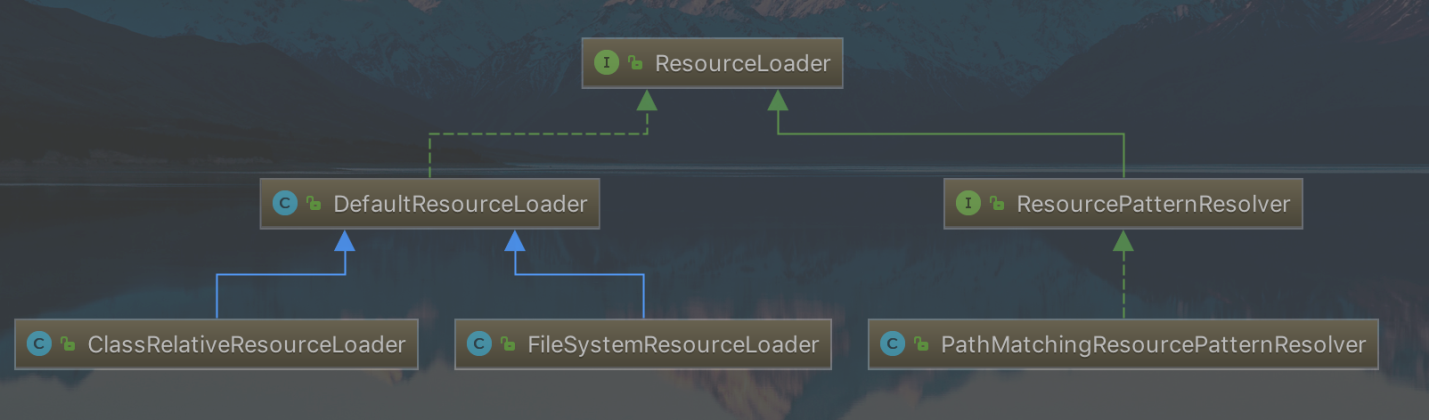
ResourceLoader，定义资源加载器，主要应用于根据给定的资源文件地址，返回对应的 Resource 。

|  |
| --- |
| public interface ResourceLoader {   String CLASSPATH\_URL\_PREFIX = ResourceUtils.CLASSPATH\_URL\_PREFIX; // CLASSPATH URL 前缀。默认为："classpath:"   Resource getResource(String location);   ClassLoader getClassLoader();  } |

* #getResource(String location) 方法，根据所提供资源的路径 location 返回 Resource 实例，但是它不确保该 Resource 一定存在，需要调用 Resource#exist() 方法来判断。
  + 该方法支持以下模式的资源加载：
    - URL位置资源，如 "file:C:/test.dat" 。
    - ClassPath位置资源，如 "classpath:test.dat 。
    - 相对路径资源，如 "WEB-INF/test.dat" ，此时返回的Resource 实例，根据实现不同而不同。
  + 该方法的主要实现是在其子类 DefaultResourceLoader 中实现，具体过程我们在分析 DefaultResourceLoader 时做详细说明。
* #getClassLoader() 方法，返回 ClassLoader 实例，对于想要获取 ResourceLoader 使用的 ClassLoader 用户来说，可以直接调用该方法来获取。在分析 Resource 时，提到了一个类 ClassPathResource ，这个类是可以根据指定的 ClassLoader 来加载资源的。

## 2.1 子类结构

作为 Spring 统一的资源加载器，它提供了统一的抽象，具体的实现则由相应的子类来负责实现，其类的类结构图如下：



## 2.1 DefaultResourceLoader

与 DefaultResource 相似，org.springframework.core.io.DefaultResourceLoader 是 ResourceLoader 的默认实现。

### 2.1.1 构造函数

它接收 ClassLoader 作为构造函数的参数，或者使用不带参数的构造函数。

* 在使用**不带**参数的构造函数时，使用的 ClassLoader 为默认的 ClassLoader（一般 Thread.currentThread()#getContextClassLoader() ）。
* 在使用**带**参数的构造函数时，可以通过 ClassUtils#getDefaultClassLoader()获取。

代码如下：

|  |
| --- |
| @Nullable private ClassLoader classLoader;   public DefaultResourceLoader() { // 无参构造函数  this.classLoader = ClassUtils.getDefaultClassLoader(); }  public DefaultResourceLoader(@Nullable ClassLoader classLoader) { // 带 ClassLoader 参数的构造函数  this.classLoader = classLoader; }  public void setClassLoader(@Nullable ClassLoader classLoader) {  this.classLoader = classLoader; }  @Override @Nullable public ClassLoader getClassLoader() {  return (this.classLoader != null ? this.classLoader : ClassUtils.getDefaultClassLoader()); } |

* 另外，也可以调用 #setClassLoader() 方法进行后续设置。

### 2.1.2 getResource 方法

ResourceLoader 中最核心的方法为 #getResource(String location) ，它根据提供的 location 返回相应的 Resource 。而 DefaultResourceLoader 对该方法提供了**核心实现**（因为，它的两个子类都没有提供覆盖该方法，所以可以断定 ResourceLoader 的资源加载策略就封装在 DefaultResourceLoader 中)，代码如下：

|  |
| --- |
| // DefaultResourceLoader.java  @Override public Resource getResource(String location) {  Assert.notNull(location, "Location must not be null");   // 首先，通过 ProtocolResolver 来加载资源  for (ProtocolResolver protocolResolver : this.protocolResolvers) {  Resource resource = protocolResolver.resolve(location, this);  if (resource != null) {  return resource;  }  }  // 其次，以 / 开头，返回 ClassPathContextResource 类型的资源  if (location.startsWith("/")) {  return getResourceByPath(location);  // 再次，以 classpath: 开头，返回 ClassPathResource 类型的资源  } else if (location.startsWith(CLASSPATH\_URL\_PREFIX)) {  return new ClassPathResource(location.substring(CLASSPATH\_URL\_PREFIX.length()), getClassLoader());  // 然后，根据是否为文件 URL ，是则返回 FileUrlResource 类型的资源，否则返回 UrlResource 类型的资源  } else {  try {  // Try to parse the location as a URL...  URL url = new URL(location);  return (ResourceUtils.isFileURL(url) ? new FileUrlResource(url) : new UrlResource(url));  } catch (MalformedURLException ex) {  // 最后，返回 ClassPathContextResource 类型的资源  // No URL -> resolve as resource path.  return getResourceByPath(location);  }  } } |

* 首先，通过 ProtocolResolver 来加载资源，成功返回 Resource 。
* 其次，若 location 以 "/" 开头，则调用 #getResourceByPath() 方法，构造 ClassPathContextResource 类型资源并返回。代码如下：

|  |
| --- |
| protected Resource getResourceByPath(String path) {  return new ClassPathContextResource(path, getClassLoader()); } |

* 再次，若 location 以 "classpath:" 开头，则构造 ClassPathResource 类型资源并返回。在构造该资源时，通过 #getClassLoader() 获取当前的 ClassLoader。
* 然后，构造 URL ，尝试通过它进行资源定位，若没有抛出 MalformedURLException 异常，则判断是否为 FileURL , 如果是则构造 FileUrlResource 类型的资源，否则构造 UrlResource 类型的资源。
* 最后，若在加载过程中抛出 MalformedURLException 异常，则委派 #getResourceByPath() 方法，实现资源定位加载。😈 实际上，和【其次】相同落。

### 2.1.3 ProtocolResolver

org.springframework.core.io.ProtocolResolver ，用户自定义协议资源解决策略，作为 DefaultResourceLoader 的 **SPI**：它允许用户自定义资源加载协议，而不需要继承 ResourceLoader 的子类。  
在介绍 Resource 时，提到如果要实现自定义 Resource，我们只需要继承 DefaultResource 即可，但是有了 ProtocolResolver 后，我们不需要直接继承 DefaultResourceLoader，改为实现 ProtocolResolver 接口也可以实现自定义的 ResourceLoader。

ProtocolResolver 接口，仅有一个方法 Resource resolve(String location, ResourceLoader resourceLoader) 。代码如下：

|  |
| --- |
| /\*\*  \* 使用指定的 ResourceLoader ，解析指定的 location 。  \* 若成功，则返回对应的 Resource 。  \*  \* Resolve the given location against the given resource loader  \* if this implementation's protocol matches.  \* @param location the user-specified resource location 资源路径  \* @param resourceLoader the associated resource loader 指定的加载器 ResourceLoader  \* @return a corresponding {@code Resource} handle if the given location  \* matches this resolver's protocol, or {@code null} otherwise 返回为相应的 Resource  \*/ @Nullable Resource resolve(String location, ResourceLoader resourceLoader); |

在 Spring 中你会发现该接口并没有实现类，它需要用户自定义，自定义的 Resolver 如何加入 Spring 体系呢？调用 DefaultResourceLoader#addProtocolResolver(ProtocolResolver) 方法即可。代码如下：

|  |
| --- |
| /\*\*  \* ProtocolResolver 集合  \*/ private final Set<ProtocolResolver> protocolResolvers = new LinkedHashSet<>(4);   public void addProtocolResolver(ProtocolResolver resolver) {  Assert.notNull(resolver, "ProtocolResolver must not be null");  this.protocolResolvers.add(resolver); } |

### 2.1.4 示例

下面示例是演示 DefaultResourceLoader 加载资源的具体策略，代码如下（该示例参考[《Spring 揭秘》](https://item.jd.com/10062476670.html?jd_pop=1b9f1d77-f489-4473-b2f0-f22036d20060&abt=0) P89）：

|  |
| --- |
| ResourceLoader resourceLoader = new DefaultResourceLoader();  Resource fileResource1 = resourceLoader.getResource("D:/Users/chenming673/Documents/spark.txt"); System.out.println("fileResource1 is FileSystemResource:" + (fileResource1 instanceof FileSystemResource));  Resource fileResource2 = resourceLoader.getResource("/Users/chenming673/Documents/spark.txt"); System.out.println("fileResource2 is ClassPathResource:" + (fileResource2 instanceof ClassPathResource));  Resource urlResource1 = resourceLoader.getResource("file:/Users/chenming673/Documents/spark.txt"); System.out.println("urlResource1 is UrlResource:" + (urlResource1 instanceof UrlResource));  Resource urlResource2 = resourceLoader.getResource("http://www.baidu.com"); System.out.println("urlResource1 is urlResource:" + (urlResource2 instanceof UrlResource)); |

运行结果：

|  |
| --- |
| fileResource1 is FileSystemResource:false fileResource2 is ClassPathResource:true urlResource1 is UrlResource:true urlResource1 is urlResource:true |

* 其实对于 fileResource1 ，我们更加希望是 FileSystemResource 资源类型。但是，事与愿违，它是 ClassPathResource 类型。为什么呢？在 DefaultResourceLoader#getResource() 方法的资源加载策略中，我们知道 "D:/Users/chenming673/Documents/spark.txt" 地址，其实在该方法中没有相应的资源类型，那么它就会在抛出 MalformedURLException 异常时，通过 DefaultResourceLoader#getResourceByPath(...) 方法，构造一个 ClassPathResource 类型的资源。
* 而 urlResource1 和 urlResource2 ，指定有协议前缀的资源路径，则通过 URL 就可以定义，所以返回的都是 UrlResource 类型。

## 2.2 FileSystemResourceLoader

从上面的示例，我们看到，其实 DefaultResourceLoader 对#getResourceByPath(String) 方法处理其实不是很恰当，这个时候我们可以使用 org.springframework.core.io.FileSystemResourceLoader 。它继承 DefaultResourceLoader ，且覆写了 #getResourceByPath(String) 方法，使之从文件系统加载资源并以 FileSystemResource 类型返回，这样我们就可以得到想要的资源类型。代码如下：

|  |
| --- |
| @Override protected Resource getResourceByPath(String path) {  // 截取首 /  if (path.startsWith("/")) {  path = path.substring(1);  }  // 创建 FileSystemContextResource 类型的资源  return new FileSystemContextResource(path); } |

### 2.2.1 FileSystemContextResource

FileSystemContextResource ，为 FileSystemResourceLoader 的内部类，它继承 FileSystemResource 类，实现 ContextResource 接口。代码如下：

|  |
| --- |
| /\*\*  \* FileSystemResource that explicitly expresses a context-relative path  \* through implementing the ContextResource interface.  \*/ private static class FileSystemContextResource extends FileSystemResource implements ContextResource {   public FileSystemContextResource(String path) {  super(path);  }   @Override  public String getPathWithinContext() {  return getPath();  } } |

* 在构造器中，也是调用 FileSystemResource 的构造函数来构造 FileSystemResource 的。
* 为什么要有 FileSystemContextResource 类的原因是，实现 ContextResource 接口，并实现对应的 #getPathWithinContext() 接口方法。

### 2.2.2 示例

😈 在回过头看 [「2.1.4 示例」](http://yudaoyuanma:ydym1024@svip.iocoder.cn/Spring/IoC-load-Resource/) ，如果将 DefaultResourceLoader 改为 FileSystemContextResource ，则 fileResource1 则为 FileSystemResource 类型的资源。

