框架篇（spring）

1. BeanFactory 和 ApplicationContext 有什么区别

一、BeanFactory 和 ApplicationContext

Bean 工厂（com.springframework.beans.factory.BeanFactory）是 Spring 框架最核心的接口，它提供了高级 IoC 的配置机制。

应用上下文（com.springframework.context.ApplicationContext）建立在 BeanFactory 基础之上。几乎所有的应用场合我们都直接使用 ApplicationContext 而非底层的 BeanFactory。

1.1 BeanFactory 的类体系结构

BeanFactory 接口位于类结构树的顶端， 它最主要的方法就是 getBean(StringbeanName)，该方法从容器中返回特定名称的 Bean，BeanFactory 的功能通过其他的接口得到不断扩展。

ListableBeanFactory：该接口定义了访问容器中 Bean 基本信息的若干方法，如查看 Bean 的个数、获取某一类型 Bean 的配置名、查看容器中是否包括某一 Bean 等方法；

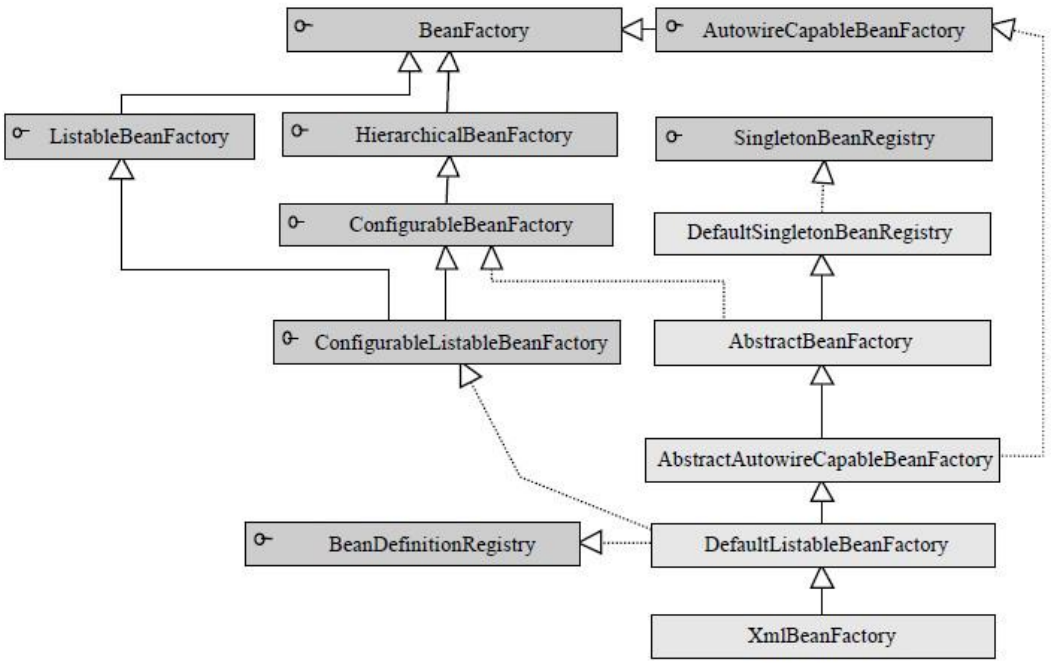
HierarchicalBeanFactory：父子级联 IoC 容器的接口，子容器可以通过接口方法访问父容器；

ConfigurableBeanFactory：是一个重要的接口，增强了 IoC 容器的可定制性，它定义了设置类装载器、属性编辑器、容器初始化后置处理器等方法；

AutowireCapableBeanFactory：定义了将容器中的 Bean 按某种规则（如按名字匹配、按类型匹配等）进行自动装配的方法；

SingletonBeanRegistry：定义了允许在运行期间向容器注册单实例 Bean 的方法；

BeanDefinitionRegistry：Spring 配置文件中每一个<bean>节点元素在 Spring 容器里都通过一个 BeanDefinition 对象表示，它描述了 Bean 的配置信息。而 BeanDefinitionRegistry 接口提供了向容器手工注册 BeanDefinition 对象的方法。



1.2 ApplicationContext 的类体系结构

ApplicationContext 由 BeanFactory 派生而来，提供了更多面向实际应用的功能。在 BeanFactory 中，很多功能需要以编程的方式实现，而在 ApplicationContext 中则可以通过配置的方式实现。

ApplicationContext 的主要实现类是 ClassPathXmlApplicationContext 和 FileSystemXmlApplicationContext，前者默认从类路径加载配置文件，后者默认从文件系统中装载配置文件。

核心接口包括：

ApplicationEventPublisher：让容器拥有发布应用上下文事件的功能，包括容器启动事件、关闭事件等。实现了 ApplicationListener 事件监听接口的 Bean 可以接收到容器事件， 并对事件进行响应处理。在 ApplicationContext 抽象实现类 AbstractApplicationContext 中，我们可以发现存在一个 ApplicationEventMulticaster，它负责保存所有监听器，以便在容器产生上下文事件时通知这些事件监听 者。

MessageSource：为应用提供 i18n 国际化消息访问的功能；

ResourcePatternResolver ： 所有 ApplicationContext 实现类都实现了类似于 PathMatchingResourcePatternResolver 的功能，可以通过带前缀的 Ant 风格的资源文件路径装载 Spring 的配置文件。

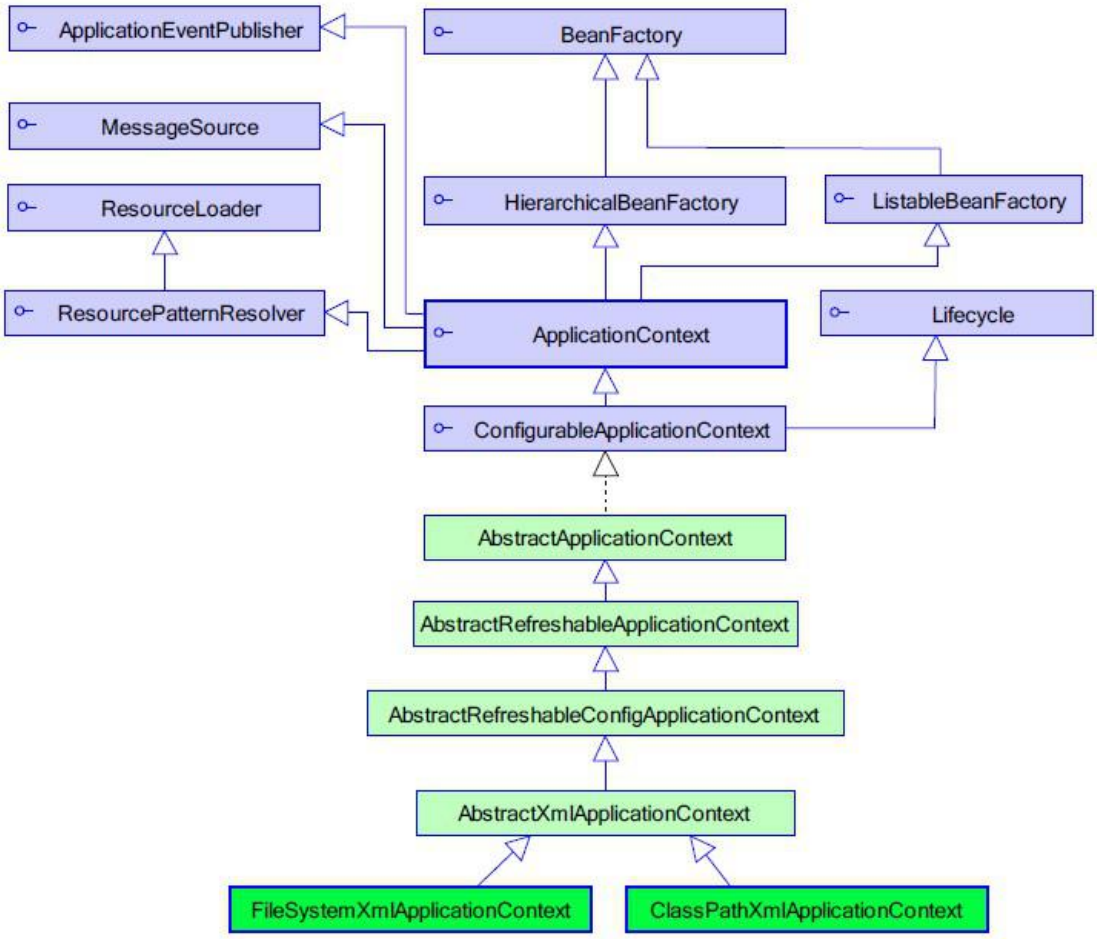
LifeCycle：该接口是 Spring 2.0 加入的，该接口提供了 start()和 stop()两个方法，主要用于控制异步处理过程。在具体使用时，该接口同时被 ApplicationContext 实现及具体 Bean 实现，ApplicationContext 会将 start/stop 的信息传递给容器中所有实现了该接口的 Bean，以达到管理和控制 JMX、任务调度等目的。

