码农求职小助手：Spring高频面试题

笔记本:	11-Spring		
创建时间:	2019/9/7 21:49	**更新时间:**	2019/9/7 21:50
作者:	pc941206@163.com		

更多资料请关注微信公众号：**码农求职小助手**

导读

因为 Spring 源码比较多，不容易阅读。但是 Spring 中最重要的其实就是 AOP 和 IOC，这里我们可以参考一些 Spring 的迷你实现框架，只要大致理解 AOP 和 IOC 的原理就行。

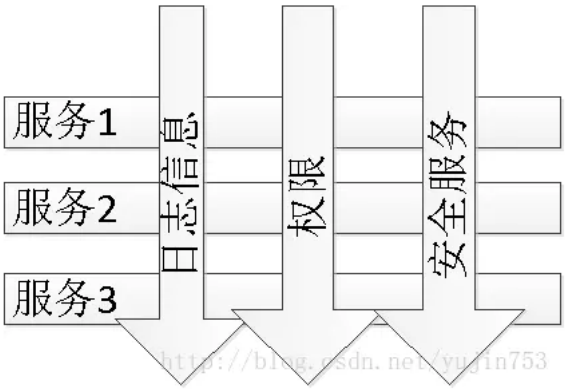
1、自己动手实现的 Spring IOC 和 AOP 上篇

2、自己动手实现的 Spring IOC 和 AOP 下篇

3、Spring Bean 的生命周期

一、AOP

在日常的软件开发中，拿日志来说，一个系统软件的开发都是必须进行日志记录的，不然万一系统出现什么 bug，你都不知道是哪里出了问题。举个小例子，当你开发一个登陆功能，你可能需要在用户登陆前后进行权限校验并将校验信息（用户名、密码、请求登陆时间、ip 地址等）记录在日志文件中，当用户登录进来之后，当他访问某个其他功能时，也需要进行合法性校验。想想看，当系统非常地庞大，系统中专门进行权限验证的代码是非常多的，而且非常地散乱，我们就想能不能**将这些权限校验、日志记录等非业务逻辑功能的部分独立拆分开，并且在系统运行时需要的地方（连接点）进行动态插入运行，不需要的时候就不理**，因此AOP是能够解决这种状况的思想吧！



1、代理

静态代理和动态代理详解：http://www.cnblogs.com/puyangsky/p/6218925.html

AOP 思想的实现一般都是基于 **代理模式** ，在 JAVA 中一般采用 JDK 动态代理模式，但是我们都知道，**JDK 动态代理模式只能代理接口而不能代理类。**因此，Spring AOP 会这样子来进行切换，因为 Spring AOP 同时支持 CGLIB、ASPECTJ、JDK 动态代理。

> 1、如果目标对象的实现类实现了接口，Spring AOP 将会采用 JDK 动态代理来生成 AOP 代理类；
>
> 2、如果目标对象的实现类没有实现接口，Spring AOP 将会采用 CGLIB 来生成 AOP 代理类——不过这个选择过程对开发者完全透明、开发者也无需关心。

- **1.1、JDK 动态代理**

JDK 动态代理最核心的一个接口和方法如下所示：

1、 java.lang.reflect 包中的 InvocationHandler 接口：

```
public interface InvocationHandler {
       public Object invoke(Object proxy, Method method, Object[] args) throws Throwable; }
```

我们对于被代理的类的操作都会由该接口中的 invoke 方法实现，其中的参数的含义分别是：

1、proxy：被代理的类的实例；

2、method：调用被代理的类的方法；

3、args：该方法需要的参数。

使用方法首先是需要实现该接口，并且我们可以在 invoke 方法中调用被代理类的方法并获得返回值，自然也可以在调用该方法的前后去做一些额外的事情，从而实现动态代理

2、java.lang.reflect 包中的 Proxy 类中的 newProxyInstance 方法：

```
public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) throws IllegalArgumentException
```

