Spring 学习笔记

<https://segmentfault.com/a/1190000015434853>

常见面试题

<https://mp.weixin.qq.com/s/RbfLWmzpvjpwVYsa8S-uJg>

<https://mp.weixin.qq.com/s/_VXbc-qhAXx96HO-Va-BXg>

15个经典的Spring面试常见问题： <https://juejin.im/post/5cf7ca6d6fb9a07ef71062d7>

# 注入依赖

## Spring IoC中到底有几种注入方式？

1.构造函数注入； 2. 属性注入； 3.注解注入，比如Autowirted, Resource

4.自动装配：三种模式 byType()、byName()、constructor()

## 为什么spring中的推荐使用构造器注入

<https://www.cnblogs.com/joemsu/p/7688307.html>

* 确保属性不可变，可以final修饰
* 确保获得的属性是完全初始化状态的对象
* 避免循环依赖

## 循环依赖

<https://juejin.im/entry/5eab6b5ff265da7bee7c8afd>

<https://www.bilibili.com/read/cv5945731>

<https://mp.weixin.qq.com/s/vOsMqDiqY-wbLJYRe6tbjg>

为啥prototype不支持循环依赖？

原型(Prototype)的场景是不支持循环依赖的，通常会走到AbstractBeanFactory类中下面的判断，抛出异常。

if (isPrototypeCurrentlyInCreation(beanName)) {

throw new BeanCurrentlyInCreationException(beanName);

}

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

这就套娃了, 你猜是先StackOverflow还是OutOfMemory？

Spring怕你不好猜，就先抛出了BeanCurrentlyInCreationException

构造函数为啥不能循环依赖？

构造函数方式的注入能保证单例对象的安全性（某种程度上）。

对象都没创建出来呢，三级缓存都找不到的，直接报错

public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry implements SingletonBeanRegistry {

//1级缓存 用于存放 已经属性赋值 初始化后的 单列BEAN

private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);

//2级缓存 用于存在已经实例化，还未做代理属性赋值操作的 单例

private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);

//3级缓存 存储单例BEAN的工厂

private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);

//已经注册的单例池里的beanName

private final Set<String> registeredSingletons = new LinkedHashSet<>(256);

//正在创建中的beanName

private final Set<String> singletonsCurrentlyInCreation =

Collections.newSetFromMap(new ConcurrentHashMap<>(16));

//缓存查找bean 如果1级没有，从2级获取,也没有,从3级创建放入2级

protected Object getSingleton(String beanName, boolean allowEarlyReference) {

Object singletonObject = this.singletonObjects.get(beanName); //1级

if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {

synchronized (this.singletonObjects) {

singletonObject = this.earlySingletonObjects.get(beanName); //2级

if (singletonObject == null && allowEarlyReference) {

//3级缓存 在doCreateBean中创建了bean的实例后，封装ObjectFactory放入缓存的

ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);

if (singletonFactory != null) {

//创建未赋值的bean

singletonObject = singletonFactory.getObject();

//放入到二级缓存

this.earlySingletonObjects.put(beanName, singletonObject);

//从三级缓存删除

this.singletonFactories.remove(beanName);

}

}

}

}

return singletonObject;

}

}

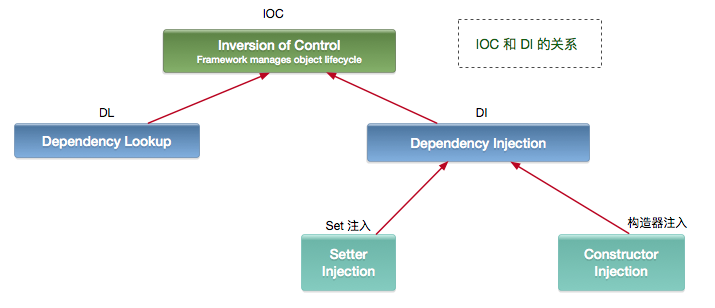
# IOC

## 广义的 IOC

IoC(Inversion of Control) 控制反转，即“不用打电话过来，我们会打给你”。

两种实现： 依赖查找（DL）和依赖注入（DI）。

IOC 和 DI 、DL 的关系（这个 DL，Avalon 和 EJB 就是使用的这种方式实现的 IoC）：



DL 已经被抛弃，因为他需要用户自己去是使用 API 进行查找资源和组装对象。即有侵入性。

DI 是 Spring 使用的方式，容器负责组件的装配。

注意：Java 使用 DI 方式实现 IoC 的不止 Spring，包括 Google 的 Guice，还有一个冷门的 PicoContainer（极度轻量，但只提供 IoC）。