ConfigurableApplicationContext 扩展于 ApplicationContext，它新增加了两个主要的方法：refresh()和 close()，让 ApplicationContext 具有启动、刷新和关闭应用上下文的能力。在应用上下文关闭的情况下调用 refresh()即可启动应用上下文，在已经启动的状态下，调用 refresh()则清除缓存并重新装载配置信息，而调用 close()则可关闭应用上下文。这些接口方法为容器的控制管理带来了便利。

代码示例：

```
ApplicationContext ctx =new
ClassPathXmlApplicationContext("com/baobaotao/context/beans.xml");
ApplicationContext ctx =new
FileSystemXmlApplicationContext("com/baobaotao/context/beans.xml");
ApplicationContext ctx = new ClassPathXmlApplicationContext(new
String[]{"conf/beans1.xml","conf/beans2.xml"});
```

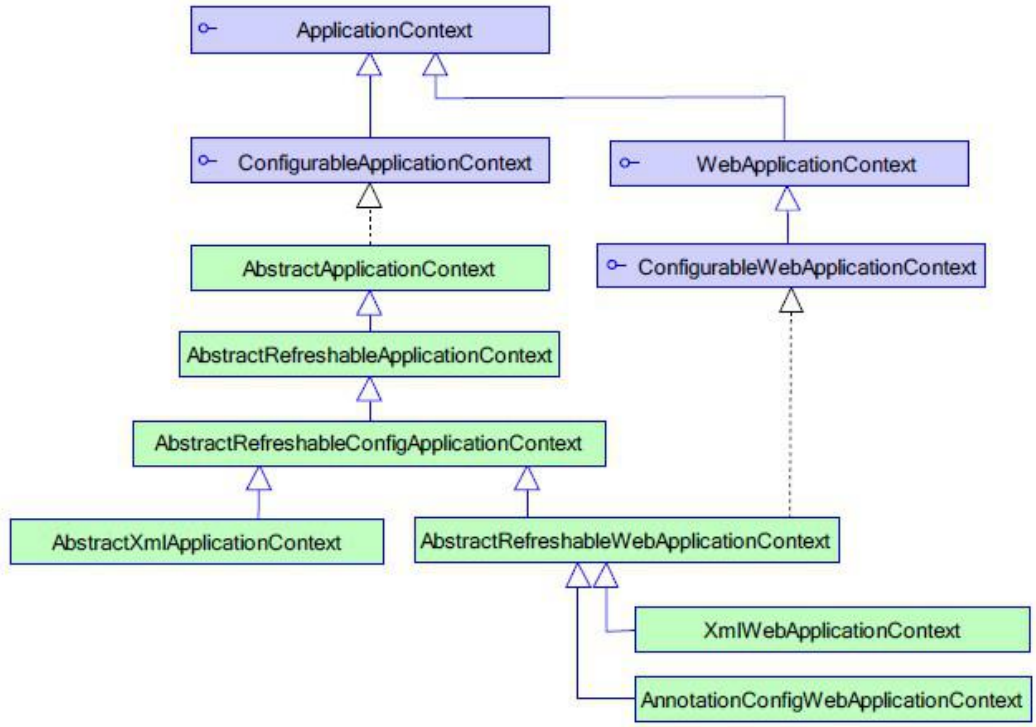
ApplicationContext 的初始化和 BeanFactory 有一个重大的区别：BeanFactory 在初始化容器时，并未实例化 Bean，直到第一次访问某个 Bean 时才实例目标 Bean；而 ApplicationContext 则在初始化应用上下文时就实例化所有单实例的 Bean 。



二、WebApplicationContext 类体系结构

WebApplicationContext 是专门为 Web 应用准备的，它允许从相对于 Web 根目录的路径中装载配置文件完成初始化工作。从 WebApplicationContext 中可以获得 ServletContext 的引用，整个 Web 应用上下文对象将作为属性放置到 ServletContext 中，以便 Web 应用环境可以访问 Spring 应用上下文（ApplicationContext）。Spring 专门为此提供一个工具类 WebApplicationContextUtils，通过该类的 getWebApplicationContext(ServletContext sc)方法，即可以从 ServletContext 中获取 WebApplicationContext 实例。

Spring 2.0 在 WebApplicationContext 中还为 Bean 添加了三个新的作用域：request 作用域、session 作用域和 global session 作用域。而在非 Web 应用的环境下，Bean 只有 singleton 和 prototype 两种作用域。



由于 Web 应用比一般的应用拥有更多的特性，因此 WebApplicationContext 扩展了 ApplicationContext 。 WebApplicationContext 定义了一个常量 ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE，在上下文启动 时，WebApplicationContext 实例即以此为键放置在 ServletContext 的属性列表中，因此我们可以直接通过以下语句从 Web 容器中获取

WebApplicationContext：

WebApplicationContext wac = (WebApplicationContext)servletContext.getAttribute(

WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);

2.1 WebApplicationContext 的初启动方式 和 BeanFactory、ApplicationContext 有所区别。

因为 WebApplicationContext 需要 ServletContext 实例，也就是说它必须在拥有 Web 容器的前提下才能完成启动的工作。有过 Web 开发经验的读者都知道可以在 web.xml 中配置自启

动的 Servlet 或定义 Web 容器监听器（ServletContextListener），借助这两者中的任何一个，我们就可以完成启动 Spring Web 应用上下文的工作。

Spring 分别提供了用于启动 WebApplicationContext 的 Servlet 和 Web 容器监听器：

org.springframework.web.context.ContextLoaderServlet；

org.springframework.web.context.ContextLoaderListener。

两者的内部都实现了启动 WebApplicationContext 实例的逻辑，我们只要根据 Web 容器的具体情况选择两者之一，并在 web.xml 中完成配置就可以了。

三、WebApplicationContext 需要使用日志功能

用户可以将 Log4J 的配置文件放置到类路径 WEB-INF/classes 下，这时 Log4J 引擎即可顺利启动。

如果 Log4J 配置文件放置在其他位置，用户还必须在 web.xml 指定 Log4J 配置文件位置。

Spring 为启用 Log4J 引擎提供了两个类似于启动 WebApplicationContext 的实现类：Log4jConfigServlet 和 Log4jConfigListener，

不管采用哪种方式都必须保证能够在装载 Spring 配置文件前先装载 Log4J 配置信息。

四、父子容器

通过 HierarchicalBeanFactory 接口，Spring 的 IoC 容器可以建立父子层级关联的容器体系，子容器可以访问父容器中的 Bean，但父容器不能访问子容器的 Bean。在容器内，Bean 的 id 必须是唯一的，但子容器可以拥有一个和父容器 id 相同的 Bean。父子容器层级体系增强了 Spring 容器架构的扩展性和灵活性，因为第三方可以通过编程的方式，为一个已经存在的容器添加一个或多个特殊用途的子容器，以提供一些额外的功能。