## 2.3 ClassRelativeResourceLoader

org.springframework.core.io.ClassRelativeResourceLoader ，是 DefaultResourceLoader 的另一个子类的实现。和 FileSystemResourceLoader 类似，在实现代码的结构上类似，也是覆写 #getResourceByPath(String path) 方法，并返回其对应的 ClassRelativeContextResource 的资源类型。

感兴趣的胖友，可以看看 [《Spring5：就这一次，搞定资源加载器之ClassRelativeResourceLoader》](https://blog.csdn.net/seasonsbin/article/details/80914911) 文章。

ClassRelativeResourceLoader 扩展的功能是，可以根据给定的class 所在包或者所在包的子包下加载资源。

## 2.4 ResourcePatternResolver

ResourceLoader 的 Resource getResource(String location) 方法，每次只能根据 location 返回**一个** Resource 。当需要加载多个资源时，我们除了多次调用 #getResource(String location) 方法外，别无他法。org.springframework.core.io.support.ResourcePatternResolver 是 ResourceLoader 的扩展，它支持根据指定的资源路径匹配模式每次返回**多个** Resource 实例，其定义如下：

|  |
| --- |
| public interface ResourcePatternResolver extends ResourceLoader {   String CLASSPATH\_ALL\_URL\_PREFIX = "classpath\*:";   Resource[] getResources(String locationPattern) throws IOException;  } |

* ResourcePatternResolver 在 ResourceLoader 的基础上增加了 #getResources(String locationPattern) 方法，以支持根据路径匹配模式返回**多个** Resource 实例。
* 同时，也新增了一种**新的协议**前缀 "classpath\*:"，该协议前缀由其子类负责实现。

## 2.5 PathMatchingResourcePatternResolver

org.springframework.core.io.support.PathMatchingResourcePatternResolver ，为 ResourcePatternResolver 最常用的子类，它除了支持 ResourceLoader 和 ResourcePatternResolver 新增的 "classpath\*:" 前缀外，**还支持 Ant 风格的路径匹配模式**（类似于 "\*\*/\*.xml"）。

### 2.5.1 构造函数

PathMatchingResourcePatternResolver 提供了三个构造函数，如下：

|  |
| --- |
| /\*\*  \* 内置的 ResourceLoader 资源定位器  \*/ private final ResourceLoader resourceLoader; /\*\*  \* Ant 路径匹配器  \*/ private PathMatcher pathMatcher = new AntPathMatcher();  public PathMatchingResourcePatternResolver() {  this.resourceLoader = new DefaultResourceLoader(); }  public PathMatchingResourcePatternResolver(ResourceLoader resourceLoader) {  Assert.notNull(resourceLoader, "ResourceLoader must not be null");  this.resourceLoader = resourceLoader; }  public PathMatchingResourcePatternResolver(@Nullable ClassLoader classLoader) {  this.resourceLoader = new DefaultResourceLoader(classLoader); } |

* PathMatchingResourcePatternResolver 在实例化的时候，可以指定一个 ResourceLoader，如果不指定的话，它会在内部构造一个 DefaultResourceLoader 。
* pathMatcher 属性，默认为 AntPathMatcher 对象，用于支持 Ant 类型的路径匹配。

### 2.5.2 getResource

|  |
| --- |
| @Override public Resource getResource(String location) {  return getResourceLoader().getResource(location); }  public ResourceLoader getResourceLoader() {  return this.resourceLoader; } |

该方法，直接委托给相应的 ResourceLoader 来实现。所以，如果我们在实例化的 PathMatchingResourcePatternResolver 的时候，如果未指定 ResourceLoader 参数的情况下，那么在加载资源时，其实就是 DefaultResourceLoader 的过程。

其实在下面介绍的 Resource[] getResources(String locationPattern) 方法也相同，只不过返回的资源是**多个**而已。

### 2.5.3 getResources

|  |
| --- |
| @Override public Resource[] getResources(String locationPattern) throws IOException {  Assert.notNull(locationPattern, "Location pattern must not be null");  // 以 "classpath\*:" 开头  if (locationPattern.startsWith(CLASSPATH\_ALL\_URL\_PREFIX)) {  // 路径包含通配符  // a class path resource (multiple resources for same name possible)  if (getPathMatcher().isPattern(locationPattern.substring(CLASSPATH\_ALL\_URL\_PREFIX.length()))) {  // a class path resource pattern  return findPathMatchingResources(locationPattern);  // 路径不包含通配符  } else {  // all class path resources with the given name  return findAllClassPathResources(locationPattern.substring(CLASSPATH\_ALL\_URL\_PREFIX.length()));  }  // 不以 "classpath\*:" 开头  } else {  // Generally only look for a pattern after a prefix here, // 通常只在这里的前缀后面查找模式  // and on Tomcat only after the "\*/" separator for its "war:" protocol. 而在 Tomcat 上只有在 “\*/ ”分隔符之后才为其 “war:” 协议  int prefixEnd = (locationPattern.startsWith("war:") ? locationPattern.indexOf("\*/") + 1 :  locationPattern.indexOf(':') + 1);  // 路径包含通配符  if (getPathMatcher().isPattern(locationPattern.substring(prefixEnd))) {  // a file pattern  return findPathMatchingResources(locationPattern);  // 路径不包含通配符  } else {  // a single resource with the given name  return new Resource[] {getResourceLoader().getResource(locationPattern)};  }  } } |

处理逻辑如下图：

* **非** "classpath\*:" 开头，且路径**不包含**通配符，直接委托给相应的 ResourceLoader 来实现。
* 其他情况，调用 #findAllClassPathResources(...)、或 #findPathMatchingResources(...) 方法，返回多个 Resource 。下面，我们来详细分析。

### 2.5.4 findAllClassPathResources

当 locationPattern 以 "classpath\*:" 开头但是不包含通配符，则调用 #findAllClassPathResources(...) 方法加载资源。该方法返回 classes 路径下和所有 jar 包中的所有相匹配的资源。

|  |
| --- |
| protected Resource[] findAllClassPathResources(String location) throws IOException {  String path = location;  // 去除首个 /  if (path.startsWith("/")) {  path = path.substring(1);  }  // 真正执行加载所有 classpath 资源  Set<Resource> result = doFindAllClassPathResources(path);  if (logger.isTraceEnabled()) {  logger.trace("Resolved classpath location [" + location + "] to resources " + result);  }  // 转换成 Resource 数组返回  return result.toArray(new Resource[0]); } |

真正执行加载的是在 #doFindAllClassPathResources(...) 方法，代码如下：

|  |
| --- |
| protected Set<Resource> doFindAllClassPathResources(String path) throws IOException {  Set<Resource> result = new LinkedHashSet<>(16);  ClassLoader cl = getClassLoader();  // <1> 根据 ClassLoader 加载路径下的所有资源  Enumeration<URL> resourceUrls = (cl != null ? cl.getResources(path) : ClassLoader.getSystemResources(path));  // <2>   while (resourceUrls.hasMoreElements()) {  URL url = resourceUrls.nextElement();  // 将 URL 转换成 UrlResource  result.add(convertClassLoaderURL(url));  }  // <3> 加载路径下得所有 jar 包  if ("".equals(path)) {  // The above result is likely to be incomplete, i.e. only containing file system references.  // We need to have pointers to each of the jar files on the classpath as well...  addAllClassLoaderJarRoots(cl, result);  }  return result; } |

* <1> 处，根据 ClassLoader 加载路径下的所有资源。在加载资源过程时，如果在构造 PathMatchingResourcePatternResolver 实例的时候如果传入了 ClassLoader，则调用该 ClassLoader 的 #getResources() 方法，否则调用 ClassLoader#getSystemResources(path) 方法。另外，ClassLoader#getResources() 方法，代码如下:

|  |
| --- |
| // java.lang.ClassLoader.java public Enumeration<URL> getResources(String name) throws IOException {  @SuppressWarnings("unchecked")  Enumeration<URL>[] tmp = (Enumeration<URL>[]) new Enumeration<?>[2];  if (parent != null) {  tmp[0] = parent.getResources(name);  } else {  tmp[0] = getBootstrapResources(name);  }  tmp[1] = findResources(name);   return new CompoundEnumeration<>(tmp); } |

* + 看到这里是不是就已经一目了然了？如果当前父类加载器不为 null ，则通过父类向上迭代获取资源，否则调用 #getBootstrapResources() 。这里是不是特别熟悉，(^▽^)。
* <2> 处，遍历 URL 集合，调用 #convertClassLoaderURL(URL url) 方法，将 URL 转换成 UrlResource 对象。代码如下：

|  |
| --- |
| protected Resource convertClassLoaderURL(URL url) {  return new UrlResource(url); } |

* <3> 处，若 path 为空（“”）时，则调用 #addAllClassLoaderJarRoots(...)方法。该方法主要是加载路径下得所有 jar 包，方法较长也没有什么实际意义就不贴出来了。感兴趣的胖友，自己可以去看看。😈 当然，可能代码也比较长哈。

通过上面的分析，我们知道 #findAllClassPathResources(...) 方法，其实就是利用 ClassLoader 来加载指定路径下的资源，不论它是在 class 路径下还是在 jar 包中。如果我们传入的路径为空或者 /，则会调用 #addAllClassLoaderJarRoots(...) 方法，加载所有的 jar 包。

### 2.5.5 findPathMatchingResources

当 locationPattern 中包含了**通配符**，则调用该方法进行资源加载。代码如下：

|  |
| --- |
| protected Resource[] findPathMatchingResources(String locationPattern) throws IOException {  // 确定根路径、子路径  String rootDirPath = determineRootDir(locationPattern);  String subPattern = locationPattern.substring(rootDirPath.length());  // 获取根据路径下的资源  Resource[] rootDirResources = getResources(rootDirPath);  // 遍历，迭代  Set<Resource> result = new LinkedHashSet<>(16);  for (Resource rootDirResource : rootDirResources) {  rootDirResource = resolveRootDirResource(rootDirResource);  URL rootDirUrl = rootDirResource.getURL();  // bundle 资源类型  if (equinoxResolveMethod != null && rootDirUrl.getProtocol().startsWith("bundle")) {  URL resolvedUrl = (URL) ReflectionUtils.invokeMethod(equinoxResolveMethod, null, rootDirUrl);  if (resolvedUrl != null) {  rootDirUrl = resolvedUrl;  }  rootDirResource = new UrlResource(rootDirUrl);  }  // vfs 资源类型  if (rootDirUrl.getProtocol().startsWith(ResourceUtils.URL\_PROTOCOL\_VFS)) {  result.addAll(VfsResourceMatchingDelegate.findMatchingResources(rootDirUrl, subPattern, getPathMatcher()));  // jar 资源类型  } else if (ResourceUtils.isJarURL(rootDirUrl) || isJarResource(rootDirResource)) {  result.addAll(doFindPathMatchingJarResources(rootDirResource, rootDirUrl, subPattern));  // 其它资源类型  } else {  result.addAll(doFindPathMatchingFileResources(rootDirResource, subPattern));  }  }  if (logger.isTraceEnabled()) {  logger.trace("Resolved location pattern [" + locationPattern + "] to resources " + result);  }  // 转换成 Resource 数组返回  return result.toArray(new Resource[0]); } |

方法有点儿长，但是思路还是很清晰的，主要分两步：

1. 确定目录，获取该目录下得所有资源。
2. 在所获得的所有资源后，进行迭代匹配获取我们想要的资源。

在这个方法里面，我们要关注两个方法，一个是 #determineRootDir(String location) 方法，一个是 #doFindPathMatchingXXXResources(...) 等方法。

#### 2.5.5.1 determineRootDir

determineRootDir(String location) 方法，主要是用于确定根路径。代码如下：

|  |
| --- |
| /\*\*  \* Determine the root directory for the given location.  \* <p>Used for determining the starting point for file matching,  \* resolving the root directory location to a {@code java.io.File}  \* and passing it into {@code retrieveMatchingFiles}, with the  \* remainder of the location as pattern.  \* <p>Will return "/WEB-INF/" for the pattern "/WEB-INF/\*.xml",  \* for example.  \* @param location the location to check  \* @return the part of the location that denotes the root directory  \* @see #retrieveMatchingFiles  \*/ protected String determineRootDir(String location) {  // 找到冒号的后一位  int prefixEnd = location.indexOf(':') + 1;  // 根目录结束位置  int rootDirEnd = location.length();  // 在从冒号开始到最后的字符串中，循环判断是否包含通配符，如果包含，则截断最后一个由”/”分割的部分。  // 例如：在我们路径中，就是最后的ap?-context.xml这一段。再循环判断剩下的部分，直到剩下的路径中都不包含通配符。  while (rootDirEnd > prefixEnd && getPathMatcher().isPattern(location.substring(prefixEnd, rootDirEnd))) {  rootDirEnd = location.lastIndexOf('/', rootDirEnd - 2) + 1;  }  // 如果查找完成后，rootDirEnd = 0 了，则将之前赋值的 prefixEnd 的值赋给 rootDirEnd ，也就是冒号的后一位  if (rootDirEnd == 0) {  rootDirEnd = prefixEnd;  }  // 截取根目录  return location.substring(0, rootDirEnd); } |

方法比较绕，效果如下示例：

| **原路径** | **确定根路径** |
| --- | --- |
| classpath\*:test/cc\*/spring-\*.xml | classpath\*:test/ |
| classpath\*:test/aa/spring-\*.xml | classpath\*:test/aa/ |