## Spring 的 IoC

Spring 的 IoC 设计支持以下功能：

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

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

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

那么， Spring 如何设计容器的呢？

Spring 作者 Rod Johnson 设计了两个接口用以表示容器。

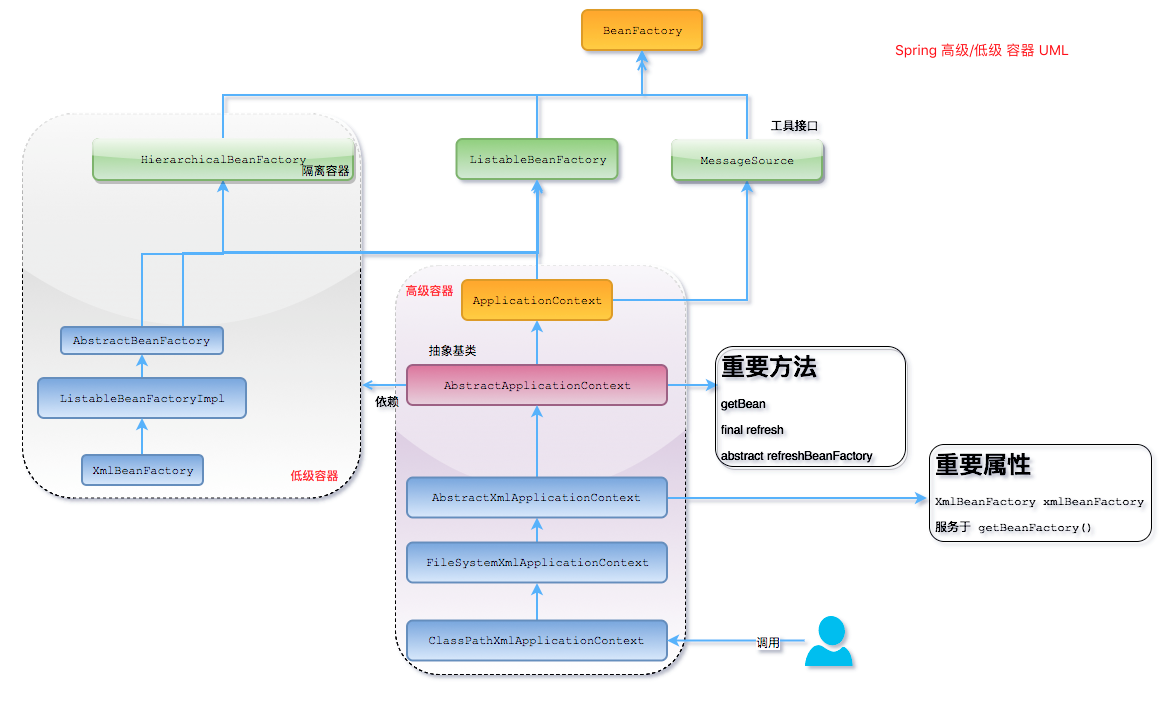
* BeanFactory
* ApplicationContext

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

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

当然，除了这两个大接口，还有其他的辅助接口，但我今天不会花太多篇幅介绍他们。

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