Spring 使用父子容器实现了很多功能，比如在 Spring MVC 中，展现层 Bean 位于一个子容器中，而业务层和持久层的 Bean 位于父容器中。这样，展现层 Bean 就可以引用业务层和持久层的 Bean，而业务层和持久层的 Bean 则看不到展现层的 Bean。

BeanFactory：

是 Spring 里面最低层的接口，提供了最简单的容器的功能，只提供了实例化对象和拿对象的功能；

ApplicationContext：

应用上下文，继承 BeanFactory 接口，它是 Spring 的一各更高级的容器，提供了更多的有用的功能；

1) 国际化（MessageSource）

2) 访问资源，如 URL 和文件（ResourceLoader）

3) 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的 web 层

4) 消息发送、响应机制（ApplicationEventPublisher）

5) AOP（拦截器）

两者装载 bean 的区别

BeanFactory：

BeanFactory 在启动的时候不会去实例化 Bean，中有从容器中拿 Bean 的时候才会去实例化；

ApplicationContext：

ApplicationContext 在启动的时候就把所有的 Bean 全部实例化了。它还可以为 Bean 配置 lazy-init=true 来让 Bean 延迟实例化；

我们该用 BeanFactory 还是 ApplicationContent

延迟实例化的优点：（BeanFactory）

应用启动的时候占用资源很少；对资源要求较高的应用，比较有优势；

不延迟实例化的优点： （ApplicationContext）

1. 所有的 Bean 在启动的时候都加载，系统运行的速度快；

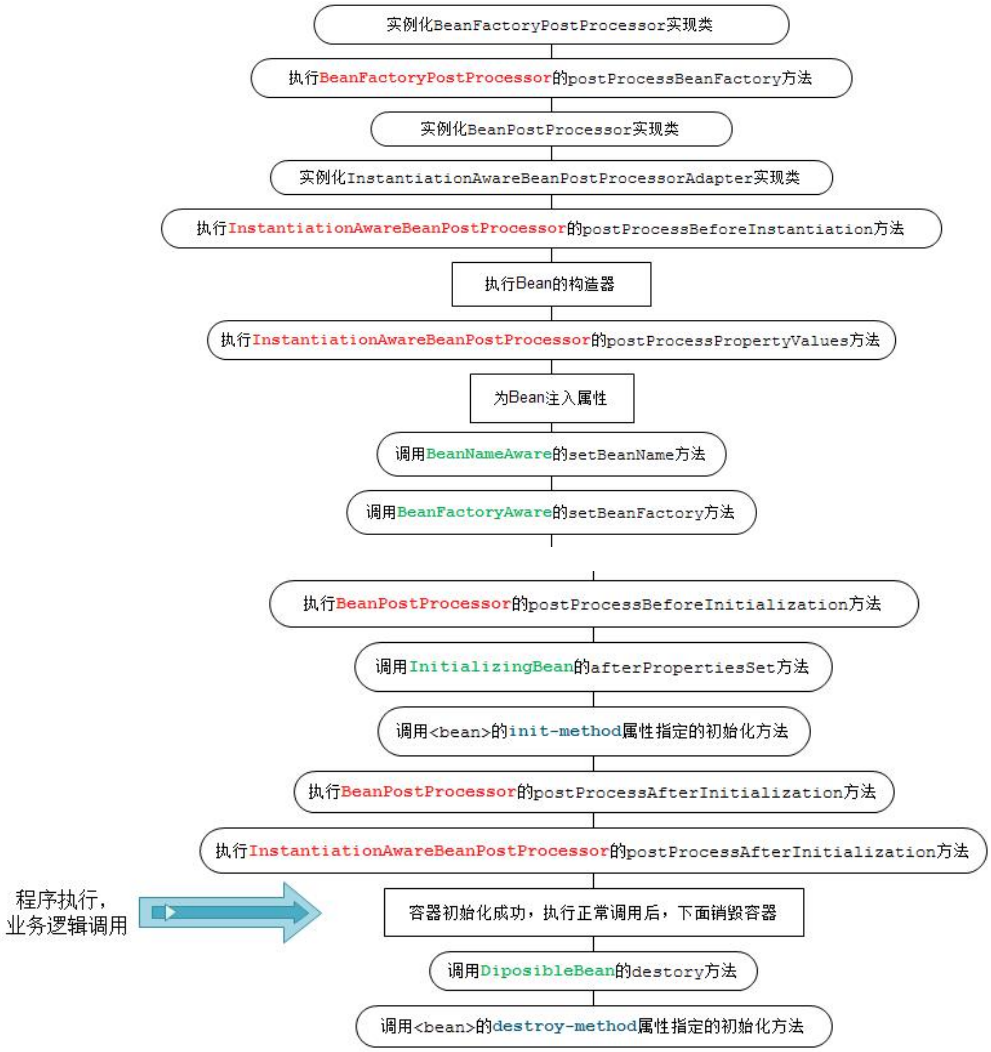
2. 在启动的时候所有的 Bean 都加载了，我们就能在系统启动的时候，尽早的发现系统中的配置问题

3. 建议 web 应用，在启动的时候就把所有的 Bean 都加载了。（把费时的操作放到系统启动中完成）

2..Spring Bean 的生命周期

一、生命周期流程图：

Spring Bean 的完整生命周期从创建 Spring 容器开始，直到最终 Spring 容器销毁 Bean，这其中包含了一系列关键点。



容器注册了以上各种接口，程序那么将会按照以上的流程进行。下面将仔细讲解各接口作用。

二、各种接口方法分类

Bean 的完整生命周期经历了各种方法调用，这些方法可以划分为以下几类：

1、Bean 自身的方法 ： 这个包括了 Bean 本身调用的方法和通过配置文件中<bean>的 init-method 和 destroy-method 指定的方法

2、Bean 级生命周期接口方法 ： 这个包括了 BeanNameAware、BeanFactoryAware、InitializingBean 和 DiposableBean 这些接口的方法

3、容器级生命周期接口方法 ： 这个包括了 InstantiationAwareBeanPostProcessor 和 BeanPostProcessor 这两个接口实现，一般称它们的实现类为“后处理器”。

4、工厂后处理器接口方法 ： 这个包括了 AspectJWeavingEnabler, ConfigurationClassPostProcessor, CustomAutowireConfigurer 等等非常有用的工厂后处理器接口的方法。工厂后处理器也是容器级的。在应用上下文装配配置文件之后立即调用。

1：Bean 的建立：

容器寻找 Bean 的定义信息并将其实例化。

2：属性注入：

使用依赖注入，Spring 按照 Bean 定义信息配置 Bean 所有属性

3：BeanNameAware 的 setBeanName()：

如果 Bean 类有实现 org.springframework.beans.BeanNameAware 接口，工厂调用 Bean 的 setBeanName()方法传递 Bean 的 ID。

4：BeanFactoryAware 的 setBeanFactory()：

如果 Bean 类有实现 org.springframework.beans.factory.BeanFactoryAware 接口，工厂调用 setBeanFactory()方法传入工厂自身。

5：BeanPostProcessors 的 ProcessBeforeInitialization()

如果有 org.springframework.beans.factory.config.BeanPostProcessors 和 Bean 关联，那么其 postProcessBeforeInitialization()方法将被将被调用。

6：initializingBean 的 afterPropertiesSet()：

如果 Bean 类已实现 org.springframework.beans.factory.InitializingBean 接口，则执行他的 afterProPertiesSet()方法

7：Bean 定义文件中定义 init-method：

可以在 Bean 定义文件中使用"init-method"属性设定方法名称例如：

如果有以上设置的话，则执行到这个阶段，就会执行 initBean()方法

8：BeanPostProcessors 的 ProcessaAfterInitialization()