#### 2.5.5.2 doFindPathMatchingXXXResources

来自艿艿

#doFindPathMatchingXXXResources(...) 方法，是个泛指，一共对应三个方法：

* #doFindPathMatchingJarResources(rootDirResource, rootDirUrl, subPatter) 方法
* #doFindPathMatchingFileResources(rootDirResource, subPattern) 方法
* VfsResourceMatchingDelegate#findMatchingResources(rootDirUrl, subPattern, pathMatcher) 方法

因为本文重在分析 Spring 统一资源加载策略的整体**流程**。相对来说，上面几个方法的代码量会比较多。所以本文不再追溯，感兴趣的胖友，推荐阅读如下文章：

* [《Spring源码情操陶冶-PathMatchingResourcePatternResolver路径资源匹配溶解器》](https://www.cnblogs.com/question-sky/p/6959493.html) ，主要针对 #doFindPathMatchingJarResources(rootDirResource, rootDirUrl, subPatter) 方法。
* [《深入 Spring IoC 源码之 ResourceLoader》](http://www.blogjava.net/DLevin/archive/2012/12/01/392337.html) ，主要针对 #doFindPathMatchingFileResources(rootDirResource, subPattern) 方法。
* [《Spring 源码学习 —— 含有通配符路径解析（上）》](http://www.coderli.com/spring-wildpath-parse/) 😈 貌似没有下

## 3. 小结

至此 Spring 整个资源记载过程已经分析完毕。下面简要总结下：

* Spring 提供了 Resource 和 ResourceLoader 来统一抽象整个资源及其定位。使得资源与资源的定位有了一个更加清晰的界限，并且提供了合适的 Default 类，使得自定义实现更加方便和清晰。
* DefaultResource 为 Resource 的默认实现，它对 Resource 接口做了一个统一的实现，子类继承该类后只需要覆盖相应的方法即可，同时对于自定义的 Resource 我们也是继承该类。
* DefaultResourceLoader 同样也是 ResourceLoader 的默认实现，在自定 ResourceLoader 的时候我们除了可以继承该类外还可以实现 ProtocolResolver 接口来实现自定资源加载协议。
* DefaultResourceLoader 每次只能返回单一的资源，所以 Spring 针对这个提供了另外一个接口 ResourcePatternResolver ，该接口提供了根据指定的 locationPattern 返回多个资源的策略。其子类 PathMatchingResourcePatternResolver 是一个集大成者的 ResourceLoader ，因为它即实现了 Resource getResource(String location) 方法，也实现了 Resource[] getResources(String locationPattern) 方法。

另外，如果胖友认真的看了本文的包结构，我们可以发现，Resource 和 ResourceLoader 核心是在，spring-core 项目中。

如果想要调试本小节的相关内容，可以直接使用 Resource 和 ResourceLoader 相关的 API ，进行操作调试。

# 【死磕 Spring】—— IoC 之加载 BeanDefinition

**本文主要基于 Spring 5.0.6.RELEASE**

摘要: 原创出处 <http://cmsblogs.com/?p=2658> 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老艿艿」略作修改，记录在理解过程中，参考的资料。

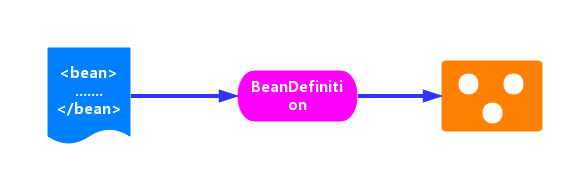
先看一段熟悉的代码：

|  |
| --- |
| ClassPathResource resource = new ClassPathResource("bean.xml"); // <1> DefaultListableBeanFactory factory = new DefaultListableBeanFactory(); // <2> XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory); // <3> reader.loadBeanDefinitions(resource); // <4> |

这段代码是 Spring 中编程式使用 IoC 容器，通过这四段简单的代码，我们可以初步判断 IoC 容器的使用过程。

1. 获取资源
2. 获取 BeanFactory
3. 根据新建的 BeanFactory 创建一个 BeanDefinitionReader 对象，该 Reader 对象为资源的**解析器**
4. 装载资源

整个过程就分为三个步骤：资源定位、装载、注册，如下：



* **资源定位**。我们一般用外部资源来描述 Bean 对象，所以在初始化 IoC 容器的第一步就是需要定位这个外部资源。在上一篇博客（[《【死磕 Spring】—— IoC 之 Spring 统一资源加载策略》](http://svip.iocoder.cn/Spring/IoC-load-Resource)）已经详细说明了资源加载的过程。
* **装载**。装载就是 BeanDefinition 的载入。BeanDefinitionReader 读取、解析 Resource 资源，也就是将用户定义的 Bean 表示成 IoC 容器的内部数据结构：BeanDefinition 。
  + 在 IoC 容器内部维护着一个 BeanDefinition Map 的数据结构
  + 在配置文件中每一个 <bean> 都对应着一个 BeanDefinition 对象。
  + 😈 本文，我们分享的就是**装载**这个步骤。

FROM 《Spring 源码深度解析》P16 页  
BeanDefinitionReader ，主要定义资源文件读取并转换为 BeanDefinition 的各个功能。

* **注册**。向 IoC 容器注册在第二步解析好的 BeanDefinition，这个过程是通过 BeanDefinitionRegistry 接口来实现的。在 IoC 容器内部其实是将第二个过程解析得到的 BeanDefinition 注入到一个 HashMap 容器中，IoC 容器就是通过这个 HashMap 来维护这些 BeanDefinition 的。
  + 在这里需要注意的一点是这个过程并没有完成依赖注入，依赖注册是发生在应用第一次调用 #getBean(...) 方法，向容器索要 Bean 时。
  + 当然我们可以通过设置预处理，即对某个 Bean 设置 lazyinit = false 属性，那么这个 Bean 的依赖注入就会在容器初始化的时候完成。

FROM 老艿艿

简单的说，上面步骤的结果是，XML Resource => XML Document => Bean Definition 。

## 1. loadBeanDefinitions

资源定位在前面已经分析了，下面我们直接分析**加载**，上面看到的 reader.loadBeanDefinitions(resource) 代码，才是加载资源的真正实现，所以我们直接从该方法入手。代码如下：

|  |
| --- |
| // XmlBeanDefinitionReader.java @Override public int loadBeanDefinitions(Resource resource) throws BeanDefinitionStoreException {  return loadBeanDefinitions(new EncodedResource(resource)); } |

* 从指定的 xml 文件加载 Bean Definition ，这里会先对 Resource 资源封装成 org.springframework.core.io.support.EncodedResource 对象。这里为什么需要将 Resource 封装成 EncodedResource 呢？主要是为了对 Resource 进行编码，保证内容读取的正确性。
* 然后，再调用 #loadBeanDefinitions(EncodedResource encodedResource) 方法，执行真正的逻辑实现。

|  |
| --- |
| /\*\*  \* 当前线程，正在加载的 EncodedResource 集合。  \*/ private final ThreadLocal<Set<EncodedResource>> resourcesCurrentlyBeingLoaded = new NamedThreadLocal<>("XML bean definition resources currently being loaded");  public int loadBeanDefinitions(EncodedResource encodedResource) throws BeanDefinitionStoreException {  Assert.notNull(encodedResource, "EncodedResource must not be null");  if (logger.isTraceEnabled()) {  logger.trace("Loading XML bean definitions from " + encodedResource);  }   // <1> 获取已经加载过的资源  Set<EncodedResource> currentResources = this.resourcesCurrentlyBeingLoaded.get();  if (currentResources == null) {  currentResources = new HashSet<>(4);  this.resourcesCurrentlyBeingLoaded.set(currentResources);  }  if (!currentResources.add(encodedResource)) { // 将当前资源加入记录中。如果已存在，抛出异常  throw new BeanDefinitionStoreException("Detected cyclic loading of " + encodedResource + " - check your import definitions!");  }  try {  // <2> 从 EncodedResource 获取封装的 Resource ，并从 Resource 中获取其中的 InputStream  InputStream inputStream = encodedResource.getResource().getInputStream();  try {  InputSource inputSource = new InputSource(inputStream);  if (encodedResource.getEncoding() != null) { // 设置编码  inputSource.setEncoding(encodedResource.getEncoding());  }  // 核心逻辑部分，执行加载 BeanDefinition  return doLoadBeanDefinitions(inputSource, encodedResource.getResource());  } finally {  inputStream.close();  }  } catch (IOException ex) {  throw new BeanDefinitionStoreException("IOException parsing XML document from " + encodedResource.getResource(), ex);  } finally {  // 从缓存中剔除该资源 <3>  currentResources.remove(encodedResource);  if (currentResources.isEmpty()) {  this.resourcesCurrentlyBeingLoaded.remove();  }  } } |

* <1> 处，通过 resourcesCurrentlyBeingLoaded.get() 代码，来获取已经加载过的资源，然后将 encodedResource 加入其中，如果 resourcesCurrentlyBeingLoaded 中已经存在该资源，则抛出 BeanDefinitionStoreException 异常。
  + 为什么需要这么做呢？答案在 "Detected cyclic loading" ，避免一个 EncodedResource 在加载时，还没加载完成，又加载自身，从而导致**死循环**。
  + 也因此，在 <3> 处，当一个 EncodedResource 加载完成后，需要从缓存中剔除。
* <2> 处理，从 encodedResource 获取封装的 Resource 资源，并从 Resource 中获取相应的 InputStream ，然后将 InputStream 封装为 InputSource ，最后调用 #doLoadBeanDefinitions(InputSource inputSource, Resource resource) 方法，执行加载 Bean Definition 的真正逻辑。

## 2. doLoadBeanDefinitions

|  |
| --- |
| /\*\*  \* Actually load bean definitions from the specified XML file.  \* @param inputSource the SAX InputSource to read from  \* @param resource the resource descriptor for the XML file  \* @return the number of bean definitions found  \* @throws BeanDefinitionStoreException in case of loading or parsing errors  \* @see #doLoadDocument  \* @see #registerBeanDefinitions  \*/ protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)  throws BeanDefinitionStoreException {  try {  // <1> 获取 XML Document 实例  Document doc = doLoadDocument(inputSource, resource);  // <2> 根据 Document 实例，注册 Bean 信息  int count = registerBeanDefinitions(doc, resource);  if (logger.isDebugEnabled()) {  logger.debug("Loaded " + count + " bean definitions from " + resource);  }  return count;  } catch (BeanDefinitionStoreException ex) {  throw ex;  } catch (SAXParseException ex) {  throw new XmlBeanDefinitionStoreException(resource.getDescription(),  "Line " + ex.getLineNumber() + " in XML document from " + resource + " is invalid", ex);  } catch (SAXException ex) {  throw new XmlBeanDefinitionStoreException(resource.getDescription(),  "XML document from " + resource + " is invalid", ex);  } catch (ParserConfigurationException ex) {  throw new BeanDefinitionStoreException(resource.getDescription(),  "Parser configuration exception parsing XML from " + resource, ex);  } catch (IOException ex) {  throw new BeanDefinitionStoreException(resource.getDescription(),  "IOException parsing XML document from " + resource, ex);  } catch (Throwable ex) {  throw new BeanDefinitionStoreException(resource.getDescription(),  "Unexpected exception parsing XML document from " + resource, ex);  } } |

* 在 <1> 处，调用 #doLoadDocument(InputSource inputSource, Resource resource) 方法，根据 xml 文件，获取 Document 实例。
* 在 <2> 处，调用 #registerBeanDefinitions(Document doc, Resource resource) 方法，根据获取的 Document 实例，注册 Bean 信息。

## 2.1 doLoadDocument

|  |
| --- |
| /\*\*  \* 获取 XML Document 实例  \*  \* Actually load the specified document using the configured DocumentLoader.  \* @param inputSource the SAX InputSource to read from  \* @param resource the resource descriptor for the XML file  \* @return the DOM Document  \* @throws Exception when thrown from the DocumentLoader  \* @see #setDocumentLoader  \* @see DocumentLoader#loadDocument  \*/ protected Document doLoadDocument(InputSource inputSource, Resource resource) throws Exception {  return this.documentLoader.loadDocument(inputSource, getEntityResolver(), this.errorHandler,  getValidationModeForResource(resource), isNamespaceAware()); } |

1. 调用 #getValidationModeForResource(Resource resource) 方法，获取指定资源（xml）的**验证模式**。详细解析，见 [《【死磕 Spring】—— IoC 之获取验证模型》](http://svip.iocoder.cn/Spring/IoC-Validation-Mode-For-Resource) 。
2. 调用 DocumentLoader#loadDocument(InputSource inputSource, EntityResolver entityResolver, ErrorHandler errorHandler, int validationMode, boolean namespaceAware) 方法，获取 XML Document 实例。详细解析，见 [《【死磕 Spring】—— IoC 之获取 Document 对象》](http://svip.iocoder.cn/Spring/IoC-load-Document) 。

## 2.2 registerBeanDefinitions

该方法的详细解析，见 [《【死磕 Spring】—— IoC 之注册 BeanDefinition》](http://svip.iocoder.cn/Spring/IoC-register-BeanDefinitions) 。