其中的参数含义如下：

1、loader：被代理的类的类加载器；

2、interfaces：被代理类的接口数组；

3、invocationHandler：调用处理器类的对象实例。

该方法会返回一个被修改过的类的实例，从而可以自由的调用该实例的方法。

- **1.2、CGLIB 动态代理**

CGLIB 是一个字节码增强库，为 AOP 等提供了底层支持。下面看看它是怎么实现动态代理的：

```
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

public class CGlibAgent implements MethodInterceptor {

                private Object proxy;

                public Object getInstance(Object proxy) {
                this.proxy = proxy;
                Enhancer enhancer = new Enhancer(); enhancer.setSuperclass(this.proxy.getClass());
                // 回调方法
                enhancer.setCallback(this);
                // 创建代理对象
                return enhancer.create();
        }
        // 回调方法
        @Override
        public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {

                System.out.println(">>>>before invoking");
                // 真正调用
                Object ret = methodProxy.invokeSuper(o, objects); System.out.println(">>>>after invoking");
                return ret;
        }

        public static void main(String[] args) {

                CGlibAgent cGlibAgent = new CGlibAgent();
                Apple apple = (Apple) cGlibAgent.getInstance(new Apple());
                apple.show();
        }
}
```

2、AOP 的基本概念

1、**切面（Aspect）**：官方的抽象定义为“一个关注点的模块化，这个关注点可能会横切多个对象”。

2、**连接点（Joinpoint）**：程序执行过程中的某一行为。

3、**通知（Advice）**：“切面”对于某个“连接点”所产生的动作。

4、**切入点（Pointcut）**：匹配连接点的断言，在 AOP 中通知和一个切入点表达式关联。

5、**目标对象（Target Object）**：被一个或者多个切面所通知的对象。

6、**AOP 代理（AOP Proxy）**：在 Spring AOP 中有两种代理方式，JDK 动态代理和 CGLIB 代理。

- **通知类型（Advice）：**

1、**前置通知**（Before advice）：在某连接点（JoinPoint）之前执行的通知，但这个通知不能阻止连接点前的执行。ApplicationContext 中在 <aop:aspect> 里面使用 <aop:before> 元素进行声明；

2、**后置通知**（After advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。ApplicationContext 中在 <aop:aspect> 里面使用 <aop:after> 元素进行声明。

3、**返回后通知**（After return advice：在某连接点正常完成后执行的通知，不包括抛出异常的情况。ApplicationContext 中在 <aop:aspect> 里面使用 <<after-returning>> 元素进行声明。

4、**环绕通知**（Around advice）：包围一个连接点的通知，类似 Web 中 Servle t规范中的 Filter 的 doFilter 方法。可以在方法的调用前后完成自定义的行为，也可以选择不执行。ApplicationContext 中在 <aop:aspect> 里面使用 <aop:around> 元素进行声明。

5、**抛出异常后通知**（After throwing advice）：在方法抛出异常退出时执行的通知。ApplicationContext 中在 <aop:aspect> 里面使用 <aop:after-throwing> 元素进行声明。

切入点表达式：如execution(* com.spring.service..(..))

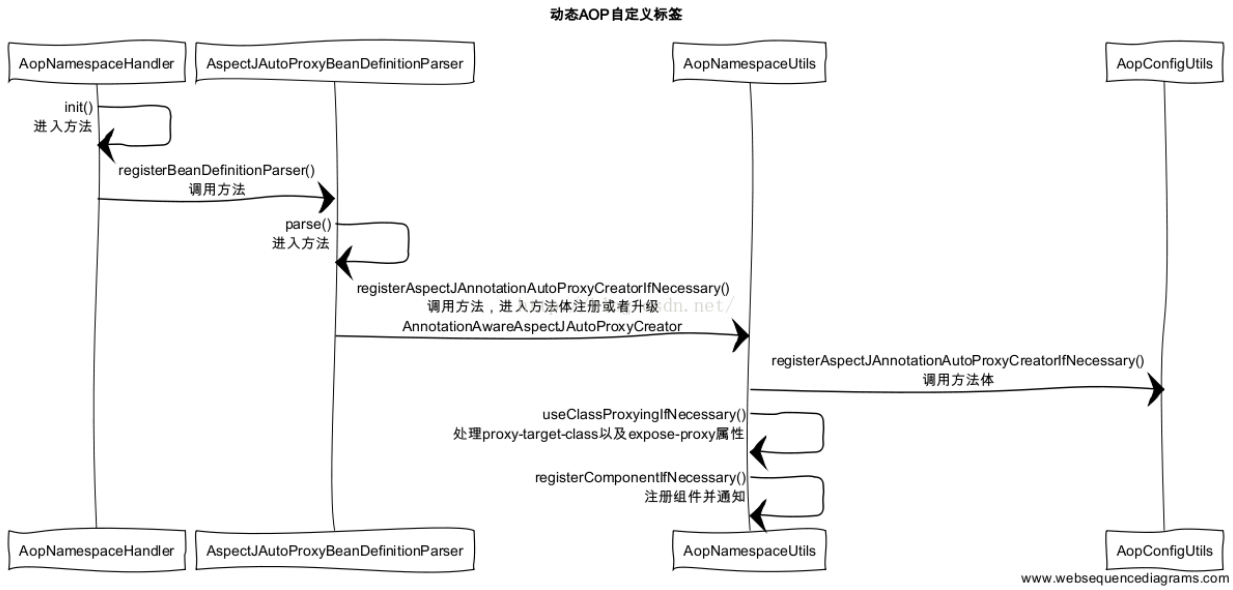
- **AOP 的配置文件举例**：

```
<aop:config>
        <!-- 这是定义一个切面，切面是切点和通知的集合-->
        <aop:aspect id="do" ref="PermissionVerification">
                <!-- 定义切点 ， 后面是 expression 语言，表示包括该接口中定义的所有方法都会被执行-->
                <aop:pointcut id="point" expression="execution(* wokao666.club.aop.spring01.Subject.*(..))" />
                <!-- 定义通知 -->
                <aop:before method="canLogin" pointcut-ref="point" />
                <aop:after method="saveMessage" pointcut-ref="point" />
        </aop:aspect>
</aop:config>
```