有点复杂？ 先不要慌，我来解释一下。

最上面的 BeanFactory 知道吧？我就不讲了。

下面的 3 个绿色的，都是功能扩展接口，这里就不展开讲。

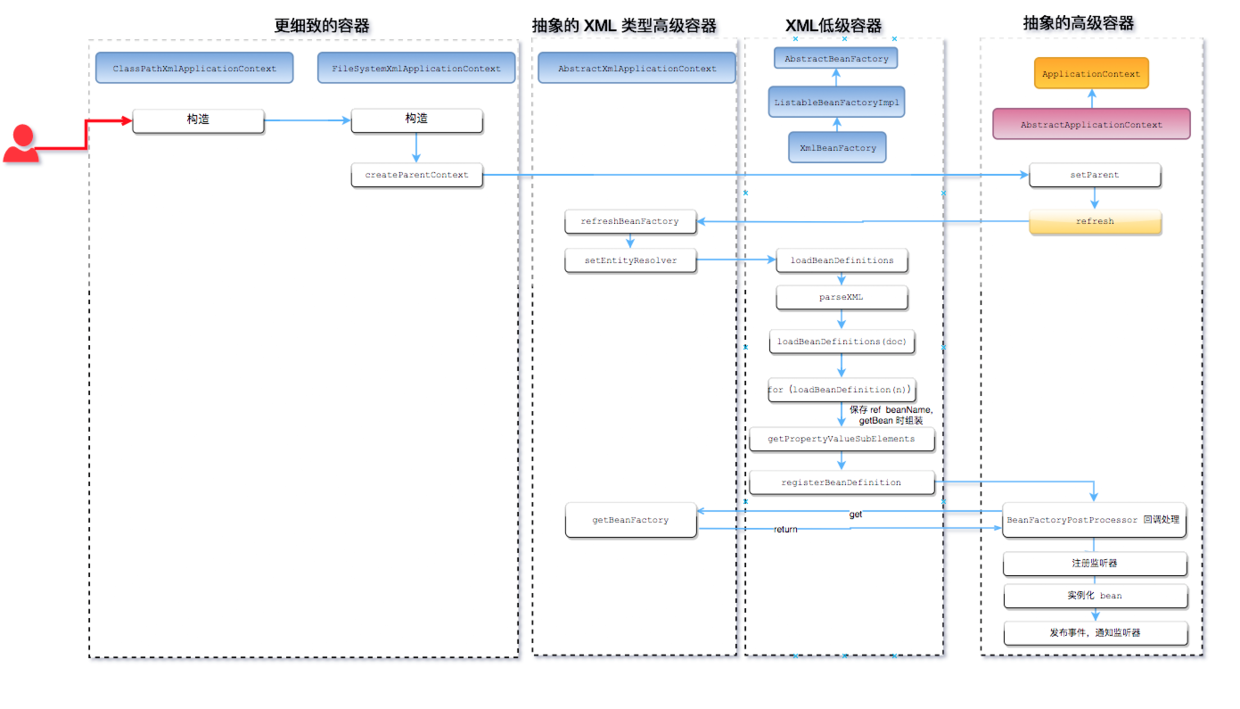
看下面的隶属 ApplicationContext 粉红色的 “高级容器”，依赖着 “低级容器”，这里说的是依赖，不是继承哦。他依赖着 “低级容器” 的 getBean 功能。而高级容器有更多的功能：支持不同的信息源头，可以访问文件资源，支持应用事件（Observer 模式）。

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

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

好，解释了低级容器和高级容器，我们可以看看一个 IoC 启动过程是什么样子的。说白了，就是 ClassPathXmlApplicationContext 这个类，在启动时，都做了啥。（由于我这是 interface21 的代码，肯定和你的 Spring 4.x 系列不同）。