## 666. 彩蛋

本文未完，需要继续阅读后续几篇文章。

# 死磕 Spring】—— IoC 之获取验证模型

**本文主要基于 Spring 5.0.6.RELEASE**

摘要: 原创出处 <http://cmsblogs.com/?p=2688> 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老艿艿」略作修改，记录在理解过程中，参考的资料。

在上篇博客[【死磕 Spring】—— IoC 之加载 Definitions](http://svip.iocoder.cn/Spring/IoC-load-BeanDefinitions) 中提到，在核心逻辑方法 #doLoadBeanDefinitions(InputSource inputSource, Resource resource) 方法中，中主要是做三件事情：

1. 调用 #getValidationModeForResource(Resource resource) 方法，获取指定资源（xml）的**验证模式**。
2. 调用 DocumentLoader#loadDocument(InputSource inputSource, EntityResolver entityResolver,ErrorHandler errorHandler, int validationMode, boolean namespaceAware) 方法，获取 XML Document 实例。
3. 调用 #registerBeanDefinitions(Document doc, Resource resource) 方法，根据获取的 Document 实例，注册 Bean 信息。

这篇博客主要**第 1 步**，分析获取 xml 文件的验证模式。为什么需要获取验证模式呢？原因如下：

XML 文件的验证模式保证了 XML 文件的正确性。

## 1. DTD 与 XSD 的区别

## 1.1 DTD

老艿艿：因为一些胖友，可能没了解过 DTD 和 XSD ，所以本小节先做一个科普。

DTD(Document Type Definition)，即文档类型定义，为 XML 文件的验证机制，属于 XML 文件中组成的一部分。DTD 是一种保证 XML 文档格式正确的有效验证方式，它定义了相关 XML 文档的元素、属性、排列方式、元素的内容类型以及元素的层次结构。其实 DTD 就相当于 XML 中的 “词汇”和“语法”，我们可以通过比较 XML 文件和 DTD 文件 来看文档是否符合规范，元素和标签使用是否正确。

要在 Spring 中使用 DTD，需要在 Spring XML 文件头部声明：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd"> |

DTD 在一定的阶段推动了 XML 的发展，但是它本身存在着一些**缺陷**：

1. 它没有使用 XML 格式，而是自己定义了一套格式，相对解析器的重用性较差；而且 DTD 的构建和访问没有标准的编程接口，因而解析器很难简单的解析 DTD 文档。
2. DTD 对元素的类型限制较少；同时其他的约束力也叫弱。
3. DTD 扩展能力较差。
4. 基于正则表达式的 DTD 文档的描述能力有限。

## 1.2 XSD

针对 DTD 的缺陷，W3C 在 2001 年推出 XSD。XSD（XML Schemas Definition）即 XML Schema 语言。XML Schema 本身就是一个 XML文档，使用的是 XML 语法，因此可以很方便的解析 XSD 文档。相对于 DTD，XSD 具有如下**优势**：

1. XML Schema 基于 XML ，没有专门的语法。
2. XML Schema 可以象其他 XML 文件一样解析和处理。
3. XML Schema 比 DTD 提供了更丰富的数据类型。
4. XML Schema 提供可扩充的数据模型。
5. XML Schema 支持综合命名空间。
6. XML Schema 支持属性组。

## 2. getValidationModeForResource

|  |
| --- |
| // XmlBeanDefinitionReader.java  // 禁用验证模式 public static final int VALIDATION\_NONE = XmlValidationModeDetector.VALIDATION\_NONE; // 自动获取验证模式 public static final int VALIDATION\_AUTO = XmlValidationModeDetector.VALIDATION\_AUTO; // DTD 验证模式 public static final int VALIDATION\_DTD = XmlValidationModeDetector.VALIDATION\_DTD; // XSD 验证模式 public static final int VALIDATION\_XSD = XmlValidationModeDetector.VALIDATION\_XSD;   /\*\*  \* 验证模式。默认为自动模式。  \*/ private int validationMode = VALIDATION\_AUTO;   protected int getValidationModeForResource(Resource resource) {  // <1> 获取指定的验证模式  int validationModeToUse = getValidationMode();  // 首先，如果手动指定，则直接返回  if (validationModeToUse != VALIDATION\_AUTO) {  return validationModeToUse;  }  // 其次，自动获取验证模式  int detectedMode = detectValidationMode(resource);  if (detectedMode != VALIDATION\_AUTO) {  return detectedMode;  }  // 最后，使用 VALIDATION\_XSD 做为默认  // Hmm, we didn't get a clear indication... Let's assume XSD,  // since apparently no DTD declaration has been found up until  // detection stopped (before finding the document's root tag).  return VALIDATION\_XSD; } |

* <1> 处，调用 #getValidationMode() 方法，获取指定的验证模式( validationMode )。如果有手动指定，则直接返回。另外，对对于 validationMode 属性的设置和获得的代码，代码如下：

|  |
| --- |
| public void setValidationMode(int validationMode) {  this.validationMode = validationMode; }  public int getValidationMode() {  return this.validationMode; } |

* <2> 处，调用 #detectValidationMode(Resource resource) 方法，自动获取验证模式。代码如下：

|  |
| --- |
| /\*\*  \* XML 验证模式探测器  \*/  private final XmlValidationModeDetector validationModeDetector = new XmlValidationModeDetector();    protected int detectValidationMode(Resource resource) { // 不可读，抛出 BeanDefinitionStoreException 异常  if (resource.isOpen()) {  throw new BeanDefinitionStoreException(  "Passed-in Resource [" + resource + "] contains an open stream: " +  "cannot determine validation mode automatically. Either pass in a Resource " +  "that is able to create fresh streams, or explicitly specify the validationMode " +  "on your XmlBeanDefinitionReader instance.");  }    // 打开 InputStream 流  InputStream inputStream;  try {  inputStream = resource.getInputStream();  } catch (IOException ex) {  throw new BeanDefinitionStoreException(  "Unable to determine validation mode for [" + resource + "]: cannot open InputStream. " +  "Did you attempt to load directly from a SAX InputSource without specifying the " +  "validationMode on your XmlBeanDefinitionReader instance?", ex);  }    // <x> 获取相应的验证模式  try {  return this.validationModeDetector.detectValidationMode(inputStream);  } catch (IOException ex) {  throw new BeanDefinitionStoreException("Unable to determine validation mode for [" +  resource + "]: an error occurred whilst reading from the InputStream.", ex);  }  } |

* + 核心在于 <x> 处，调用 XmlValidationModeDetector#detectValidationMode(InputStream inputStream) 方法，获取相应的验证模式。详细解析，见 [「3. XmlValidationModeDetector」](http://svip.iocoder.cn/Spring/IoC-Validation-Mode-For-Resource/) 中。
* <3> 处，使用 VALIDATION\_XSD 做为默认。

## 3. XmlValidationModeDetector

org.springframework.util.xml.XmlValidationModeDetector ，XML 验证模式探测器。

|  |
| --- |
| public int detectValidationMode(InputStream inputStream) throws IOException {  // Peek into the file to look for DOCTYPE.  BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));  try {  // 是否为 DTD 校验模式。默认为，非 DTD 模式，即 XSD 模式  boolean isDtdValidated = false;  String content;  // <0> 循环，逐行读取 XML 文件的内容  while ((content = reader.readLine()) != null) {  content = consumeCommentTokens(content);  // 跳过，如果是注释，或者  if (this.inComment || !StringUtils.hasText(content)) {  continue;  }  // <1> 包含 DOCTYPE 为 DTD 模式  if (hasDoctype(content)) {  isDtdValidated = true;  break;  }  // <2> hasOpeningTag 方法会校验，如果这一行有 < ，并且 < 后面跟着的是字母，则返回 true 。  if (hasOpeningTag(content)) {  // End of meaningful data...  break;  }  }  // 返回 VALIDATION\_DTD or VALIDATION\_XSD 模式  return (isDtdValidated ? VALIDATION\_DTD : VALIDATION\_XSD);  } catch (CharConversionException ex) {    // <3> 返回 VALIDATION\_AUTO 模式  // Choked on some character encoding...  // Leave the decision up to the caller.  return VALIDATION\_AUTO;  } finally {  reader.close();  } } |

* <0> 处，从代码中看，主要是通过读取 XML 文件的内容，来进行自动判断。
* <1> 处，调用 #hasDoctype(String content) 方法，判断内容中如果包含有 "DOCTYPE“ ，则为 DTD 验证模式。代码如下：

|  |
| --- |
| /\*\*  \* The token in a XML document that declares the DTD to use for validation  \* and thus that DTD validation is being used.  \*/ private static final String DOCTYPE = "DOCTYPE";  private boolean hasDoctype(String content) {  return content.contains(DOCTYPE); } |

* <2> 处，调用 #hasOpeningTag(String content) 方法，判断如果这一行包含 < ，并且 < 紧跟着的是字幕，则为 XSD 验证模式。代码如下：

|  |
| --- |
| /\*\*  \* Does the supplied content contain an XML opening tag. If the parse state is currently  \* in an XML comment then this method always returns false. It is expected that all comment  \* tokens will have consumed for the supplied content before passing the remainder to this method.  \*/ private boolean hasOpeningTag(String content) {  if (this.inComment) {  return false;  }  int openTagIndex = content.indexOf('<');  return (openTagIndex > -1 // < 存在  && (content.length() > openTagIndex + 1) // < 后面还有内容  && Character.isLetter(content.charAt(openTagIndex + 1))); // < 后面的内容是字幕 } |

* <3> 处，如果发生 CharConversionException 异常，则为 VALIDATION\_AUTO 模式。
* 关于 #consumeCommentTokens(String content) 方法，代码比较复杂。感兴趣的胖友可以看看。代码如下：

|  |
| --- |
| /\*\*  \* The token that indicates the start of an XML comment.  \*/ private static final String START\_COMMENT = "<!--";  /\*\*  \* The token that indicates the end of an XML comment.  \*/ private static final String END\_COMMENT = "-->";  /\*\*  \* Consumes all the leading comment data in the given String and returns the remaining content, which  \* may be empty since the supplied content might be all comment data. For our purposes it is only important  \* to strip leading comment content on a line since the first piece of non comment content will be either  \* the DOCTYPE declaration or the root element of the document.  \*/ @Nullable private String consumeCommentTokens(String line) {  // 非注释  if (!line.contains(START\_COMMENT) && !line.contains(END\_COMMENT)) {  return line;  }  String currLine = line;  while ((currLine = consume(currLine)) != null) {  if (!this.inComment && !currLine.trim().startsWith(START\_COMMENT)) {  return currLine;  }  }  return null; }  /\*\*  \* Consume the next comment token, update the "inComment" flag  \* and return the remaining content.  \*/ @Nullable private String consume(String line) {  int index = (this.inComment ? endComment(line) : startComment(line));  return (index == -1 ? null : line.substring(index)); }  /\*\*  \* Try to consume the {@link #START\_COMMENT} token.  \* @see #commentToken(String, String, boolean)  \*/ private int startComment(String line) {  return commentToken(line, START\_COMMENT, true); }  private int endComment(String line) {  return commentToken(line, END\_COMMENT, false); }  /\*\*  \* Try to consume the supplied token against the supplied content and update the  \* in comment parse state to the supplied value. Returns the index into the content  \* which is after the token or -1 if the token is not found.  \*/ private int commentToken(String line, String token, boolean inCommentIfPresent) {  int index = line.indexOf(token);  if (index > - 1) {  this.inComment = inCommentIfPresent;  }  return (index == -1 ? index : index + token.length()); } |

* + 😈 反正老艿艿没细看。哈哈哈哈。如果真看，如下两篇文章，有一定的辅助：
    - [《spring源码（六）–XmlValidationModeDetector（获取xml文档校验模式）》](https://blog.csdn.net/ljz2016/article/details/82686884)
    - [《XmlValidationModeDetector》](https://my.oschina.net/u/3579120/blog/1532852)

## 666. 彩蛋

好了，XML 文件的验证模式分析完毕。下篇，我们来分析 #doLoadBeanDefinitions(InputSource inputSource, Resource resource) 方法的**第 2 个**步骤：获取 Document 实例。

# 【死磕 Spring】—— IoC 之获取 Document 对象

**本文主要基于 Spring 5.0.6.RELEASE**

摘要: 原创出处 <http://cmsblogs.com/?p=2695> 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老艿艿」略作修改，记录在理解过程中，参考的资料。

在 XmlBeanDefinitionReader#doLoadDocument(InputSource inputSource, Resource resource) 方法，中做了两件事情：

* 调用 #getValidationModeForResource(Resource resource) 方法，获取指定资源（xml）的**验证模式**。
  + 上篇博客，我们已经详细解析。
* 调用 DocumentLoader#loadDocument(InputSource inputSource, EntityResolver entityResolver, ErrorHandler errorHandler, int validationMode, boolean namespaceAware) 方法，获取 XML Document 实例。
  + 本篇博客，我们来详细解析。

## 1. DocumentLoader

获取 Document 的策略，由接口 org.springframework.beans.factory.xml.DocumentLoader 定义。代码如下：