3、AOP 的源码分析

详解：https://blog.csdn.net/fighterandknight/article/details/51209822

3.1、动态 AOP 自定义标签



流程说明：

1、AOP 标签的定义解析肯定是从 **NamespaceHandlerSupport** 的实现类开始解析的，这个实现类就是 AopNamespaceHandler；

2、要启用 AOP，我们一般会在 Spring 里面配置 <aop:aspectj-autoproxy/> ，所以在配置文件中在遇到 aspectj-autoproxy 标签的时候我们会采用 **AspectJAutoProxyBeanDefinitionParser** 解析器；

3、进入 AspectJAutoProxyBeanDefinitionParser 解析器后，调用AspectJAutoProxyBeanDefinitionParser 已覆盖 BeanDefinitionParser 的 parser 方法，然后 parser 方法把请求转交给了 AopNamespaceUtils 的 registerAspectJAnnotationAutoProxyCreatorIfNecessary 去处理；

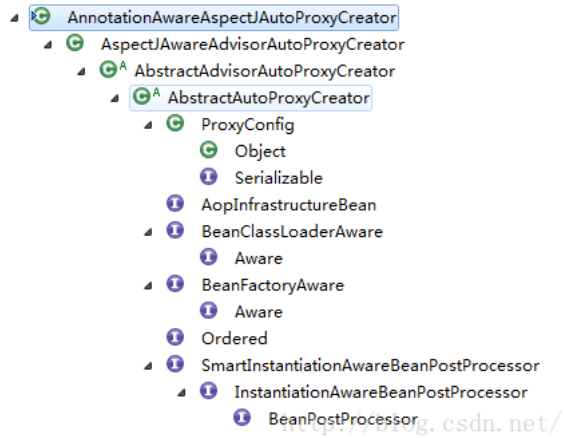
4、进入 AopNamespaceUtils 的registerAspectJAnnotationAutoProxyCreatorIfNecessary 方法后，先调用 AopConfigUtils 的 registerAspectJAnnotationAutoProxyCreatorIfNecessary 方法，里面再转发调用给 registerOrEscalateApcAsRequired，注册或者升级 AnnotationAwareAspectJAutoProxyCreator 类。对于 AOP 的实现，基本是靠 AnnotationAwareAspectJAutoProxyCreator 去完成的，它可以根据 @point 注解定义的切点来代理相匹配的 bean。

5、AopConfigUtils 的 registerAspectJAnnotationAutoProxyCreatorIfNecessary 方法处理完成之后，接下来会调用 useClassProxyingIfNecessary() 处理 proxy-target-class 以及 expose-proxy 属性。**如果将 proxy-target-class 设置为 true 的话，那么会强制使用 CGLIB 代理，否则使用 jdk 动态代理**，expose-proxy 属性是为了解决有时候目标对象内部的自我调用无法实现切面增强。