下图是 ClassPathXmlApplicationContext 的构造过程，实际就是 Spring IoC 的初始化过程。



注意，这里为了理解方便，有所简化。

这里再用文字来描述这个过程：

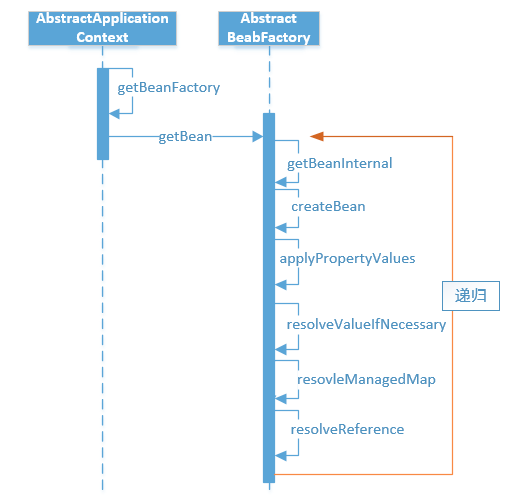
1. 用户构造 ClassPathXmlApplicationContext（简称 CPAC）
2. CPAC 首先访问了 “抽象高级容器” 的 final 的 refresh 方法，这个方法是模板方法。所以要回调子类（低级容器）的 refreshBeanFactory 方法，这个方法的作用是使用低级容器加载所有 BeanDefinition 和 Properties 到容器中。
3. 低级容器加载成功后，高级容器开始处理一些回调，例如 Bean 后置处理器。回调 setBeanFactory 方法。或者注册监听器等，发布事件，实例化单例 Bean 等等功能，这些功能，随着 Spring 的不断升级，功能越来越多，很多人在这里迷失了方向 ：）。

简单说就是：

1. 低级容器 加载配置文件（从 XML，数据库，Applet），并解析成 BeanDefinition 到低级容器中。
2. 加载成功后，高级容器启动高级功能，例如接口回调，监听器，自动实例化单例，发布事件等等功能。

所以，一定要把 “低级容器” 和“高级容器” 的区别弄清楚。不能一叶障目不见泰山。

好，当我们创建好容器，就会使用 getBean 方法，获取 Bean，而 getBean 的流程如下：



从图中可以看出，getBean 的操作都是在低级容器里操作的。其中有个递归操作，这个是什么意思呢？

假设 ： 当 Bean\_A 依赖着 Bean\_B，而这个 Bean\_A 在加载的时候，其配置的 ref = “Bean\_B” 在解析的时候只是一个占位符，被放入了 Bean\_A 的属性集合中，当调用 getBean 时，需要真正 Bean\_B 注入到 Bean\_A 内部时，就需要从容器中获取这个 Bean\_B，因此产生了递归。

为什么不是在加载的时候，就直接注入呢？因为加载的顺序不同，很可能 Bean\_A 依赖的 Bean\_B 还没有加载好，也就无法从容器中获取，你不能要求用户把 Bean 的加载顺序排列好，这是不人道的。

所以，Spring 将其分为了 2 个步骤：

1. 加载所有的 Bean 配置成 BeanDefinition 到容器中，如果 Bean 有依赖关系，则使用占位符暂时代替。
2. 然后，在调用 getBean 的时候，进行真正的依赖注入，即如果碰到了属性是 ref 的（占位符），那么就从容器里获取这个 Bean，然后注入到实例中 —— 称之为依赖注入。

可以看到，依赖注入实际上，只需要 “低级容器” 就可以实现。

这就是 IoC。

所以 ApplicationContext refresh 方法里面的操作不只是 IoC，是高级容器的所有功能（包括 IoC），IoC 的功能在低级容器里就可以实现。

## ICO初始化过程

IoC 在 Spring 里，只需要低级容器就可以实现，2 个步骤：

a. 加载配置文件，解析成 BeanDefinition 放在 Map 里。

b. 调用 getBean 的时候，从 BeanDefinition 所属的 Map 里，拿出 Class 对象进行实例化，同时，如果有依赖关系，将递归调用 getBean 方法 —— 完成依赖注入。

上面就是 Spring 低级容器（BeanFactory）的 IoC。

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

可以预见，随着 Spring 的不断发展，高级容器的功能会越来越多。

诚然，了解 IoC 的过程，实际上为了了解 Spring 初始化时，各个接口的回调时机。例如 InitializingBean，BeanFactoryAware，ApplicationListener 等等接口，这些接口的作用，笔者之前写过一篇文章进行介绍，有兴趣可以看一下，关键字：Spring 必知必会 扩展接口。