如果有任何的 BeanPostProcessors 实例与 Bean 实例关联，则执行 BeanPostProcessors 实例的 ProcessaAfterInitialization()方法

此时，Bean 已经可以被应用系统使用，并且将保留在 BeanFactory 中知道它不在被使用。有两种方法可以将其从 BeanFactory 中删除掉(如图 1.2):

1：DisposableBean 的 destroy()

在容器关闭时，如果 Bean 类有实现 org.springframework.beans.factory.DisposableBean 接口，则执行他的 destroy()方法

2：Bean 定义文件中定义 destroy-method

在容器关闭时，可以在 Bean 定义文件中使用"destroy-method"属性设定方法名称，例如：

如果有以上设定的话，则进行至这个阶段时，就会执行 destroy()方法，如果是使用 ApplicationContext 来生成并管理 Bean 的话则稍有不同，使用 ApplicationContext 来生成及管理 Bean 实例的话，在执行 BeanFactoryAware 的 setBeanFactory()阶段后，若 Bean 类上有实现 org.springframework.context.ApplicationContextAware 接口，则执行其 setApplicationContext()方法，接着才执行 BeanPostProcessors 的 ProcessBeforeInitialization()及之后的流程。

在说明前可以思考一下 Servlet 的生命周期：实例化，初始 init，接收请求 service，销毁 destroy；

Spring 上下文中的 Bean 也类似，如下

1、实例化一个 Bean——也就是我们常说的 new；

2、按照 Spring 上下文对实例化的 Bean 进行配置——也就是 IOC 注入；

3、如果这个 Bean 已经实现了 BeanNameAware 接口，会调用它实现的 setBeanName(String)方法，此处传递的就是 Spring 配置文件中 Bean 的 id 值

4、如果这个 Bean 已经实现了 BeanFactoryAware 接口，会调用它实现的 setBeanFactory(setBeanFactory(BeanFactory)传递的是 Spring 工厂自身（可以用这个方式来获

取其它 Bean，只需在 Spring 配置文件中配置一个普通的 Bean 就可以）；

5、如果这个 Bean 已经实现了 ApplicationContextAware 接口，会调用 setApplicationContext(ApplicationContext)方法，传入 Spring 上下文（同样这个方式也可以实现步骤 4 的内容，但比 4 更好，因为 ApplicationContext 是 BeanFactory 的子接口，有更多的实现方法）；

6、如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用 postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor 经常被用作是 Bean 内容的更改，并且由于这个是在 Bean 初始化结束时调用那个的方法，也可以被应用于内存或缓存技术；

7、如果 Bean 在 Spring 配置文件中配置了 init-method 属性会自动调用其配置的初始化方法。

8、如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用 postProcessAfterInitialization(Object obj, String s)方法、；

注：以上工作完成以后就可以应用这个 Bean 了，那这个 Bean 是一个 Singleton 的，所以一般情况下我们调用同一个 id 的 Bean 会是在内容地址相同的实例，当然在 Spring 配置文件中也可以配置非 Singleton，这里我们不做赘述。

9、当 Bean 不再需要时，会经过清理阶段，如果 Bean 实现了 DisposableBean 这个接口，会调用那个其实现的 destroy()方法；

10、最后，如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性，会自动调用其配置的销毁方法。

以上 10 步骤可以作为面试或者笔试的模板，另外我们这里描述的是应用 Spring 上下文 Bean 的生命周期，如果应用 Spring 的工厂也就是 BeanFactory 的话去掉第 5 步就 Ok 了。

这 Spring 框架中，一旦把一个 bean 纳入到 Spring IoC 容器之中，这个 bean 的生命周期就会交由容器进行管理，一般担当管理者角色的是 BeanFactory 或 ApplicationContext。认识一下 Bean 的生命周期活动，对更好的利用它有很大的帮助。

下面以 BeanFactory 为例，说明一个 Bean 的生命周期活动：

Bean 的建立

由 BeanFactory 读取 Bean 定义文件，并生成各个实例。

Setter 注入: 执行 Bean 的属性依赖注入。

BeanNameAware 的 setBeanName()

如果 Bean 类实现了 org.springframework.beans.factory.BeanNameAware 接口，则执行其 setBeanName()方法。

BeanFactoryAware 的 setBeanFactory()

如果 Bean 类实现了 org.springframework.beans.factory.BeanFactoryAware 接口，则执行其 setBeanFactory()方法。

BeanPostProcessors 的 processBeforeInitialization()

容器中如果有实现 org.springframework.beans.factory.BeanPostProcessors 接口的实例，则任何 Bean 在初始化之前都会执行这个实例的 processBeforeInitialization()方法。

InitializingBean 的 afterPropertiesSet()

如果 Bean 类实现了 org.springframework.beans.factory.InitializingBean 接口，则执行其 afterPropertiesSet()方法。

Bean 定义文件中定义 init-method

在 Bean 定义文件中使用“init-method”属性设定方法名称，如下：

```
<bean id="demoBean" class="com.yangsq.bean.DemoBean" init-method="initMethod">
   .......
 </bean>
```

这时会执行 initMethod()方法，注意，这个方法是不带参数的。

BeanPostProcessors 的 processAfterInitialization()

容器中如果有实现 org.springframework.beans.factory.BeanPostProcessors 接口的实例，则任何 Bean 在初始化之前都会执行这个实例的 processAfterInitialization()方法。

DisposableBean 的 destroy()

在容器关闭时，如果 Bean 类实现了 org.springframework.beans.factory.DisposableBean 接口，则执行它的 destroy()方法。

Bean 定义文件中定义 destroy-method

在容器关闭时，可以在 Bean 定义文件中使用“destory-method”定义的方法

```
<bean id="demoBean" class="com.yangsq.bean.DemoBean" destory-method="destroyMethod">
   .......
</bean>
```

这时会执行 destroyMethod()方法，注意，这个方法是不带参数的。

如果使用 ApplicationContext 来维护一个 Bean 的生命周期，则基本上与上边的流程相同，只不过在执行 BeanNameAware 的 setBeanName()后，若有 Bean 类实现了 org.springframework.context.ApplicationContextAware 接口，则执行其 setApplicationContext()方法，然后再进行 BeanPostProcessors 的 processBeforeInitialization()

实际上，ApplicationContext 除了向 BeanFactory 那样维护容器外，还提供了更加丰富的框架功能，如 Bean 的消息，事件处理机制等。

3.Spring IOC 如何实现

Spring 中的 org.springframework.beans 包和 org.springframework.context 包构成了 Spring 框架 IoC 容器的基础。

BeanFactory 接口提供了一个先进的配置机制，使得任何类型的对象的配置成为可能。ApplicationContex 接口对 BeanFactory（是一个子接口）进行了扩展，在 BeanFactory 的基础上添加了其他功能，比如与 Spring 的 AOP 更容易集成，也提供了处理 message resource 的机制（用于国际化）、事件传播以及应用层的特别配置，比如针对 Web 应用的 WebApplicationContext。

org.springframework.beans.factory.BeanFactory 是 Spring IoC 容器的具体实现，用来包装和管理前面提到的各种 bean。BeanFactory 接口是 Spring IoC 容器的核心接口。

1、 IOC 容器就是具有依赖注入功能的容器，IOC 容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。应用程序无需直接在代码中 new 相关的对象，应用程序由 IOC 容器进行组装。在 Spring 中 BeanFactory 是 IOC 容器的实际代表者。

Spring IOC 容器如何知道哪些是它管理的对象呢？这就需要配置文件，Spring IOC 容器通过读取配置文件中的配置元数据，通过元数据对应用中的各个对象进行实例化及装配。一般使用基于 xml 配置文件进行配置元数据，而且 Spring 与配置文件完全解耦的，可以使用其他任何可能的方式进行配置元数据，比如注解、基于 java 文件的、基于属性文件的配置都可以。