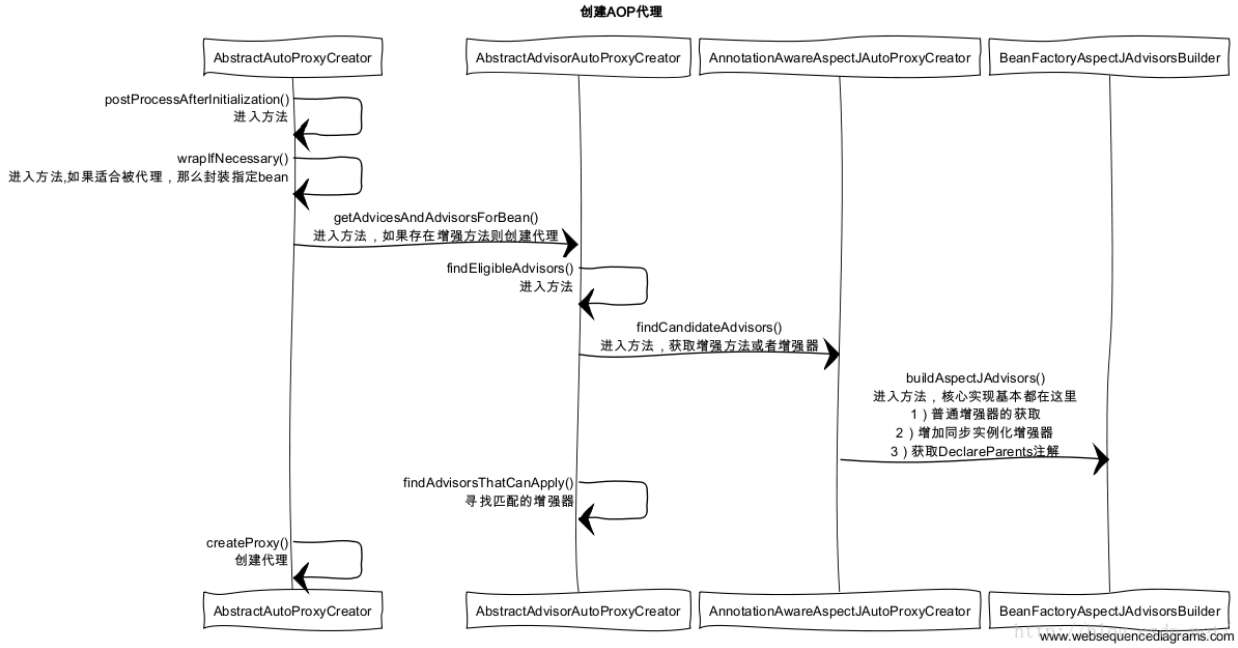
6、最后的调用 registerComponentIfNecessary 方法，注册组建并且通知便于监听器做进一步处理。

3.2 创建 AOP 代理

上面说到 **AOP 的核心逻辑是在 AnnotationAwareAspectJAutoProxyCreator 类里面实现**，那么我们先来看看这个类的层次关系：



我们可以看到这个类实现了 BeanPostProcessor 接口，那就意味着这个类在 Spring 加载实例化前会调用 postProcessAfterInitialization 方法，对于 AOP 的逻辑也是由此开始的。



1、Spring 容器启动，每个 bean 的实例化之前都会先经过 AbstractAutoProxyCreator 类的 postProcessAfterInitialization() 这个方法，然后接下来是调用 wrapIfNecessary 方法；

2、进入 wrapIfNecessary 方法后，我们直接看重点实现逻辑的方法 getAdvicesAndAdvisorsForBean，这个方法会提取当前 bean 的所有增强方法，然后获取到适合的当前 bean 的增强方法，然后对增强方法进行排序，最后返回；

3、**获取到当前 bean 的增强方法后，便调用 createProxy 方法，创建代理。先创建代理工厂 proxyFactory，然后获取当前 bean 的增强器 advisors，把当前获取到的增强器添加到代理工厂 proxyFactory，然后设置当前的代理工厂的代理目标对象为当前 bean，最后根据配置创建 JDK 的动态代理工厂，或者 CGLIB 的动态代理工厂，然后返回 proxyFactory**；