但是请注意，实现 Spring 接口代表着你这个应用就绑定死 Spring 了！代表 Spring 具有侵入性！要知道，Spring 发布时，无侵入性就是他最大的宣传点之一 —— 即 IoC 容器可以随便更换，代码无需变动。而现如今，Spring 已然成为 J2EE 社区准官方解决方案，也没有了所谓的侵入性这个说法。因为他就是标准，和 Servlet 一样，你能不实现 Servlet 的接口吗？: -)

## Spring中的那些扩展机制

<https://mp.weixin.qq.com/s/zv9K5lVGTUpsWb4_fuXC-g>

<https://juejin.im/entry/5ba45b0f6fb9a05d0a052a34>

基本思路有两个： 扩展容器初始化 或者 扩展bean创建

* 扩展容器初始化
* 扩展BeanDefinition: 比如，dubbo，通过扩展xml schema，实现namespaceHandler和BeanDefinition的相关接口，实现扩展的
* 扩展BeanFactoryPostProcesser:只要是跟配置的内容相关的，可以修改beanDefinition中的内容。Nacos的配置中间就是扩展了这个注入值的。
* 扩展bean创建（生命周期）

可以扩展的点比较多。比如通过beanName找到需要创建的对象的beanDefinition。之后逐步实现这个对象的类的，beanFactory、applicationAware、beanPostProcesser#前置方法、afterPropertiesSet()、initMethod、beanPostProcesser#后置方法。

之后正常使用

容器关闭或重启的时候，会调用DispossableBean接口

例子：dubbo的 serviceBean 实现了很多接口。

## BeanPostProcessor应用

在spring IOC容器初始化的过程中，其实是看不到bean的动态代理相关的内容。这是因为动态代理只是spring 扩展出来的一种应用。常见的方式就是使用beanPostProcessor进行代理对象的生成。

BeanPostProcessor的使用：<https://www.cnblogs.com/jyyzzjl/p/5417418.html>

## 【美团】spring 对象的生命周期？ spring 启动过程？怎么加载某个类的实现类的？

<https://juejin.im/entry/5e47c7a3e51d4527151553f5>

<https://www.jianshu.com/nb/29052266>

spring 启动过程：

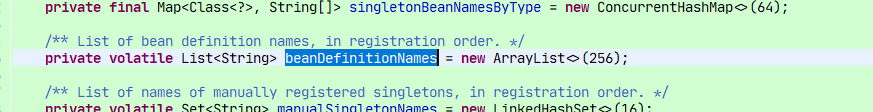
* 通过xml的配置，将xml配置转换为BeanDefinition对象。主要是一些
  + 一个bean的xml配置，我们将配置转为一个BeanDefinition对象
  + 开启注解，并扫描指定的包路径的话，每发现一个被注解修饰的类就创建一BeanDefinition个，比如注解service、component、controller、reponsitory、bean等
* 调用AbstractApplicationContext的refresh方法。

大致流程是： <https://blog.csdn.net/boling_cavalry/article/details/81045637>

* 主要是解析配置文件，比如properties。注意：BeanDefinition对象中保存的数据最原始的，保留${URL}的，只有getBean时解析。
* 准备beanfactory相关的类加载器，监听器等。
* 开始创建单例的bean

其实spring bean的生命周期粗略分为：创建bean、getBean、销毁bean。这是singleton的

创建bean的时候，beanDefinitionNames是不需要什么特别排序的，除非设置了order



每次实例化bean的时候，都是按照下面链接中的顺序调用方法

<https://www.cnblogs.com/caoxb/p/12435197.html#top>

当一个接口有多个实现类的时候，spring注入的顺序是这样的https://www.cnblogs.com/xrq730/p/6685528.html

* 判断注解，是否指定了resource或Qualefer,这样就通过byName去找
* 没有上面两个注解的话，先byType，遍历所有的所有的spring负责管理的bean的class，找到实现类。正常是一个
  + 如果实现类有两个，则使用字段的名称去byName。
  + 匹配不上的话，则判断实现类是否使用了主键/优先级的注解修饰，注入主要或优先级高的
  + 都不行的话就报错了。Spring启动失败