那 Spring IOC 容器管理的对象叫什么呢？

2、 Bean 的概念

由 IOC 容器管理的那些组成你应用程序的对象我们就叫它 Bean， Bean 就是由 Spring 容器初始化、装配及管理的对象，除此之外，bean 就与应用程序中的其他对象没有什么区别了。那 IOC 怎样确定如何实例化 Bean、管理 Bean 之间的依赖关系以及管理 Bean 呢？这就需要配置元数据，在 Spring 中由 BeanDefinition 代表，后边会详细介绍，配置元数据指定如何实例化 Bean、如何组装 Bean 等。概念知道的差不多了，让我们来做个简单的例子。

```
public interface HelloService {
    public void sayHello();
}

public class HelloServiceImpl implements HelloService{
    public void sayHello(){
        System.out.println("Hello World!");
    }
}

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <!-- id 表示组件的名字，class 表示组件类 -->
    <bean id="helloService" class="com.ljq.test.HelloServiceImpl" />
</beans>

public class HelloServiceTest {
    @Test
    public void testHelloWorld() {
        // 1、读取配置文件实例化一个 IOC 容器
        ApplicationContext context = new
ClassPathXmlApplicationContext("helloworld.xml");
        // 2、从容器中获取 Bean，注意此处完全“面向接口编程，而不是面向实现”
        HelloService helloService = context.getBean("helloService", HelloService.class);
        // 3、执行业务逻辑
        helloService.sayHello();
    }
}
```

在 Spring IOC 容器的代表就是 org.springframework.beans 包中的 BeanFactory 接口，BeanFactory 接口提供了 IOC 容器最基本功能；而 org.springframework.context 包下的

ApplicationContext 接口扩展了 BeanFactory，还提供了与 Spring AOP 集成、国际化处理、事件传播及提供不同层次的 context 实现 (如针对 web 应用的 WebApplicationContext)。简单说，BeanFactory 提供了 IOC 容器最基本功能，而 ApplicationContext 则增加了更多支持企业级功能支持。ApplicationContext 完全继承 BeanFactory，因而 BeanFactory 所具有的语义也适用于 ApplicationContext。

容器实现一览：

• XmlBeanFactory：BeanFactory 实现，提供基本的 IOC 容器功能，可以从 classpath 或文件系统等获取资源；

（1）File file = new File("fileSystemConfig.xml");

Resource resource = new FileSystemResource(file);

BeanFactory beanFactory = new XmlBeanFactory(resource);

（2）

Resource resource = new ClassPathResource("classpath.xml");

BeanFactory beanFactory = new XmlBeanFactory(resource);

ClassPathXmlApplicationContext：ApplicationContext 实现，从 classpath 获取配置文件；

BeanFactory beanFactory = new ClassPathXmlApplicationContext("classpath.xml");

FileSystemXmlApplicationContext：ApplicationContext 实现，从文件系统获取配置文件。

BeanFactory beanFactory = new FileSystemXmlApplicationContext("fileSystemConfig.xml");

4. 说说 Spring AOP

AOP 技术利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。使用“横切”技术，AOP 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。Aop 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

通知(Adivce)

通知有 5 种类型：

Before 在方法被调用之前调用

After 在方法完成后调用通知，无论方法是否执行成功

After-returning 在方法成功执行之后调用通知

After-throwing 在方法抛出异常后调用通知

Around 通知了好、包含了被通知的方法，在被通知的方法调用之前后调用之后执行自定义的行为

我们可能会问，那通知对应系统中的代码是一个方法、对象、类、还是接口什么的呢？我想说一点，其实都不是，你可以理解通知就是对应我们日常生活中所说的通知，比如‘某某人，你 2019 年 9 月 1 号来学校报个到’，通知更多地体现一种告诉我们（告诉系统何）何时执

行，规定一个时间，在系统运行中的某个时间点（比如抛异常啦！方法执行前啦！），并非对应代码中的方法！并非对应代码中的方法！并非对应代码中的方法！

切点（Pointcut）

哈哈，这个你可能就比较容易理解了，切点在 Spring AOP 中确实是对应系统中的方法。但是这个方法是定义在切面中的方法，一般和通知一起使用，一起组成了切面。

连接点（Join point）

比如：方法调用、方法执行、字段设置/获取、异常处理执行、类初始化、甚至是 for 循环中的某个点

理论上，程序执行过程中的任何时点都可以作为作为织入点，而所有这些执行时点都是 Joint point

但 Spring AOP 目前仅支持方法执行 (method execution) 也可以这样理解，连接点就是你准备在系统中执行切点和切入通知的地方（一般是一个方法，一个字段）

切面（Aspect）

切面是切点和通知的集合，一般单独作为一个类。通知和切点共同定义了关于切面的全部内容，它是什么时候，在何时和何处完成功能。

引入（Introduction）

引用允许我们向现有的类添加新的方法或者属性

织入（Weaving）

组装方面来创建一个被通知对象。这可以在 编译 时完成（例如使用 AspectJ 编译器），也可以在运行时完成。Spring 和其他纯 Java AOP 框架一样，在运行时完成织入。

5. Spring AOP 实现原理

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

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。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。

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class DynaProxyHello implements InvocationHandler {
    private Object target;//目标对象
    /**
     * 通过反射来实例化目标对象
     * @param object
     * @return
     */
    public Object bind(Object object){
        this.target = object;
        return Proxy.newProxyInstance(this.target.getClass().getClassLoader(),
```

```
                this.target.getClass().getInterfaces(),this);
        }
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            Object result = null;
            Logger.start();
            result = method.invoke(this.target, args);
            Logger.end();
            return result;
        }
    }
```

6.动态代理（cglib 与 JDK）

JDK 动态代理类和委托类需要都实现同一个接口。也就是说只有实现了某个接口的类可以使用 Java 动态代理机制。但是，事实上使用中并不是遇到的所有类都会给你实现一个接口。因此，对于没有实现接口的类，就不能使用该机制。而 CGLIB 则可以实现对类的动态代理。

（1）静态代理：

```
    package net.battier.dao;
    public interface Count {
        public void queryCount();
        public void updateCount();
    }
    public class CountImpl implements Count {
        @Override
        public void queryCount() {
            System.out.println("查看账户...");
        }
        @Override
        public void updateCount() {
            System.out.println("修改账户...");
        }
    }
    public class CountProxy implements Count {
        private CountImpl countImpl;   //组合一个业务实现类对象来进行真正的业务方法的调用
        public CountProxy(CountImpl countImpl) {
            this.countImpl = countImpl;
        }
        @Override
        public void queryCount() {
```

```
            System.out.println("查询账户的预处理————————");
            // 调用真正的查询账户方法
            countImpl.queryCount();
            System.out.println("查询账户之后————————————");
        }
        @Override
        public void updateCount() {
            System.out.println("修改账户之前的预处理————————");
            // 调用真正的修改账户操作
            countImpl.updateCount();
            System.out.println("修改账户之后————————————————");
        }
    }
        public static void main(String[] args) {
            CountImpl countImpl = new CountImpl();
            CountProxy countProxy = new CountProxy(countImpl);
            countProxy.updateCount();
            countProxy.queryCount();

        }
```

（2）jdk 动态代理

```
    public interface BookFacade {
        public void addBook();
    }
    public class BookFacadeImpl implements BookFacade {
        @Override
        public void addBook() {
            System.out.println("增加图书方法。。。");
        }
    }
    public class BookFacadeProxy implements InvocationHandler {
        private Object target;//这其实业务实现类对象，用来调用具体的业务方法
        /**
         * 绑定业务对象并返回一个代理类
         */
        public Object bind(Object target) {
            this.target = target;   //接收业务实现类对象参数
           //通过反射机制，创建一个代理类对象实例并返回。用户进行方法调用时使用
            //创建代理对象时，需要传递该业务类的类加载器（用来获取业务实现类的元数据，在包装方法是调用真正的业务方法）、接口、handler 实现类
            return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                    target.getClass().getInterfaces(), this); }
        /**
         * 包装调用方法：进行预处理、调用后处理
```