3.3 AOP 动态代理执行

关于 AOP 的动态代理执行，有两种主要的方式 JDK 的动态代理和 CGLIB 的动态代理。我们先来看看 AOP 动态代理的实现选择方式，先上核心实现代码：

```
public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
        if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {
                Class targetClass = config.getTargetClass();
                if (targetClass == null) {
                        throw new AopConfigException("TargetSource cannot determine target class: " +
                                        "Either an interface or a target is required for proxy creation.");
                }
                if (targetClass.isInterface()) {
                        return new JdkDynamicAopProxy(config);  // 如果目标类是接口，则使用JDK动态代理
                }
                return CglibProxyFactory.createCglibProxy(config); // 否则使用CGLIB动态代理
        }
```

```
            else {
                return new JdkDynamicAopProxy(config);
            }
    }
```

3.3.1 Spring JDK 动态代理实现

在上文中说到可以根据用户的配置（例如是否配置了 proxyTargetClass 属性为 true），选择创建代理类型，两种代理类型的实现都是比较高效的，下面根据 JDK 的动态代理来说明 AOP 的执行，也是先上 **JdkDynamicAopProxy** 的核心代码 **invoke** 方法：

```
//  源码省略：可以看原文。大致功能流程如下所示：
```

1、获取拦截器；

2、判断拦截器链是否为空，如果是空的话直接调用切点方法；

3、如果拦截器不为空的话那么便创建 ReflectiveMethodInvocation 类，把拦截器方法都封装在里面，也就是执行 getInterceptorsAndDynamicInterceptionAdvice 方法。

4、其实实际的获取工作是由 AdvisorChainFactory. getInterceptorsAndDynamicInterceptionAdvice() 这个方法来完成的，获取到的结果会被缓存；

> **AdvisorChainFactory. getInterceptorsAndDynamicInterceptionAdvice() 方法的实现**：
>
> 从提供的配置实例 config 中获取 advisor 列表，遍历处理这些 advisor。如果是 IntroductionAdvisor，则判断此 Advisor 能否应用到目标类 targetClass 上；如果是 PointcutAdvisor，则判断此 Advisor 能否应用到目标方法 method 上。将满足条件的 Advisor 通过 AdvisorAdaptor 转化成 Interceptor 列表返回。

5、这个方法执行完成后，Advised 中配置能够应用到连接点或者目标类的 Advisor 全部被转化成了 MethodInterceptor；

6、接下来回到 invoke 方法中的 **proceed 方法**，我们再看下得到的拦截器链是怎么起作用的，也就是 proceed 方法的执行过程：

```
public Object proceed() throws Throwable {
        if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size()- 1) {
            //  如果Interceptor执行完了，则执行joinPoint
            return invokeJoinpoint();
        }

        Object interceptorOrInterceptionAdvice =
      this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);

        //  如果要动态匹配 joinPoint
        if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher){
            InterceptorAndDynamicMethodMatcher dm =
                 (InterceptorAndDynamicMethodMatcher)interceptorOrInterceptionAdvice;
            //  动态匹配：运行时参数是否满足匹配条件
            if (dm.methodMatcher.matches(this.method, this.targetClass,this.arguments)) {
                 // 执行当前 Intercetpor
                 return dm.interceptor.invoke(this);
            }
            else {
                 //  动态匹配失败时，略过当前 Intercetpor，调用下一个Interceptor
                 return proceed();
            }
        }
        else {
            // 执行当前Intercetpor
            return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(this);
        }
}
```

7、好了拦截器到这边就可以执行了，复杂的代理终于可以起到他的作用了。

3.3.2 Spring CGLIB 动态代理实现

由于 CGLIB 的动态代理代码量比较长，在这就不贴出来代码了，其实这两个代理的实现方式都差不多，都是创建方法调用链，不同的是 **JDK 的动态代理创建的是 ReflectiveMethodInvocation 调用链，而 CGLIB 创建的是 CglibMethodInvocation。**