# AOP

Aop原理与动态代理：<https://mp.weixin.qq.com/s/PX7A3pUqQHhxkO3zJBWSJg>

## 什么是AOP？Spring AOP和AspectJ的区别是什么

<https://zhuanlan.zhihu.com/p/113256693>

## Spring AOP中何时使用JDK动态代理，何时使用CGLIB动态代理？

首先，并不是全部的spring bean都使用aop。常见的场景有权限管理、缓存、日志、全局异常、事务、监控普米尔修斯等

如果真需要进行动态代理的话：

1. 目标对象实现了接口的话，默认使用JDK的动态代理

2. 目标对象不是接口，而是pojo的话，只能使用CGLib

3. 如果，目标是配置的需要aop代理的类，强制使用 CGLib

<https://www.cnblogs.com/caoxb/p/12406005.html>

实际中，高版本的JDK 动态代理的性能是高于CGLib

在aop方面，关于性能，我们其实可以得出一个排序

AspectJ > JDK > CGLib

AspectJ的缺点是编译时直接改写class文件，没有扩展性

CGLib 是扩展性最好

## 【快手】【FGA】Aop哪些常见会失效

<https://blog.csdn.net/blueheart20/article/details/97258549>

* 对象内部方法间调用
* 使用spring aop，方法必须是public，不能是final和static的
* 使用aspectJ aop,方法不能是private，不能是final和static
* 方法中开启新的线程，因为事务是基于ThreadLocal实现的，所以这个线程无法继承父线程的事务。

# 设计模式

## 简单工厂模式

又叫做静态工厂方法（StaticFactory Method）模式，但不属于23种GOF设计模式之一。

简单工厂模式的实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。如下配置，就是在 HelloItxxz 类中创建一个 itxxzBean。

<beans>

<bean id="singletonBean" class="com.itxxz.HelloItxxz">

<constructor-arg>

<value>Hello! 这是singletonBean!value>

</constructor-arg>

</ bean>

<bean id="itxxzBean" class="com.itxxz.HelloItxxz" singleton="false">

<constructor-arg>

<value>Hello! 这是itxxzBean! value>

</constructor-arg>

</bean>

</beans>

## 工厂方法模式

通常由应用程序直接使用new创建新的对象，为了将对象的创建和使用相分离，采用工厂模式,即应用程序将对象的创建及初始化职责交给工厂对象。

一般情况下,应用程序有自己的工厂对象来创建bean.如果将应用程序自己的工厂对象交给Spring管理,那么Spring管理的就不是普通的bean,而是工厂Bean。

## 单例模式

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

spring中的单例模式完成了后半句话，即提供了全局的访问点BeanFactory。但没有从构造器级别去控制单例，这是因为spring管理的是是任意的java对象。

核心提示点：Spring下默认的bean均为singleton，可以通过singleton=“true|false” 或者 scope="?"来指定。

## 代理模式

在Spring的Aop中，使用的Advice（通知）来增强被代理类的功能。Spring实现这一AOP功能的原理就使用代理模式（1、JDK动态代理。2、CGLib字节码生成技术代理。）对类进行方法级别的切面增强，即，生成被代理类的代理类， 并在代理类的方法前，设置拦截器，通过执行拦截器重的内容增强了代理方法的功能，实现的面向切面编程

## 包装器模式

在我们的项目中遇到这样一个问题：我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。我们以往在spring和hibernate框架中总是配置一个数据源，因而sessionFactory的dataSource属性总是指向这个数据源并且恒定不变，所有DAO在使用sessionFactory的时候都是通过这个数据源访问数据库。

但是现在，由于项目的需要，我们的DAO在访问sessionFactory的时候都不得不在多个数据源中不断切换，问题就出现了：如何让sessionFactory在执行数据持久化的时候，根据客户的需求能够动态切换不同的数据源？我们能不能在spring的框架下通过少量修改得到解决？是否有什么设计模式可以利用呢？