FROM 《Spring 源码深度解析》P16 页

定义从资源文件加载到转换为 Document 的功能。

|  |
| --- |
| public interface DocumentLoader {   Document loadDocument(  InputSource inputSource, EntityResolver entityResolver,  ErrorHandler errorHandler, int validationMode, boolean namespaceAware)  throws Exception;  } |

* inputSource 方法参数，加载 Document 的 Resource 资源。
* entityResolver 方法参数，解析文件的解析器。
* errorHandler 方法参数，处理加载 Document 对象的过程的错误。
* validationMode 方法参数，验证模式。
* namespaceAware 方法参数，命名空间支持。如果要提供对 XML 名称空间的支持，则需要值为 true 。

## 1.1 DefaultDocumentLoader

该方法由 DocumentLoader 的默认实现类 org.springframework.beans.factory.xml.DefaultDocumentLoader 实现。代码如下：

|  |
| --- |
| /\*\*  \* Load the {@link Document} at the supplied {@link InputSource} using the standard JAXP-configured  \* XML parser.  \*/ @Override public Document loadDocument(InputSource inputSource, EntityResolver entityResolver,  ErrorHandler errorHandler, int validationMode, boolean namespaceAware) throws Exception {  // <1> 创建 DocumentBuilderFactory  DocumentBuilderFactory factory = createDocumentBuilderFactory(validationMode, namespaceAware);  if (logger.isTraceEnabled()) {  logger.trace("Using JAXP provider [" + factory.getClass().getName() + "]");  }  // <2> 创建 DocumentBuilder  DocumentBuilder builder = createDocumentBuilder(factory, entityResolver, errorHandler);  // <3> 解析 XML InputSource 返回 Document 对象  return builder.parse(inputSource); } |

* 首先，调用 #createDocumentBuilderFactory(...) 方法，创建 javax.xml.parsers.DocumentBuilderFactory 对象。代码如下：

|  |
| --- |
| /\*\*  \* JAXP attribute used to configure the schema language for validation.  \*/ private static final String SCHEMA\_LANGUAGE\_ATTRIBUTE = "http://java.sun.com/xml/jaxp/properties/schemaLanguage"; /\*\*  \* JAXP attribute value indicating the XSD schema language.  \*/ private static final String XSD\_SCHEMA\_LANGUAGE = "http://www.w3.org/2001/XMLSchema"; protected DocumentBuilderFactory createDocumentBuilderFactory(int validationMode, boolean namespaceAware)  throws ParserConfigurationException {  // 创建 DocumentBuilderFactory  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();  factory.setNamespaceAware(namespaceAware); // 设置命名空间支持  if (validationMode != XmlValidationModeDetector.VALIDATION\_NONE) {  factory.setValidating(true); // 开启校验  // XSD 模式下，设置 factory 的属性  if (validationMode == XmlValidationModeDetector.VALIDATION\_XSD) {  // Enforce namespace aware for XSD...  factory.setNamespaceAware(true); // XSD 模式下，强制设置命名空间支持  // 设置 SCHEMA\_LANGUAGE\_ATTRIBUTE  try {  factory.setAttribute(SCHEMA\_LANGUAGE\_ATTRIBUTE, XSD\_SCHEMA\_LANGUAGE);  } catch (IllegalArgumentException ex) {  ParserConfigurationException pcex = new ParserConfigurationException(  "Unable to validate using XSD: Your JAXP provider [" + factory +  "] does not support XML Schema. Are you running on Java 1.4 with Apache Crimson? " +  "Upgrade to Apache Xerces (or Java 1.5) for full XSD support.");  pcex.initCause(ex);  throw pcex;  }  }  }  return factory; } |

* 然后，调用 #createDocumentBuilder(DocumentBuilderFactory factory, EntityResolver entityResolver,ErrorHandler errorHandler) 方法，创建 javax.xml.parsers.DocumentBuilder 对象。代码如下：

|  |
| --- |
| protected DocumentBuilder createDocumentBuilder(DocumentBuilderFactory factory,  @Nullable EntityResolver entityResolver, @Nullable ErrorHandler errorHandler)  throws ParserConfigurationException {  // 创建 DocumentBuilder 对象  DocumentBuilder docBuilder = factory.newDocumentBuilder();  // <x> 设置 EntityResolver 属性  if (entityResolver != null) {  docBuilder.setEntityResolver(entityResolver);  }  // 设置 ErrorHandler 属性  if (errorHandler != null) {  docBuilder.setErrorHandler(errorHandler);  }  return docBuilder; } |

* + 在 <x> 处，设置 DocumentBuilder 的 **EntityResolver** 属性。关于它，在 [「2. EntityResolver」](http://svip.iocoder.cn/Spring/IoC-load-Document/) 会详细解析。
* 最后，调用 DocumentBuilder#parse(InputSource) 方法，解析 InputSource ，返回 Document 对象。

## 2. EntityResolver

通过 DocumentLoader#loadDocument(...) 方法来获取 Document 对象时，有一个方法参数 entityResolver 。该参数是通过 XmlBeanDefinitionReader#getEntityResolver() 方法来获取的。代码如下：

#getEntityResolver() 方法，返回指定的解析器，如果没有指定，则构造一个未指定的默认解析器。

|  |
| --- |
| // XmlBeanDefinitionReader.java  /\*\*  \* EntityResolver 解析器  \*/ @Nullable private EntityResolver entityResolver;  protected EntityResolver getEntityResolver() {  if (this.entityResolver == null) {  // Determine default EntityResolver to use.  ResourceLoader resourceLoader = getResourceLoader();  if (resourceLoader != null) {  this.entityResolver = new ResourceEntityResolver(resourceLoader);  } else {  this.entityResolver = new DelegatingEntityResolver(getBeanClassLoader());  }  }  return this.entityResolver; } |

* 如果 ResourceLoader 不为 null，则根据指定的 ResourceLoader 创建一个 ResourceEntityResolver 对象。
* 如果 ResourceLoader 为 null ，则创建 一个 DelegatingEntityResolver 对象。该 Resolver 委托给默认的 BeansDtdResolver 和 PluggableSchemaResolver 。

## 2.1 子类

上面的方法，一共涉及**四个** EntityResolver 的子类：

* org.springframework.beans.factory.xm.BeansDtdResolver ：实现 EntityResolver 接口，Spring Bean dtd 解码器，用来从 classpath 或者 jar 文件中加载 dtd 。部分代码如下：

|  |
| --- |
| private static final String DTD\_EXTENSION = ".dtd";  private static final String DTD\_NAME = "spring-beans"; |

* org.springframework.beans.factory.xml.PluggableSchemaResolver ，实现 EntityResolver 接口，读取 classpath 下的所有 "META-INF/spring.schemas" 成一个 namespaceURI 与 Schema 文件地址的 map 。代码如下：

|  |
| --- |
| /\*\*  \* The location of the file that defines schema mappings.  \* Can be present in multiple JAR files.  \*  \* 默认 {@link #schemaMappingsLocation} 地址  \*/ public static final String DEFAULT\_SCHEMA\_MAPPINGS\_LOCATION = "META-INF/spring.schemas";  @Nullable private final ClassLoader classLoader;  /\*\*  \* Schema 文件地址  \*/ private final String schemaMappingsLocation;  /\*\* Stores the mapping of schema URL -> local schema path. \*/ @Nullable private volatile Map<String, String> schemaMappings; // namespaceURI 与 Schema 文件地址的映射集合 |

* org.springframework.beans.factory.xml.DelegatingEntityResolver ：实现 EntityResolver 接口，分别代理 dtd 的 BeansDtdResolver 和 xml schemas 的 PluggableSchemaResolver 。代码如下：

|  |
| --- |
| /\*\* Suffix for DTD files. \*/ public static final String DTD\_SUFFIX = ".dtd";  /\*\* Suffix for schema definition files. \*/ public static final String XSD\_SUFFIX = ".xsd";  private final EntityResolver dtdResolver;  private final EntityResolver schemaResolver;  // 默认 public DelegatingEntityResolver(@Nullable ClassLoader classLoader) {  this.dtdResolver = new BeansDtdResolver();  this.schemaResolver = new PluggableSchemaResolver(classLoader); }  // 自定义 public DelegatingEntityResolver(EntityResolver dtdResolver, EntityResolver schemaResolver) {  Assert.notNull(dtdResolver, "'dtdResolver' is required");  Assert.notNull(schemaResolver, "'schemaResolver' is required");  this.dtdResolver = dtdResolver;  this.schemaResolver = schemaResolver; } |

* org.springframework.beans.factory.xml.ResourceEntityResolver ：继承自 DelegatingEntityResolver 类，通过 ResourceLoader 来解析实体的引用。代码如下：

|  |
| --- |
| private final ResourceLoader resourceLoader;  public ResourceEntityResolver(ResourceLoader resourceLoader) {  super(resourceLoader.getClassLoader());  this.resourceLoader = resourceLoader; } |

## 2.2 作用

EntityResolver 的作用就是，通过实现它，应用可以自定义如何**寻找**【验证文件】的逻辑。

FROM 《Spring 源码深度解析》

在 loadDocument 方法中涉及一个参数 EntityResolver ，何为EntityResolver？官网这样解释：如果 SAX 应用程序需要实现自定义处理外部实体，则必须实现此接口并使用 setEntityResolver 方法向SAX 驱动器注册一个实例。也就是说，对于解析一个XML，SAX 首先读取该 XML 文档上的声明，根据声明去寻找相应的 DTD 定义，以便对文档进行一个验证。默认的寻找规则，即通过网络（实现上就是声明的DTD的URI地址）来下载相应的DTD声明，并进行认证。下载的过程是一个漫长的过程，而且当网络中断或不可用时，这里会报错，就是因为相应的DTD声明没有被找到的原因。

EntityResolver 的作用是项目本身就可以提供一个如何寻找 DTD 声明的方法，即由程序来实现寻找 DTD 声明的过程，比如我们将 DTD 文件放到项目中某处，在实现时直接将此文档读取并返回给 SAX 即可。这样就避免了通过网络来寻找相应的声明。

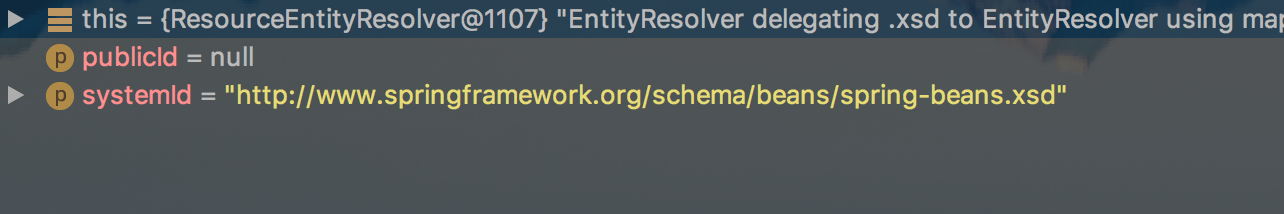
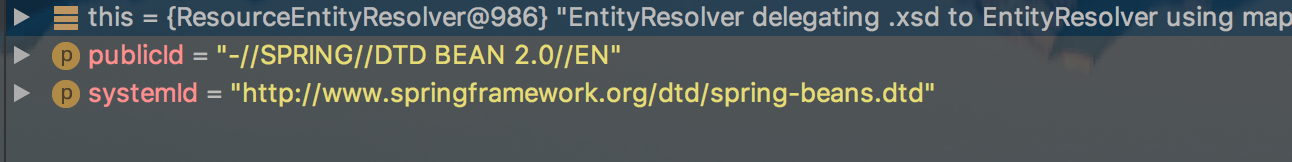
org.xml.sax.EntityResolver 接口，代码如下：

|  |
| --- |
| public interface EntityResolver {   public abstract InputSource resolveEntity (String publicId, String systemId)  throws SAXException, IOException;   } |

接口方法接收两个参数 publicId 和 systemId ，并返回 InputSource 对象。两个参数声明如下：

* publicId ：被引用的外部实体的公共标识符，如果没有提供，则返回 null 。
* systemId ：被引用的外部实体的系统标识符。

这两个参数的实际内容和具体的验证模式的关系如下：

* XSD 验证模式
  + publicId：null
  + systemId：<http://www.springframework.org/schema/beans/spring-beans.xsd>
  + 
* DTD 验证模式
  + publicId：-//SPRING//DTD BEAN 2.0//EN
  + systemId：<http://www.springframework.org/dtd/spring-beans.dtd>
  + 

## 2.3 DelegatingEntityResolver

我们知道在 Spring 中使用 DelegatingEntityResolver 为 EntityResolver 的实现类。#resolveEntity(String publicId, String systemId) 方法，实现如下：

|  |
| --- |
| @Override @Nullable public InputSource resolveEntity(String publicId, @Nullable String systemId) throws SAXException, IOException {  if (systemId != null) {  // DTD 模式  if (systemId.endsWith(DTD\_SUFFIX)) {  return this.dtdResolver.resolveEntity(publicId, systemId);  // XSD 模式  } else if (systemId.endsWith(XSD\_SUFFIX)) {  return this.schemaResolver.resolveEntity(publicId, systemId);  }  }  return null; } |

* 如果是 DTD 验证模式，则使用 BeansDtdResolver 来进行解析
* 如果是 XSD 验证模式，则使用 PluggableSchemaResolver 来进行解析。