```
         */
        public Object invoke(Object proxy, Method method, Object[] args)
                throws Throwable {
            Object result=null;
            System.out.println("预处理操作————————");
            //调用真正的业务方法
            result=method.invoke(target, args);
            System.out.println("调用后处理————————");
            return result;
        }
    }
    public static void main(String[] args) {
            BookFacadeImpl bookFacadeImpl=new BookFacadeImpl();
            BookFacadeProxy proxy = new BookFacadeProxy();
            BookFacade bookfacade = (BookFacade) proxy.bind(bookFacadeImpl);
            bookfacade.addBook();
        }
```

（3）Cglib 动态代理

```
    public class BookFacadeImpl1 {
        public void addBook() {
            System.out.println("新增图书...");
        }
    }
    public class BookFacadeCglib implements MethodInterceptor {
        private Object target;//业务类对象，供代理方法中进行真正的业务方法调用
        //相当于 JDK 动态代理中的绑定
        public Object getInstance(Object target) {
            this.target = target;   //给业务对象赋值
            Enhancer enhancer = new Enhancer(); //创建加强器，用来创建动态代理类
            enhancer.setSuperclass(this.target.getClass());   //为加强器指定要代理的业务类（即：为下面生成的代理类指定父类）
            //设置回调：对于代理类上所有方法的调用，都会调用 CallBack，而 Callback 则需要实现 intercept()方法进行拦
            enhancer.setCallback(this);
           // 创建动态代理类对象并返回
           return enhancer.create();
        }
        // 实现回调方法
        public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {
            System.out.println("预处理————————");
            proxy.invokeSuper(obj, args); //调用业务类（父类中）的方法
            System.out.println("调用后操作————————");
            return null;
```

```
    }
public static void main(String[] args) {
        BookFacadeImpl1 bookFacade=new BookFacadeImpl1()；
        BookFacadeCglib   cglib=new BookFacadeCglib();
        BookFacadeImpl1 bookCglib=(BookFacadeImpl1)cglib.getInstance(bookFacade);
        bookCglib.addBook();
    }
```

7.Spring 事务实现方式

一.事务的 4 个特性：

原子性：一个事务中所有对数据库的操作是一个不可分割的操作序列，要么全做，要么全部做。

一致性：数据不会因为事务的执行而遭到破坏。

隔离性：一个事务的执行，不受其他事务（进程）的干扰。既并发执行的个事务之间互不干扰。

持久性：一个事务一旦提交，它对数据库的改变将是永久的。

二.事务的实现方式：

实现方式共有两种：编码方式；声明式事务管理方式。

基于 AOP 技术实现的声明式事务管理，实质就是：在方法执行前后进行拦截，然后在目标方法开始之前创建并加入事务，执行完目标方法后根据执行情况提交或回滚事务。

声明式事务管理又有两种方式：基于 XML 配置文件的方式；另一个是在业务方法上进行 @Transactional 注解，将事务规则应用到业务逻辑中。

三.创建事务的时机：

是否需要创建事务，是由事务传播行为控制的。读数据不需要或只为其指定只读事务，而数据的插入，修改，删除就需要事务管理了。

Spring 配置文件中关于事务配置总是由三个组成部分，分别是 DataSource、TransactionManager 和代理机制这三部分，无论哪种配置方式，一般变化的只是代理机制这部分。

DataSource、TransactionManager 这两部分只是会根据数据访问方式有所变化，比如使用 Hibernate 进行数据访问时，DataSource 实际为 SessionFactory，TransactionManager 的实现为 HibernateTransactionManager。

```
<tx:annotation-driven transaction-manager="transactionManager"/>
    <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property                                          name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>
    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
```

```
            <property name="sessionFactory" ref="sessionFactory" />
        </bean>
    </beans>
```

此时在 Service 类上需加上@Transactional 注解

8.Spring 事务底层原理

a、划分处理单元——IOC

由于 spring 解决的问题是对单个数据库进行局部事务处理的，具体的实现首相用 spring 中的 IOC 划分了事务处理单元。并且将对事务的各种配置放到了 ioc 容器中（设置事务管理器，设置事务的传播特性及隔离机制）。

b、AOP 拦截需要进行事务处理的类

Spring 事务处理模块是通过 AOP 功能来实现声明式事务处理的，具体操作（比如事务实行的配置和读取，事务对象的抽象），用 TransactionProxyFactoryBean 接口来使用 AOP 功能，生成 proxy 代理对象，通过 TransactionInterceptor 完成对代理方法的拦截，将事务处理的功能编织到拦截的方法中。读取 ioc 容器事务配置属性，转化为 spring 事务处理需要的内部数据结构（TransactionAttributeSourceAdvisor），转化为 TransactionAttribute 表示的数据对象。

c、对事物处理实现（事务的生成、提交、回滚、挂起）

spring 委托给具体的事务处理器实现。实现了一个抽象和适配。适配的具体事务处理器：DataSource 数据源支持、hibernate 数据源事务处理支持、JDO 数据源事务处理支持，JPA、JTA 数据源事务处理支持。这些支持都是通过设计 PlatformTransactionManager、AbstractPlatforTransaction 一系列事务处理的支持。 为常用数据源支持提供了一系列的 TransactionManager。

d、结合

PlatformTransactionManager 实现了 TransactionInterception 接口，让其与 TransactionProxyFactoryBean 结合起来，形成一个 Spring 声明式事务处理的设计体系。

9. 自定义注解实现功能

创建自定义注解和创建一个接口相似，但是注解的 interface 关键字需要以@符号开头。

注解方法不能带有参数；

注解方法返回值类型限定为：基本类型、String、Enums、Annotation 或者是这些类型的数组；

注解方法可以有默认值；

注解本身能够包含元注解，元注解被用来注解其它注解。

注解有什么用？ 注解的作用基本有三个：

生成文档。这是最常见的，也是 java 最早提供的注解。常用的有 @see @param @return 等

跟踪代码依赖性，实现替代配置文件功能。比较常见的是 spring 2.5 开始的基于注解配置。作用就是减少配置。现在的框架基本都使用了这种配置来减少配置文件的数量。也是

在编译时进行格式检查。如@override 放在方法前，如果你这个方法并不是覆盖了超类方法，则编译时就能检查出。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    RetentionPolicy value();
}
```

用@Retention(RetentionPolicy.CLASS)修饰的注解，表示注解的信息被保留在 class 文件(字节码文件)中当程序编译时，但不会被虚拟机读取在运行的时候；

用@Retention(RetentionPolicy.SOURCE)修饰的注解,表示注解的信息会被编译器抛弃，不会留在 class 文件中，注解的信息只会留在源文件中；

用@Retention(RetentionPolicy.RUNTIME)修饰的注解，表示注解的信息被保留在 class 文件(字节码文件)中当程序编译时，会被虚拟机保留在运行时。

还需要注意的是 java 的元注解一共有四个：

@Document ：一个简单的 Annotations 标记注解，表示是否将注解信息添加在 java 文档中。

@Target：表示该注解用于什么地方。默认值为任何元素，表示该注解用于什么地方。可用的 ElementType 参数包括：

- ElementType.CONSTRUCTOR: 用于描述构造器
- ElementType.FIELD: 成员变量、对象、属性（包括 enum 实例）
- ElementType.LOCAL_VARIABLE: 用于描述局部变量
- ElementType.METHOD: 用于描述方法
- ElementType.PACKAGE: 用于描述包
- ElementType.PARAMETER: 用于描述参数
- ElementType.TYPE: 用于描述类、接口(包括注解类型) 或 enum 声明