二、IOC

https://javadoop.com/post/spring-ioc

Spring IOC 的初始化过程：

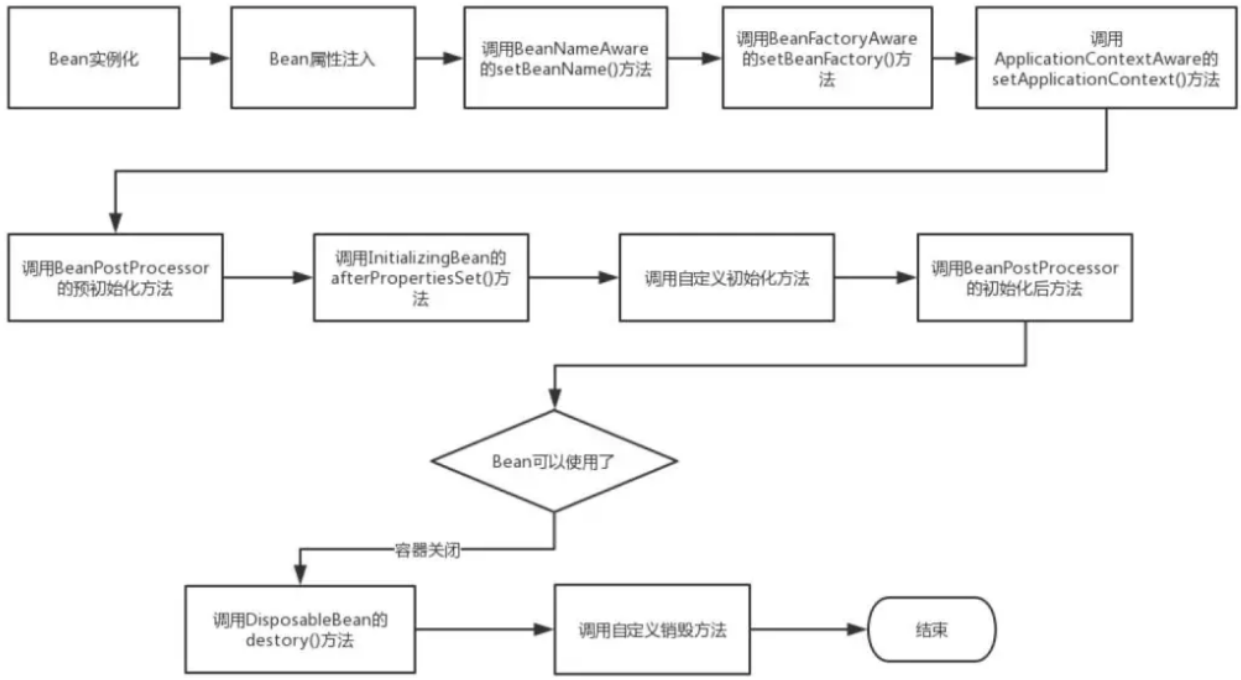
三、Spring 中的 Bean

1、Bean 的生命周期

Bean生命周期详解：https://mp.weixin.qq.com/s/GczkZHJ2DdI7cf9g0e6t_w

在传统的 Java 应用中，bean 的生命周期很简单，使用 Java 关键字 new 进行 Bean 的实例化，然后该 Bean 就能够使用了。一旦 Bean 不再被使用，则由 Java 自动进行垃圾回收。

相比之下，Spring 管理 Bean 的生命周期就复杂多了，正确理解 Bean 的生命周期非常重要，因为 Spring 对 Bean 的管理可扩展性非常强，下面展示了一个 Bean 的构造过程：



如上图所示，Bean 的生命周期还是比较复杂的，下面来对上图每一个步骤做文字描述：

1、Spring 启动，查找并加载需要被 Spring 管理的 Bean，进行 Bean 的实例化；

2、Bean 实例化后，对 Bean 的引入和值注入到 Bean 的属性中；

3、如果 Bean 实现了 BeanNameAware 接口的话，Spring 将 Bean 的 Id 传递给 setBeanName() 方法；

4、如果 Bean 实现了 BeanFactoryAware 接口的话，Spring 将调用 setBeanFactory() 方法，将 BeanFactory 容器实例传入；

5、如果 Bean 实现了 ApplicationContextAware 接口的话，Spring 将调用 Bean 的 setApplicationContext() 方法，将 Bean 所在应用上下文引用传入进来；

6、如果 Bean 实现了 BeanPostProcessor 接口，Spring 就将调用它们的 postProcessBeforeInitialization() 方法；

7、如果 Bean 实现了 InitializingBean 接口，Spring 将调用它们的 afterPropertiesSet() 方法。类似地，如果 Bean 使用 init-method 声明了初始化方法，该方法也会被调用；