首先想到在spring的applicationContext中配置所有的dataSource。这些dataSource可能是各种不同类型的，比如不同的数据库：Oracle、SQL Server、MySQL等，也可能是不同的数据源：比如apache 提供的org.apache.commons.dbcp.BasicDataSource、spring提供的org.springframework.jndi.JndiObjectFactoryBean等。然后sessionFactory根据客户的每次请求，将dataSource属性设置成不同的数据源，以到达切换数据源的目的。

spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。基本上都是动态地给一个对象添加一些额外的职责。

## 观察者模式

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

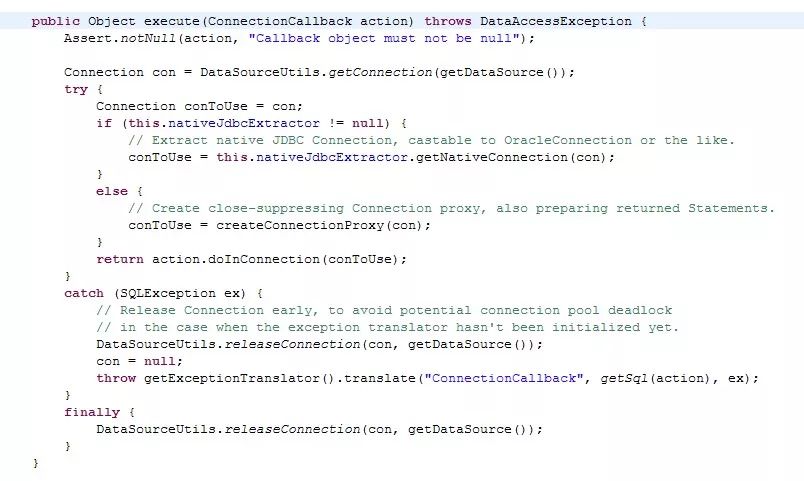
spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

## 策略模式

定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

spring中在实例化对象的时候用到Strategy模式

在SimpleInstantiationStrategy中有如下代码说明了策略模式的使用情况：



## 模板方法模式

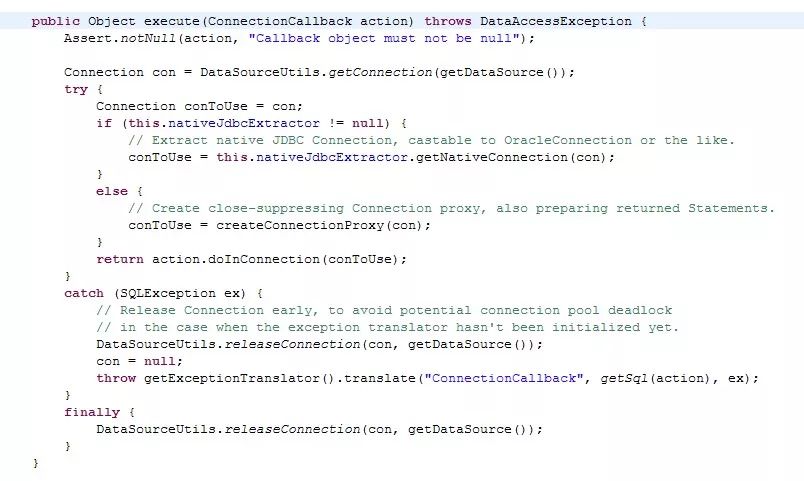
定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

Template Method模式一般是需要继承的。这里想要探讨另一种对Template Method的理解。spring中的JdbcTemplate，在用这个类时并不想去继承这个类，因为这个类的方法太多，但是我们还是想用到JdbcTemplate已有的稳定的、公用的数据库连接，那么我们怎么办呢？我们可以把变化的东西抽出来作为一个参数传入JdbcTemplate的方法中。但是变化的东西是一段代码，而且这段代码会用到JdbcTemplate中的变量。怎么办？那我们就用回调对象吧。