@Inherited：@Inherited 阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited 修饰的 annotation 类型被用于一个 class，则这个 annotation 将被用于该 class 的子类。

常见标准的 Annotation：

1.）Override

java.lang.Override 是一个标记类型注解，它被用作标注方法。它说明了被标注的方法重载了父类的方法，起到了断言的作用。如果我们使用了这种注解在一个没有覆盖父类方法的方法时，java 编译器将以一个编译错误来警示。

2.）Deprecated

Deprecated 也是一种标记类型注解。当一个类型或者类型成员使用@Deprecated 修饰的话，编译器将不鼓励使用这个被标注的程序元素。所以使用这种修饰具有一定的“延续性”：如果我们在代码中通过继承或者覆盖的方式使用了这个过时的类型或者成员，虽然继承或者覆盖后的类型或者成员并不是被声明为@Deprecated，但编译器仍然要报警。

3.）SuppressWarnings

SuppressWarning 不是一个标记类型注解。它有一个类型为 String[] 的成员，这个成员的值为被禁止的警告名。对于 javac 编译器来讲，被-Xlint 选项有效的警告名也同样对@SuppressWarings 有效，同时编译器忽略掉无法识别的警告名。

```
@SuppressWarnings("unchecked")
```

自定义注解：

自定义注解类编写的一些规则:

1. Annotation 型定义为 @interface, 所有的 Annotation 会自动继承

java.lang.Annotation 这一接口,并且不能再去继承别的类或是接口.

2. 参数成员只能用 public 或默认(default) 这两个访问权修饰

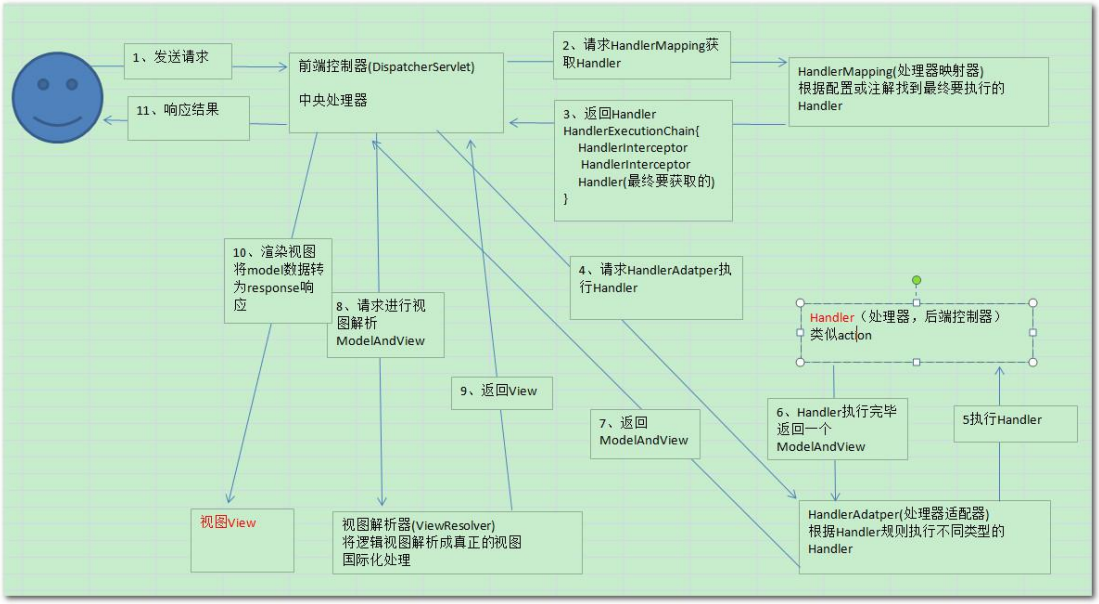
3. 参数成员只能用基本类型 byte、short、char、int、long、float、double、boolean 八种基本数据类型和 String、Enum、Class、annotations 等数据类型，以及这一些类型的数组.

4. 要获取类方法和字段的注解信息，必须通过 Java 的反射技术来获取 Annotation 对象，因为你除此之外没有别的获取注解对象的方法

5. 注解也可以没有定义成员,，不过这样注解就没啥用了

PS:自定义注解需要使用到元注解。

10. Spring MVC 运行流程



SpringMVC 流程:

1、 用户发送请求至前端控制器 DispatcherServlet。

2、 DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。

3、 处理器映射器找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

4、 DispatcherServlet 调用 HandlerAdapter 处理器适配器。

5、 HandlerAdapter 经过适配调用具体的处理器(Controller，也叫后端控制器)。

6、 Controller 执行完成返回 ModelAndView。

7、 HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。

8、 DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器。

9、 ViewReslover 解析后返回具体 View。

10、DispatcherServlet 根据 View 进行渲染视图（即将模型数据填充至视图中）。

11、 DispatcherServlet 响应用户。

组件说明：

以下组件通常使用框架提供实现：

DispatcherServlet：作为前端控制器，整个流程控制的中心，控制其它组件执行，统一调度，降低组件之间的耦合性，提高每个组件的扩展性。

HandlerMapping：通过扩展处理器映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

HandlAdapter：通过扩展处理器适配器，支持更多类型的处理器。

ViewResolver：通过扩展视图解析器，支持更多类型的视图解析，例如：jsp、freemarker、pdf、excel 等。

组件：

1、前端控制器 DispatcherServlet（不需要工程师开发）,由框架提供

作用：接收请求，响应结果，相当于转发器，中央处理器。有了 dispatcherServlet 减少了其它组件之间的耦合度。

用户请求到达前端控制器，它就相当于 mvc 模式中的 c，dispatcherServlet 是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet 的存在降低了组件之间的耦合性。

2、处理器映射器 HandlerMapping(不需要工程师开发),由框架提供

作用：根据请求的 url 查找 Handler

HandlerMapping 负责根据用户请求找到 Handler 即处理器，springmvc 提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

3、处理器适配器 HandlerAdapter

作用：按照特定规则（HandlerAdapter 要求的规则）去执行 Handler

通过 HandlerAdapter 对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

4、处理器 Handler(需要工程师开发)

注意：编写 Handler 时按照 HandlerAdapter 的要求去做，这样适配器才可以去正确执行 Handler

Handler 是继 DispatcherServlet 前端控制器的后端控制器，在 DispatcherServlet 的控制下 Handler 对具体的用户请求进行处理。

由于 Handler 涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发 Handler。

5、视图解析器 View resolver(不需要工程师开发),由框架提供

作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）

View Resolver 负责将处理结果生成 View 视图，View Resolver 首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成 View 视图对象，最后对 View 进行渲染将处理结果通过页面展示给用户。 springmvc 框架提供了很多的 View 视图类型，包括：jstlView、freemarkerView、pdfView 等。

一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

6、视图 View(需要工程师开发 jsp...)