## 2.4 BeansDtdResolver

BeansDtdResolver 的解析过程，代码如下：

|  |
| --- |
| /\*\*  \* DTD 文件的后缀  \*/ private static final String DTD\_EXTENSION = ".dtd"; /\*\*  \* Spring Bean DTD 的文件名  \*/ private static final String DTD\_NAME = "spring-beans";  @Override @Nullable public InputSource resolveEntity(String publicId, @Nullable String systemId) throws IOException {  if (logger.isTraceEnabled()) {  logger.trace("Trying to resolve XML entity with public ID [" + publicId +  "] and system ID [" + systemId + "]");  }  // 必须以 .dtd 结尾  if (systemId != null && systemId.endsWith(DTD\_EXTENSION)) {  // 获取最后一个 / 的位置  int lastPathSeparator = systemId.lastIndexOf('/');  // 获取 spring-beans 的位置  int dtdNameStart = systemId.indexOf(DTD\_NAME, lastPathSeparator);  if (dtdNameStart != -1) { // 找到  String dtdFile = DTD\_NAME + DTD\_EXTENSION;  if (logger.isTraceEnabled()) {  logger.trace("Trying to locate [" + dtdFile + "] in Spring jar on classpath");  }  try {  // 创建 ClassPathResource 对象  Resource resource = new ClassPathResource(dtdFile, getClass());  // 创建 InputSource 对象，并设置 publicId、systemId 属性  InputSource source = new InputSource(resource.getInputStream());  source.setPublicId(publicId);  source.setSystemId(systemId);  if (logger.isTraceEnabled()) {  logger.trace("Found beans DTD [" + systemId + "] in classpath: " + dtdFile);  }  return source;  }  catch (IOException ex) {  if (logger.isDebugEnabled()) {  logger.debug("Could not resolve beans DTD [" + systemId + "]: not found in classpath", ex);  }  }  }  }   // 使用默认行为，从网络上下载  // Use the default behavior -> download from website or wherever.  return null; } |

从上面的代码中，我们可以看到，加载 DTD 类型的 BeansDtdResolver#resolveEntity(...) 过程，只是对 systemId 进行了简单的校验（从最后一个 / 开始，内容中是否包含 spring-beans），然后构造一个 InputSource 对象，并设置 publicId、systemId 属性，然后返回。

## 2.5 PluggableSchemaResolver

PluggableSchemaResolver 的解析过程，代码如下:

|  |
| --- |
| @Nullable private final ClassLoader classLoader;  /\*\*  \* Schema 文件地址  \*/ private final String schemaMappingsLocation;  /\*\* Stores the mapping of schema URL -> local schema path. \*/ @Nullable private volatile Map<String, String> schemaMappings; // namespaceURI 与 Schema 文件地址的映射集合  @Override @Nullable public InputSource resolveEntity(String publicId, @Nullable String systemId) throws IOException {  if (logger.isTraceEnabled()) {  logger.trace("Trying to resolve XML entity with public id [" + publicId +  "] and system id [" + systemId + "]");  }   if (systemId != null) {  // 获得 Resource 所在位置  String resourceLocation = getSchemaMappings().get(systemId);  if (resourceLocation != null) {  // 创建 ClassPathResource  Resource resource = new ClassPathResource(resourceLocation, this.classLoader);  try {  // 创建 InputSource 对象，并设置 publicId、systemId 属性  InputSource source = new InputSource(resource.getInputStream());  source.setPublicId(publicId);  source.setSystemId(systemId);  if (logger.isTraceEnabled()) {  logger.trace("Found XML schema [" + systemId + "] in classpath: " + resourceLocation);  }  return source;  }  catch (FileNotFoundException ex) {  if (logger.isDebugEnabled()) {  logger.debug("Could not find XML schema [" + systemId + "]: " + resource, ex);  }  }  }  }  return null; } |

* 首先调用 #getSchemaMappings() 方法，获取一个映射表(systemId 与其在本地的对照关系)。代码如下：

|  |
| --- |
| private Map<String, String> getSchemaMappings() {  Map<String, String> schemaMappings = this.schemaMappings;  // 双重检查锁，实现 schemaMappings 单例  if (schemaMappings == null) {  synchronized (this) {  schemaMappings = this.schemaMappings;  if (schemaMappings == null) {  if (logger.isTraceEnabled()) {  logger.trace("Loading schema mappings from [" + this.schemaMappingsLocation + "]");  }  try {  // 以 Properties 的方式，读取 schemaMappingsLocation  Properties mappings = PropertiesLoaderUtils.loadAllProperties(this.schemaMappingsLocation, this.classLoader);  if (logger.isTraceEnabled()) {  logger.trace("Loaded schema mappings: " + mappings);  }  // 将 mappings 初始化到 schemaMappings 中  schemaMappings = new ConcurrentHashMap<>(mappings.size());  CollectionUtils.mergePropertiesIntoMap(mappings, schemaMappings);  this.schemaMappings = schemaMappings;  } catch (IOException ex) {  throw new IllegalStateException(  "Unable to load schema mappings from location [" + this.schemaMappingsLocation + "]", ex);  }  }  }  }  return schemaMappings; } |

* + 映射表如下（**部分**）:
* 然后，根据传入的 systemId 获取该 systemId 在本地的路径 resourceLocation 。
* 最后，根据 resourceLocation ，构造 InputSource 对象。

## 2.6 ResourceEntityResolver

ResourceEntityResolver 的解析过程，代码如下:

|  |
| --- |
| private final ResourceLoader resourceLoader;  @Override @Nullable public InputSource resolveEntity(String publicId, @Nullable String systemId) throws SAXException, IOException {  // 调用父类的方法，进行解析  InputSource source = super.resolveEntity(publicId, systemId);  // 解析失败，resourceLoader 进行解析  if (source == null && systemId != null) {  // 获得 resourcePath ，即 Resource 资源地址  String resourcePath = null;  try {  String decodedSystemId = URLDecoder.decode(systemId, "UTF-8"); // 使用 UTF-8 ，解码 systemId  String givenUrl = new URL(decodedSystemId).toString(); // 转换成 URL 字符串  // 解析文件资源的相对路径（相对于系统根路径）  String systemRootUrl = new File("").toURI().toURL().toString();  // Try relative to resource base if currently in system root.  if (givenUrl.startsWith(systemRootUrl)) {  resourcePath = givenUrl.substring(systemRootUrl.length());  }  } catch (Exception ex) {  // Typically a MalformedURLException or AccessControlException.  if (logger.isDebugEnabled()) {  logger.debug("Could not resolve XML entity [" + systemId + "] against system root URL", ex);  }  // No URL (or no resolvable URL) -> try relative to resource base.  resourcePath = systemId;  }  if (resourcePath != null) {  if (logger.isTraceEnabled()) {  logger.trace("Trying to locate XML entity [" + systemId + "] as resource [" + resourcePath + "]");  }  // 获得 Resource 资源  Resource resource = this.resourceLoader.getResource(resourcePath);  // 创建 InputSource 对象  source = new InputSource(resource.getInputStream());  // 设置 publicId 和 systemId 属性  source.setPublicId(publicId);  source.setSystemId(systemId);  if (logger.isDebugEnabled()) {  logger.debug("Found XML entity [" + systemId + "]: " + resource);  }  }  }  return source; } |

* 首先，调用**父类**的方法，进行解析。
* 如果失败，使用 resourceLoader ，尝试读取 systemId 对应的 Resource 资源。

## 2.7 自定义 EntityResolver

老艿艿：本小节，为**选读**内容。

#getEntityResolver() 方法返回 EntityResolver 对象。那么怎么进行自定义 EntityResolver 呢?

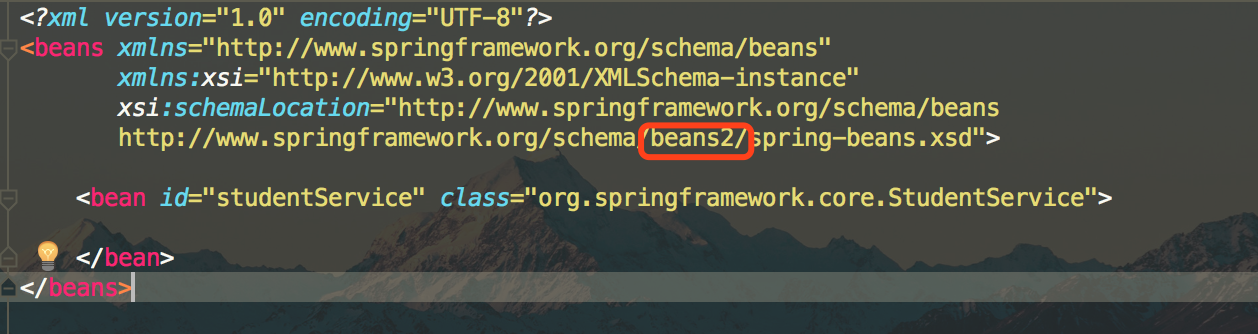
If a SAX application needs to implement customized handling for external entities, it must implement this interface and register an instance with the SAX driver using the setEntityResolver method.

就是说：如果 SAX 应用程序需要实现自定义处理外部实体，则必须实现此接口，并使用 #setEntityResolver(EntityResolver entityResolver) 方法，向 SAX 驱动器注册一个 EntityResolver 实例。

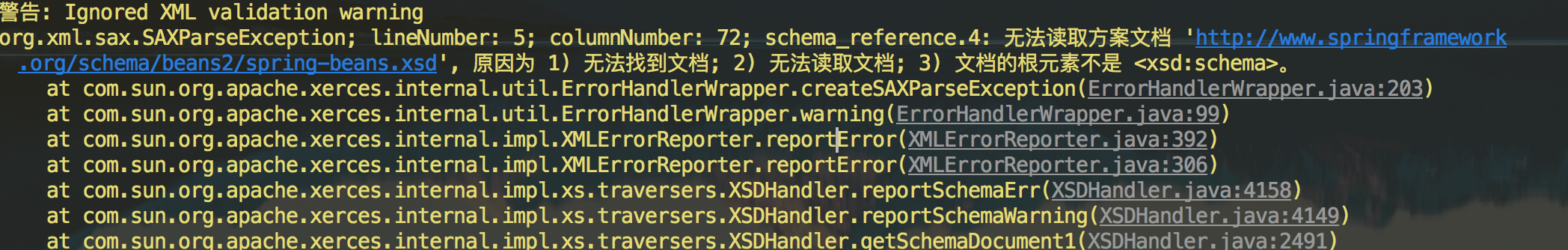
示例如下：

|  |
| --- |
| public class MyResolver implements EntityResolver {   @Override  public InputSource resolveEntity(String publicId, String systemId) {  if (systemId.equals("http://www.myhost.com/today")) {  MyReader reader = new MyReader();  return new InputSource(reader);  } else {  // use the default behaviour  return null;  }  }   } |

我们首先将 "spring-student.xml" 文件中的 XSD 声明的地址改掉，如下：



如果我们再次运行，则会报如下错误：



从上面的错误可以看到，是在进行文档验证时，无法根据声明找到 XSD 验证文件而导致无法进行 XML 文件验证。在 [《【死磕 Spring】—— IoC 之获取验证模型》](http://svip.iocoder.cn/Spring/IoC-Validation-Mode-For-Resource) 中讲到，如果要解析一个 XML 文件，SAX 首先会读取该 XML 文档上的声明，然后根据声明去寻找相应的 DTD 定义，以便对文档进行验证。**默认的加载规则是通过网络方式下载验证文件**，而在实际生产环境中我们会遇到网络中断或者不可用状态，那么就应用就会因为无法下载验证文件而报错。

## 666. 彩蛋

是不是看到此处，有点懵逼，不是说好了分享**获取 Document 对象**，结果内容主要是 EntityResolver 呢？因为，从 XML 中获取 Document 对象，已经有 javax.xml 库进行解析。而 EntityResolver 的重点，是在于如何获取【验证文件】，从而验证用户写的 XML 是否通过验证。

# 【死磕 Spring】—— IoC 之注册 BeanDefinitions

**本文主要基于 Spring 5.0.6.RELEASE**

摘要: 原创出处 <http://cmsblogs.com/?p=2697> 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老艿艿」略作修改，记录在理解过程中，参考的资料。

获取 XML Document 对象后，会根据该对象和 Resource 资源对象调用 XmlBeanDefinitionReader#registerBeanDefinitions(Document doc, Resource resource) 方法，开始注册 BeanDefinitions 之旅。代码如下：

|  |
| --- |
| // AbstractBeanDefinitionReader.java private final BeanDefinitionRegistry registry;  // XmlBeanDefinitionReader.java public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {  // <1> 创建 BeanDefinitionDocumentReader 对象  BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();  // <2> 获取已注册的 BeanDefinition 数量  int countBefore = getRegistry().getBeanDefinitionCount();  // <3> 创建 XmlReaderContext 对象  // <4> 注册 BeanDefinition  documentReader.registerBeanDefinitions(doc, createReaderContext(resource));  // 计算新注册的 BeanDefinition 数量  return getRegistry().getBeanDefinitionCount() - countBefore; } |