在这个回调对象中定义一个操纵JdbcTemplate中变量的方法，我们去实现这个方法，就把变化的东西集中到这里了。然后我们再传入这个回调对象到JdbcTemplate，从而完成了调用。这可能是Template Method不需要继承的另一种实现方式。

以下是一个具体的例子：

JdbcTemplate中的execute方法



# 线程安全问题

singleton 是线程不安全的，对象有spring管理

prototype 每次从SpringApplicationContent中getBean的时候都生产一个新的对象，spring负责实例化后就不由spring管理他的生命周期了。

注意：

* 这就出现一个问题，比如controller类是prototype ，我们通过URL访问的时候，spring mvc通过URL获得controller的class，之后getBean获得controller的实例。这就造成我们访问的时候每次都是不同的controller的实例。这样controller的实例是线程安全的。
* 但是，如果service类是prototype，在controller初始化的时候（controller是singleton 的），不同的controller中的service是不同的。因为controller只初始化一次，getBean一次。索引一个controller中的service是不变的，也就是线程不安全的。

总结：使用prototype 能够实现线程安全，但是使用的时候往往会忽视掉上面的第二点问题。另外，我们真想要使用prototype 实现线程安全的话，需要我们把controller、service、dao，由外往内的逐层设置为prototype，虽然保证了线程安全，但是每次访问都生产很多对象出来，还不由spring管理对象的生命周期，造成资源消耗。这也是为什么spring默认为单例的了。

@RestController

@Slf4j

@Scope(ConfigurableBeanFactory.SCOPE\_SINGLETON) // 默认值是 singleton，单例的，整个SpringContent只有一个

//@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE) // prototype 表示每次调用该接口都会生成一个新的Bean, spring 会重新实例化DemoController，之后生命周期就不管spring 去管了，反正谁用的时候都是创建新的

//@Scope(WebApplicationContext.SCOPE\_SESSION) // session 每个session是一个对象

//@Scope(WebApplicationContext.SCOPE\_REQUEST) // request 每个request是一个对象

public class DemoController {

private String a = "0";

@RequestMapping("/get")

public String get(String b) {

log.info("当前值={}", a);

a = b;

log.info("执行后={}", a);

log.info("当前对象={}", this);

log.info("service1={}", service1);

log.info("service2={}", service2);

return a;

}

}

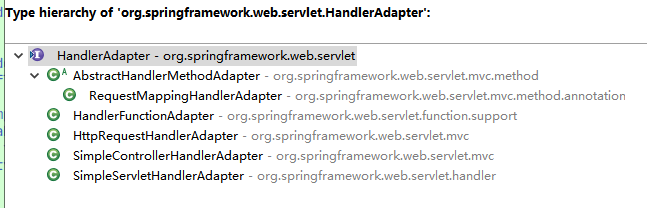
# Spring mvc

## 【快手】Spring MVC 一个请求的基本处理过程是什么？

springMVC请求流程详解： <https://www.cnblogs.com/leskang/p/6101368.html>

<https://blog.csdn.net/u010859650/article/details/81167223>

调用过程中，比较重要的是HandlerAdapter，根据controller的不同的实现方式，会适配到不同的handler处理方式。因为不同的实现方式，HandlerExecutionChain中保存实际需要调用的对象，也就是handler字段，会被封装成不用的对象。注解方式都是HandlerMethod。



## Spring mvc如何统计URL调用耗时？

* 拦截器，使用request属性或者threadLocal都可以
* 过滤器，很简单的实现，但是调用时间为整个spring MVC的处理时间
* request的监听器很简单的实现，但是调用时间为整个spring MVC和tomcat的处理时间
* 重新实现DispatcherServlet的方法，想监控哪里就监控哪里
* 使用动态代理,一般使用aspect，进行编译期的字节码增强，防止对spring aop产生不良影响

## Spring MVC项目中暴露请求的方式

* 继承Servlet
* 实现Controller接口
* 实现httpRequestHandler接口
* 使用@RequestMapping注解

<https://cloud.tencent.com/developer/article/1497623>