View 是一个接口，实现类支持不同的 View 类型（jsp、freemarker、pdf...）

11. Spring MVC 启动流程

接下来以一个常见的简单 web.xml 配置进行 Spring MVC 启动过程的分析，web.xml 配置内容如下:

```
<web-app>
  <display-name>Web Application</display-name>
  <!--全局变量配置-->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext-*.xml</param-value>
  </context-param>
  <!--监听器-->
  <listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  <!--解决乱码问题的 filter-->
  <filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
      <param-name>encoding</param-name>
      <param-value>utf-8</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <!--Restful 前端控制器-->
  <servlet>
    <servlet-name>springMVC_rest</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>springMVC_rest</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

tomcat web 容器启动时会去读取 web.xml 这样的部署描述文件，相关组件启动顺序为:解析<context-param> => 解析<listener> => 解析<filter> => 解析<servlet>，具体初始化过程如下:

1、解析<context-param>里的键值对。

2、创建一个 application 内置对象即 ServletContext，servlet 上下文，用于全局共享。

3、将<context-param>的键值对放入 ServletContext 即 application 中，Web 应用内全局共享。

4、读取<listener>标签创建监听器，一般会使用 ContextLoaderListener 类，如果使用了 ContextLoaderListener 类，Spring 就会创建一个 WebApplicationContext 类的对象，WebApplicationContext 类就是 IoC 容器，ContextLoaderListener 类创建的 IoC 容器是根 IoC 容器为全局性的，并将其放置在 appication 中，作为应用内全局共享，键名为 WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE，可以通过以下两种方法获取:

WebApplicationContext applicationContext = (WebApplicationContext) application.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);

WebApplicationContext applicationContext1 = WebApplicationContextUtils.getWebApplicationContext(application);

这个全局的根 IoC 容器只能获取到在该容器中创建的 Bean 不能访问到其他容器创建的 Bean，也就是读取 web.xml 配置的 contextConfigLocation 参数的 xml 文件来创建对应的 Bean。

5、listener 创建完成后如果有<filter>则会去创建 filter。

6、初始化创建<servlet>，一般使用 DispatchServlet 类。

7、DispatchServlet 的父类 FrameworkServlet 会重写其父类的 initServletBean 方法，并调用 initWebApplicationContext()以及 onRefresh()方法。

8、initWebApplicationContext()方法会创建一个当前 servlet 的一个 IoC 子容器，如果存在上述的全局 WebApplicationContext 则将其设置为父容器，如果不存在上述全局的则父容器为 null。

9、读取<servlet>标签的<init-param>配置的 xml 文件并加载相关 Bean。

10、onRefresh()方法创建 Web 应用相关组件。

12.Spring 的单例实现原理

单例模式有饿汉模式、懒汉模式、静态内部类、枚举等方式实现，但由于以上模式的构造方法是私有的，不可继承，Spring 为实现单例类可继承，使用的是单例注册表的方式（登记式单例）。

什么是单例注册表呢，

登记式单例实际上维护的是一组单例类的实例，将这些实例存储到一个 Map(登记簿)中，对于已经登记过的单例，则从工厂直接返回，对于没有登记的，则先登记，而后返回

1. 使用 map 实现注册表；

2. 使用 protect 修饰构造方法；

有的时候，我们不希望在一开始的时候就把一个类写成单例模式，但是在运用的时候，我们却可以像单例一样使用他

最典型的例子就是 spring，他的默认类型就是单例，spring 是如何做到把不是单例的类

变成单例呢？

这就用到了登记式单例

其实登记式单例并没有去改变类，他所做的就是起到一个登记的作用，如果没有登记，他就给你登记，并把生成的实例保存起来，下次你要用的时候直接给你。

IOC 容器就是做的这个事，你需要就找他去拿，他就可以很方便的实现 Bean 的管理。

懒汉式饿汉式这种通过私有化构造函数，静态方法提供实例的单例类而言，是不支持继承的。这种模式的单例实现要求每个具体的单例类自身来维护单例实例和限制多个实例的生成。可以采用另外一种实现单例的思路：登记式单例，来使得单例对继承开放。

懒汉式饿汉式的 getInstance()方法都是无参的，返回本类的单例实例。而登记式单例是有参的，根据参数创建不同类的实例加入 Map 中，根据参数返回不同类的单例实例

我们看一个例子：

```
Import java.util.HashMap;
Public class RegSingleton{
     //使用一个 map 来当注册表
    Static private HashMap registry=new HashMap();
    //静态块，在类被加载时自动执行，把 RegistSingleton 自己也纳入容器管理
     Static{
       RegSingleton rs=new RegSingleton();
       Registry.put(rs.getClass().getName(),rs);
    }
//受保护的默认构造函数，如果为继承关系，则可以调用，克服了单例类不能为继承的缺点
Protected RegSingleton(){}
//静态工厂方法，返回此类的唯一实例
public static RegSingleton getInstance(String name){
     if(name==null){
        name=” RegSingleton”;
     }if(registry.get(name)==null){
        try{
             registry.put(name,Class.forName(name).newInstance());
         }Catch(Exception ex){ex.printStackTrace();}
     }
     Return (RegSingleton)registry.get(name);
}
}
```

受保护的构造函数，不能是私有的，但是这样子类可以直接访问构造方法了

解决方式是把你的单例类放到一个外在的包中，以便在其它包中的类（包括缺省的包）无法实例化一个单例类。

```
public abstract class AbstractBeanFactory implements ConfigurableBeanFactory{
    /**
     * 充当了 Bean 实例的缓存，实现方式和单例注册表相同
     */
    private final Map singletonCache=new HashMap();
    public Object getBean(String name)throws BeansException{
```

```
        return getBean(name,null,null);
    }
...
    public Object getBean(String name,Class requiredType,Object[] args)throws BeansException{
        //对传入的 Bean name 稍做处理，防止传入的 Bean name 名有非法字符(或则做转码)
        String beanName=transformedBeanName(name);
        Object bean=null;
        //手工检测单例注册表
        Object sharedInstance=null;
        //使用了代码锁定同步块，原理和同步方法相似，但是这种写法效率更高
        synchronized(this.singletonCache){
            sharedInstance=this.singletonCache.get(beanName);
        }
        if(sharedInstance!=null){
            ...
            //返回合适的缓存 Bean 实例
            bean=getObjectForSharedInstance(name,sharedInstance);
        }else{
            ...
            //取得 Bean 的定义
            RootBeanDefinition mergedBeanDefinition=getMergedBeanDefinition(beanName,false);
            ...
            //根据 Bean 定义判断，此判断依据通常来自于组件配置文件的单例属性开关
            //<bean id="date" class="java.util.Date" scope="singleton"/>
            //如果是单例，做如下处理
            if(mergedBeanDefinition.isSingleton()){
                synchronized(this.singletonCache){
                    //再次检测单例注册表
                    sharedInstance=this.singletonCache.get(beanName);
                    if(sharedInstance==null){
                        ...
                        try {
                            //真正创建 Bean 实例
                            sharedInstance=createBean(beanName,mergedBeanDefinition,args);
                            //向单例注册表注册 Bean 实例
                            addSingleton(beanName,sharedInstance);
                        }catch (Exception ex) {
                            ...
                        }finally{
                            ...
                        }
```

```
                    }
                }
                bean=getObjectForSharedInstance(name,sharedInstance);
            }
            //如果是非单例，即 prototpye，每次都要新创建一个 Bean 实例
            //<bean id="date" class="java.util.Date" scope="prototype"/>
            else{
                bean=createBean(beanName,mergedBeanDefinition,args);
            }
        }
        ...
        return bean;
    }
}
```

13.Spring 框架中用到了哪些设计模式

1. 简单工厂

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。

2. 工厂方法（Factory Method）

定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。

Spring 中的 FactoryBean 就是典型的工厂方法模式。

3. 单例（Singleton）

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。

4. 适配器（Adapter）

将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

Spring 中在对于 AOP 的处理中有 Adapter 模式的例子。

由于 Advisor 链需要的是 MethodInterceptor（拦截器）对象，所以每一个 Advisor 中的 Advice 都要适配成对应的 MethodInterceptor 对象。

5.包装器（Decorator）

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

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。

6. 代理（Proxy）

为其他对象提供一种代理以控制对这个对象的访问。

从结构上来看和 Decorator 模式类似，但 Proxy 是控制，更像是一种对功能的限制，而 Decorator 是增加职责。

Spring 的 Proxy 模式在 aop 中有体现，比如 JdkDynamicAopProxy 和 Cglib2AopProxy。

7.观察者（Observer）

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。

8. 策略（Strategy）

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

9.模板方法（Template Method）

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吧。

14. Spring 其他产品（Srping Boot、Spring Cloud、Spring Secuirity、Spring Data、Spring AMQP 等）