* <1> 处，调用 #createBeanDefinitionDocumentReader() 方法，实例化 BeanDefinitionDocumentReader 对象。

FROM 《Spring 源码深度解析》P16 页

定义读取 Document 并注册 BeanDefinition 功能

* <2> 处，调用 BeanDefinitionRegistry#getBeanDefinitionCount() 方法，获取**已注册**的 BeanDefinition 数量。
* <3> 处，调用 #createReaderContext(Resource resource) 方法，创建 XmlReaderContext 对象。
* <4> 处，调用 BeanDefinitionDocumentReader#registerBeanDefinitions(Document doc, XmlReaderContext readerContext) 方法，读取 XML 元素，注册 BeanDefinition 们。
* <5> 处，计**算新注册**的 BeanDefinition 数量。

## 1. createBeanDefinitionDocumentReader

#createBeanDefinitionDocumentReader()，实例化 BeanDefinitionDocumentReader 对象。代码如下：

|  |
| --- |
| /\*\*  \* documentReader 的类  \*  \* @see #createBeanDefinitionDocumentReader()   \*/ private Class<? extends BeanDefinitionDocumentReader> documentReaderClass = DefaultBeanDefinitionDocumentReader.class;  protected BeanDefinitionDocumentReader createBeanDefinitionDocumentReader() {  return BeanUtils.instantiateClass(this.documentReaderClass); } |

* documentReaderClass 的默认值为 DefaultBeanDefinitionDocumentReader.class 。关于它，我们在后续的文章，详细解析。

## 2. registerBeanDefinitions

BeanDefinitionDocumentReader#registerBeanDefinitions(Document doc, XmlReaderContext readerContext) 方法，注册 BeanDefinition ，在接口 BeanDefinitionDocumentReader 中定义。代码如下：

|  |
| --- |
| public interface BeanDefinitionDocumentReader {   /\*\*  \* Read bean definitions from the given DOM document and  \* register them with the registry in the given reader context.  \* @param doc the DOM document  \* @param readerContext the current context of the reader  \* (includes the target registry and the resource being parsed)  \* @throws BeanDefinitionStoreException in case of parsing errors  \*/  void registerBeanDefinitions(Document doc, XmlReaderContext readerContext)  throws BeanDefinitionStoreException;  } |

**从给定的 Document 对象中解析定义的 BeanDefinition 并将他们注册到注册表中**。方法接收两个参数：

* doc 方法参数：待解析的 Document 对象。
* readerContext 方法，解析器的当前上下文，包括目标注册表和被解析的资源。它是根据 Resource 来创建的，见 [「3. createReaderContext」](http://svip.iocoder.cn/Spring/IoC-register-BeanDefinitions/) 。

## 2.1 DefaultBeanDefinitionDocumentReader

BeanDefinitionDocumentReader 有且只有一个默认实现类 DefaultBeanDefinitionDocumentReader 。它对 #registerBeanDefinitions(...) 方法的实现代码如下：

DefaultBeanDefinitionDocumentReader 对该方法提供了实现：

|  |
| --- |
| @Nullable private XmlReaderContext readerContext;  @Nullable private BeanDefinitionParserDelegate delegate;   /\*\*  \* This implementation parses bean definitions according to the "spring-beans" XSD  \* (or DTD, historically).  \* <p>Opens a DOM Document; then initializes the default settings  \* specified at the {@code <beans/>} level; then parses the contained bean definitions.  \*/ @Override public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {  this.readerContext = readerContext;  // 获得 XML Document Root Element  // 执行注册 BeanDefinition  doRegisterBeanDefinitions(doc.getDocumentElement()); }  /\*\*  \* Register each bean definition within the given root {@code <beans/>} element.  \*/ @SuppressWarnings("deprecation") // for Environment.acceptsProfiles(String...) protected void doRegisterBeanDefinitions(Element root) {  // Any nested <beans> elements will cause recursion in this method. In  // order to propagate and preserve <beans> default-\* attributes correctly,  // keep track of the current (parent) delegate, which may be null. Create  // the new (child) delegate with a reference to the parent for fallback purposes,  // then ultimately reset this.delegate back to its original (parent) reference.  // this behavior emulates a stack of delegates without actually necessitating one.  // 记录老的 BeanDefinitionParserDelegate 对象  BeanDefinitionParserDelegate parent = this.delegate;  // <1> 创建 BeanDefinitionParserDelegate 对象，并进行设置到 delegate  this.delegate = createDelegate(getReaderContext(), root, parent);  // <2> 检查 <beans /> 根标签的命名空间是否为空，或者是 http://www.springframework.org/schema/beans  if (this.delegate.isDefaultNamespace(root)) {  // <2.1> 处理 profile 属性。可参见《Spring3自定义环境配置 <beans profile="">》http://nassir.iteye.com/blog/1535799  String profileSpec = root.getAttribute(PROFILE\_ATTRIBUTE);  if (StringUtils.hasText(profileSpec)) {  // <2.2> 使用分隔符切分，可能有多个 profile 。  String[] specifiedProfiles = StringUtils.tokenizeToStringArray(  profileSpec, BeanDefinitionParserDelegate.MULTI\_VALUE\_ATTRIBUTE\_DELIMITERS);  // <2.3> 如果所有 profile 都无效，则不进行注册  // We cannot use Profiles.of(...) since profile expressions are not supported  // in XML config. See SPR-12458 for details.  if (!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {  if (logger.isDebugEnabled()) {  logger.debug("Skipped XML bean definition file due to specified profiles [" + profileSpec +  "] not matching: " + getReaderContext().getResource());  }  return;  }  }  }   // <3> 解析前处理  preProcessXml(root);  // <4> 解析  parseBeanDefinitions(root, this.delegate);  // <5> 解析后处理  postProcessXml(root);   // 设置 delegate 回老的 BeanDefinitionParserDelegate 对象  this.delegate = parent; } |

* <1> 处，创建 BeanDefinitionParserDelegate 对象，并进行设置到 delegate 。BeanDefinitionParserDelegate 是一个重要的类，它负责**解析 BeanDefinition**。代码如下：

FROM 《Spring 源码深度解析》P16

定义解析 XML Element 的各种方法

|  |
| --- |
| protected BeanDefinitionParserDelegate createDelegate(  XmlReaderContext readerContext, Element root, @Nullable BeanDefinitionParserDelegate parentDelegate) {  // 创建 BeanDefinitionParserDelegate 对象  BeanDefinitionParserDelegate delegate = new BeanDefinitionParserDelegate(readerContext);  // 初始化默认  delegate.initDefaults(root, parentDelegate);  return delegate; } |

* <2> 处，检查 <beans /> **根**标签的命名空间是否为空，或者是 <http://www.springframework.org/schema/beans> 。
  + <2.1> 处，判断是否 <beans /> 上配置了 profile 属性。不了解这块的胖友，可以看下 [《《Spring3自定义环境配置 》》](http://nassir.iteye.com/blog/1535799) 。
  + <2.2> 处，使用分隔符切分，可能有**多个** profile 。
  + <2.3> 处，判断，如果所有 profile 都无效，则 return 不进行注册。
* <4> 处，调用 #parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) 方法，进行解析逻辑。详细解析，见 [「3.1 parseBeanDefinitions」](http://svip.iocoder.cn/Spring/IoC-register-BeanDefinitions/) 。
* <3> / <5> 处，解析**前后**的处理，目前这两个方法都是空实现，交由子类来实现。代码如下：

|  |
| --- |
| protected void preProcessXml(Element root) {}  protected void postProcessXml(Element root) {} |

### 2.1.1 parseBeanDefinitions

#parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) 方法，进行解析逻辑。代码如下：

|  |
| --- |
| /\*\*  \* Parse the elements at the root level in the document:  \* "import", "alias", "bean".  \* @param root the DOM root element of the document  \*/ protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {  // <1> 如果根节点使用默认命名空间，执行默认解析  if (delegate.isDefaultNamespace(root)) {  // 遍历子节点  NodeList nl = root.getChildNodes();  for (int i = 0; i < nl.getLength(); i++) {  Node node = nl.item(i);  if (node instanceof Element) {  Element ele = (Element) node;  // <1> 如果该节点使用默认命名空间，执行默认解析  if (delegate.isDefaultNamespace(ele)) {  parseDefaultElement(ele, delegate);  // 如果该节点非默认命名空间，执行自定义解析  } else {  delegate.parseCustomElement(ele);  }  }  }  // 如果根节点非默认命名空间，执行自定义解析  } else {  delegate.parseCustomElement(root);  } } |

* Spring 有**两种** Bean 声明方式：
  + 配置文件式声明：<bean id="studentService" class="org.springframework.core.StudentService" /> 。对应 <1> 处。
  + 自定义注解方式：<tx:annotation-driven> 。对应 <2> 处。
* <1> 处，如果**根**节点或**子**节点**使用**默认命名空间，调用 #parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) 方法，执行默认解析。代码如下：

|  |
| --- |
| private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {  if (delegate.nodeNameEquals(ele, IMPORT\_ELEMENT)) { // import  importBeanDefinitionResource(ele);  } else if (delegate.nodeNameEquals(ele, ALIAS\_ELEMENT)) { // alias  processAliasRegistration(ele);  } else if (delegate.nodeNameEquals(ele, BEAN\_ELEMENT)) { // bean  processBeanDefinition(ele, delegate);  } else if (delegate.nodeNameEquals(ele, NESTED\_BEANS\_ELEMENT)) { // beans  // recurse  doRegisterBeanDefinitions(ele);  } } |

* + 详细的解析，见后续文章。
* <2> 处，如果**根**节点或**子**节点**不使用**默认命名空间，调用 BeanDefinitionParserDelegate#parseCustomElement(Element ele) 方法，执行**自定义**解析。详细的解析，见后续文章。

## 3. createReaderContext

#createReaderContext(Resource resource) 方法，创建 XmlReaderContext 对象。代码如下：

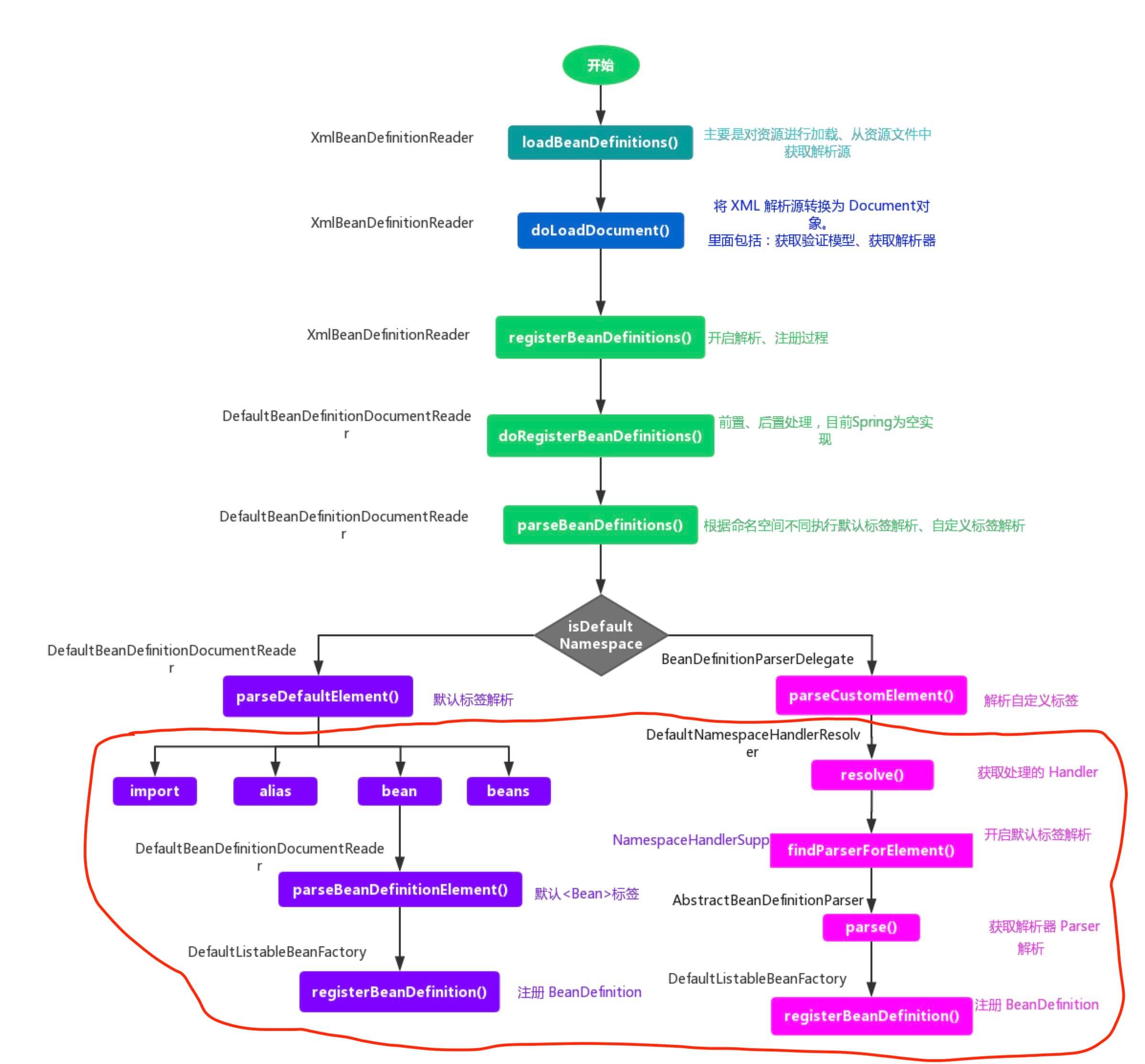
|  |
| --- |
| private ProblemReporter problemReporter = new FailFastProblemReporter();  private ReaderEventListener eventListener = new EmptyReaderEventListener();  private SourceExtractor sourceExtractor = new NullSourceExtractor();  @Nullable private NamespaceHandlerResolver namespaceHandlerResolver;  /\*\*  \* Create the {@link XmlReaderContext} to pass over to the document reader.  \*/ public XmlReaderContext createReaderContext(Resource resource) {  return new XmlReaderContext(resource, this.problemReporter, this.eventListener,  this.sourceExtractor, this, getNamespaceHandlerResolver()); } |

关于 XmlReaderContext 的详细解析，见后续文章。

## 4. 小结

至此，XmlBeanDefinitionReader#doLoadBeanDefinitions(InputSource inputSource, Resource resource) 方法中，做的三件事情已经全部分析完毕，下面将对 **BeanDefinition 的解析过程**做详细分析说明。

另外，XmlBeanDefinitionReader#doLoadBeanDefinitions(InputSource inputSource, Resource resource) 方法，整体时序图如下：



* 红框部分，就是 **BeanDefinition 的解析过程**。

# 【死磕 Spring】—— IoC 之解析Bean：解析 import 标签

**本文主要基于 Spring 5.0.6.RELEASE**

摘要: 原创出处 <http://cmsblogs.com/?p=2724> 「小明哥」，谢谢！

作为「小明哥」的忠实读者，「老艿艿」略作修改，记录在理解过程中，参考的资料。

在博客 [【死磕 Spring】—— IoC 之注册 BeanDefinitions](http://svip.iocoder.cn/Spring/IoC-register-BeanDefinitions) 中分析到，Spring 中有两种解析 Bean 的方式：

* 如果根节点或者子节点采用默认命名空间的话，则调用 #parseDefaultElement(...) 方法，进行**默认**标签解析
* 否则，调用 BeanDefinitionParserDelegate#parseCustomElement(...) 方法，进行**自定义**解析。

所以，以下博客就这两个方法进行详细分析说明。而本文，先从**默认标签**解析过程开始。代码如下：

|  |
| --- |
| // DefaultBeanDefinitionDocumentReader.java  public static final String IMPORT\_ELEMENT = "import"; public static final String ALIAS\_ATTRIBUTE = "alias"; public static final String BEAN\_ELEMENT = BeanDefinitionParserDelegate.BEAN\_ELEMENT; public static final String NESTED\_BEANS\_ELEMENT = "beans";  private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {  if (delegate.nodeNameEquals(ele, IMPORT\_ELEMENT)) { // import  importBeanDefinitionResource(ele);  } else if (delegate.nodeNameEquals(ele, ALIAS\_ELEMENT)) { // alias  processAliasRegistration(ele);  } else if (delegate.nodeNameEquals(ele, BEAN\_ELEMENT)) { // bean  processBeanDefinition(ele, delegate);  } else if (delegate.nodeNameEquals(ele, NESTED\_BEANS\_ELEMENT)) { // beans  // recurse  doRegisterBeanDefinitions(ele);  } } |

该方法的功能一目了然，分别是对四种不同的标签进行解析，分别是 import、alias、bean、beans 。咱门从第一个标签 import 开始。

## 1. import 示例

经历过 Spring 配置文件的小伙伴都知道，如果工程比较大，配置文件的维护会让人觉得恐怖，文件太多了，想象将所有的配置都放在一个 spring.xml 配置文件中，哪种后怕感是不是很明显？

所有针对这种情况 Spring 提供了一个分模块的思路，利用 import 标签，例如我们可以构造一个这样的 spring.xml 。

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

spring.xml 配置文件中，使用 import 标签的方式导入其他模块的配置文件。

* 如果有配置需要修改直接修改相应配置文件即可。
* 若有新的模块需要引入直接增加 import 即可。

这样大大简化了配置后期维护的复杂度，同时也易于管理。

## 2. importBeanDefinitionResource

Spring 使用 #importBeanDefinitionResource(Element ele) 方法，完成对 import 标签的解析。

|  |
| --- |
| // DefaultBeanDefinitionDocumentReader.java  /\*\*  \* Parse an "import" element and load the bean definitions  \* from the given resource into the bean factory.  \*/ protected void importBeanDefinitionResource(Element ele) {  // <1> 获取 resource 的属性值  String location = ele.getAttribute(RESOURCE\_ATTRIBUTE);  // 为空，直接退出  if (!StringUtils.hasText(location)) {  getReaderContext().error("Resource location must not be empty", ele); // 使用 problemReporter 报错  return;  }   // <2> 解析系统属性，格式如 ："${user.dir}"  // Resolve system properties: e.g. "${user.dir}"  location = getReaderContext().getEnvironment().resolveRequiredPlaceholders(location);   // 实际 Resource 集合，即 import 的地址，有哪些 Resource 资源  Set<Resource> actualResources = new LinkedHashSet<>(4);   // <3> 判断 location 是相对路径还是绝对路径  // Discover whether the location is an absolute or relative URI  boolean absoluteLocation = false;  try {  absoluteLocation = ResourcePatternUtils.isUrl(location) || ResourceUtils.toURI(location).isAbsolute();  } catch (URISyntaxException ex) {  // cannot convert to an URI, considering the location relative  // unless it is the well-known Spring prefix "classpath\*:"  }   // Absolute or relative?  // <4> 绝对路径  if (absoluteLocation) {  try {  // 添加配置文件地址的 Resource 到 actualResources 中，并加载相应的 BeanDefinition 们  int importCount = getReaderContext().getReader().loadBeanDefinitions(location, actualResources);  if (logger.isTraceEnabled()) {  logger.trace("Imported " + importCount + " bean definitions from URL location [" + location + "]");  }  } catch (BeanDefinitionStoreException ex) {  getReaderContext().error(  "Failed to import bean definitions from URL location [" + location + "]", ele, ex);  }  // <5> 相对路径  } else {  // No URL -> considering resource location as relative to the current file.  try {  int importCount;  // 创建相对地址的 Resource  Resource relativeResource = getReaderContext().getResource().createRelative(location);  // 存在  if (relativeResource.exists()) {  // 加载 relativeResource 中的 BeanDefinition 们  importCount = getReaderContext().getReader().loadBeanDefinitions(relativeResource);  // 添加到 actualResources 中  actualResources.add(relativeResource);  // 不存在  } else {  // 获得根路径地址  String baseLocation = getReaderContext().getResource().getURL().toString();  // 添加配置文件地址的 Resource 到 actualResources 中，并加载相应的 BeanDefinition 们  importCount = getReaderContext().getReader().loadBeanDefinitions(  StringUtils.applyRelativePath(baseLocation, location) /\* 计算绝对路径 \*/, actualResources);  }  if (logger.isTraceEnabled()) {  logger.trace("Imported " + importCount + " bean definitions from relative location [" + location + "]");  }  } catch (IOException ex) {  getReaderContext().error("Failed to resolve current resource location", ele, ex);  } catch (BeanDefinitionStoreException ex) {  getReaderContext().error(  "Failed to import bean definitions from relative location [" + location + "]", ele, ex);  }  }  // <6> 解析成功后，进行监听器激活处理  Resource[] actResArray = actualResources.toArray(new Resource[0]);  getReaderContext().fireImportProcessed(location, actResArray, extractSource(ele)); } |

解析 import 标签的过程较为清晰，整个过程如下：

* <1> 处，获取 source 属性的值，该值表示资源的路径。
  + <2> 处，解析路径中的系统属性，如 "${user.dir}" 。
* <3> 处，判断资源路径 location 是绝对路径还是相对路径。详细解析，见 [「2.1 判断路径」](http://svip.iocoder.cn/Spring/IoC-parse-BeanDefinitions-for-import/) 。
  + <4> 处，如果是绝对路径，则调递归调用 Bean 的解析过程，进行另一次的解析。详细解析，见 [「2.2 处理绝对路径」](http://svip.iocoder.cn/Spring/IoC-parse-BeanDefinitions-for-import/) 。
  + <5> 处，如果是相对路径，则先计算出绝对路径得到 Resource，然后进行解析。详细解析，见 [「2.3 处理相对路径」](http://svip.iocoder.cn/Spring/IoC-parse-BeanDefinitions-for-import/) 。
* <6> 处，通知监听器，完成解析。

## 2.1 判断路径

通过以下代码，来判断 location 是为相对路径还是绝对路径：

|  |
| --- |
| absoluteLocation = ResourcePatternUtils.isUrl(location) // <1>  || ResourceUtils.toURI(location).isAbsolute(); // <2> |

判断绝对路径的规则如下：

* <1> 以 classpath\*: 或者 classpath: 开头的为绝对路径。
* <1> 能够通过该 location 构建出 java.net.URL 为绝对路径。
* <2> 根据 location 构造 java.net.URI 判断调用 #isAbsolute() 方法，判断是否为绝对路径。

## 2.2 处理绝对路径

如果 location 为绝对路径，则调用 #loadBeanDefinitions(String location, Set<Resource> actualResources)， 方法。该方法在 org.springframework.beans.factory.support.AbstractBeanDefinitionReader 中定义，代码如下：

|  |
| --- |
| /\*\*  \* Load bean definitions from the specified resource location.  \* <p>The location can also be a location pattern, provided that the  \* ResourceLoader of this bean definition reader is a ResourcePatternResolver.  \* @param location the resource location, to be loaded with the ResourceLoader  \* (or ResourcePatternResolver) of this bean definition reader  \* @param actualResources a Set to be filled with the actual Resource objects  \* that have been resolved during the loading process. May be {@code null}  \* to indicate that the caller is not interested in those Resource objects.  \* @return the number of bean definitions found  \* @throws BeanDefinitionStoreException in case of loading or parsing errors  \* @see #getResourceLoader()  \* @see #loadBeanDefinitions(org.springframework.core.io.Resource)  \* @see #loadBeanDefinitions(org.springframework.core.io.Resource[])  \*/ public int loadBeanDefinitions(String location, @Nullable Set<Resource> actualResources) throws BeanDefinitionStoreException {  // 获得 ResourceLoader 对象  ResourceLoader resourceLoader = getResourceLoader();  if (resourceLoader == null) {  throw new BeanDefinitionStoreException(  "Cannot load bean definitions from location [" + location + "]: no ResourceLoader available");  }   if (resourceLoader instanceof ResourcePatternResolver) {  // Resource pattern matching available.  try {  // 获得 Resource 数组，因为 Pattern 模式匹配下，可能有多个 Resource 。例如说，Ant 风格的 location  Resource[] resources = ((ResourcePatternResolver) resourceLoader).getResources(location);  // 加载 BeanDefinition 们  int count = loadBeanDefinitions(resources);  // 添加到 actualResources 中  if (actualResources != null) {  Collections.addAll(actualResources, resources);  }  if (logger.isTraceEnabled()) {  logger.trace("Loaded " + count + " bean definitions from location pattern [" + location + "]");  }  return count;  } catch (IOException ex) {  throw new BeanDefinitionStoreException(  "Could not resolve bean definition resource pattern [" + location + "]", ex);  }  } else {  // Can only load single resources by absolute URL.  // 获得 Resource 对象，  Resource resource = resourceLoader.getResource(location);  // 加载 BeanDefinition 们  int count = loadBeanDefinitions(resource);  // 添加到 actualResources 中  if (actualResources != null) {  actualResources.add(resource);  }  if (logger.isTraceEnabled()) {  logger.trace("Loaded " + count + " bean definitions from location [" + location + "]");  }  return count;  } } |

整个逻辑比较简单：

* 首先，获取 ResourceLoader 对象。
* 然后，根据不同的 ResourceLoader 执行不同的逻辑，主要是可能存在多个 Resource 。
* 最终，都会回归到 XmlBeanDefinitionReader#loadBeanDefinitions(Resource... resources) 方法，所以这是一个递归的过程。
* 另外，获得到的 Resource 的对象或数组，都会添加到 actualResources 中。

## 2.3 处理相对路径

如果 location 是相对路径，则会根据相应的 Resource 计算出相应的相对路径的 Resource 对象 ，然后：

* 若该 Resource 存在，则调用 XmlBeanDefinitionReader#loadBeanDefinitions() 方法，进行 BeanDefinition 加载。
* 否则，构造一个绝对 location( 即 StringUtils.applyRelativePath(baseLocation, location) 处的代码)，并调用 #loadBeanDefinitions(String location, Set<Resource> actualResources) 方法，**与绝对路径过程一样**。

## 3. 小结

至此，import 标签解析完毕，整个过程比较清晰明了：**获取 source 属性值，得到正确的资源路径，然后调用 XmlBeanDefinitionReader#loadBeanDefinitions(Resource... resources) 方法，进行递归的 BeanDefinition 加载**。