8、如果 Bean 实现了 BeanPostProcessor 接口，Spring 就将调用它们的 postProcessAfterInitialization() 方法；

9、此时，Bean 已经准备就绪，可以被应用程序使用了。它们将一直驻留在应用上下文中，直到应用上下文被销毁；

10、如果 Bean 实现了 DisposableBean 接口，Spring 将调用它的 destory() 接口方法，同样，如果 Bean 使用了 destory-method 声明销毁方法，该方法也会被调用。

2、Bean 的作用域

1、**singleton** : 唯一 bean 实例，Spring 中的 bean 默认都是单例的；

2、**prototype** : 每次请求都会创建一个新的 bean 实例；

3、**request** : 每一次 HTTP 请求都会产生一个新的bean，该 bean 仅在当前 HTTP request 内有效；

4、**session** : 每一次 HTTP 请求都会产生一个新的 bean，该 bean 仅在当前 HTTP session 内有效；

5、**global-session**： 全局 session 作用域，仅仅在基于 portlet 的 web 应用中才有意义，Spring5 已经没有了。Portlet 是能够生成语义代码(例如：HTML)片段的小型 Java Web 插件。它们基于 portlet 容器，可以像 servlet 一样处理 HTTP 请求。但是，与 servlet 不同，每个 portlet 都有不同的会话

3、Spring 中的单例 bean 的线程安全问题了解吗?

大部分时候我们并没有在系统中使用多线程，所以很少有人会关注这个问题。单例 bean 存在线程问题，主要是因为：**当多个线程操作同一个对象的时候，对这个对象的非静态成员变量的写操作会存在线程安全问题。**常见的有两种解决办法：

1、在 Bean 对象中尽量避免定义可变的成员变量（不太现实）。

2、**在类中定义一个 ThreadLocal 成员变量，将需要的可变成员变量保存在 ThreadLocal 中（推荐的一种方式）。**

四、Spring 中的事务

Spring 事务详解：https://juejin.im/post/5b00c52ef265da0b95276091

事务是逻辑上的一组操作，要么都执行，要么都不执行。

4.1 事务特性

原子性： 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；

一致性： 执行事务前后，数据保持一致；

隔离性： 并发访问数据库时，一个用户的事物不被其他事物所干扰，各并发事务之间数据库是独立的；

持久性: 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

4.2 Spring 事务管理接口介绍

- **Spring 事务管理接口：**

1、**PlatformTransactionManager**： （平台）事务管理器；

2、**TransactionDefinition**： 事务定义信息（事务隔离级别、传播行为、超时、只读、回滚规则）；

3、**TransactionStatus**： 事务运行状态；

所谓事务管理，其实就是“按照给定的事务规则来执行提交或者回滚操作”。

4.2.1 PlatformTransactionManager 接口介绍

Spring 并不直接管理事务，而是提供了多种事务管理器 ，它们将事务管理的职责委托给 Hibernate 或者 JTA 等持久化机制所提供的相关平台框架的事务来实现。 **Spring 事务管理器的接口是： org.springframework.transaction.PlatformTransactionManager ，通过这个接口，Spring 为各个平台如 JDBC、Hibernate 等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。**

PlatformTransactionManager 接口中定义了三个方法，代码如下：

```
Public interface PlatformTransactionManager()...{

        //  根据指定的传播行为，返回当前活动的事务或创建一个新事务。
        TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;

        //  使用事务目前的状态提交事务
        Void commit(TransactionStatus status) throws TransactionException;

        //  对执行的事务进行回滚
        Void rollback(TransactionStatus status) throws TransactionException; }
```

我们刚刚也说了 Spring 中 PlatformTransactionManager 根据不同持久层框架所对应的接口实现类,几个比较常见的如下图所示

事务	说明
org.springframework.jdbc.datasource.DataSourceTransactionManager	使用Spring JDBC或者iBatis进行持久化数据时使用
org.springframework.orm.hibernate3.HibernateTransactionManager	使用Hibernate3.0版本进行持久化数据时使用
org.springframework.orm.jpa.JpaTransactionManager	使用JPA进行数据持久化时使用
org.springframework.transaction.jta.JtaTransactionManager	使用一个JTA实现来管理事务，在一个事务跨越多个资源时使用

比如我们在使用 JDBC 或者 Mybatis 进行数据持久化操作时，我们的 xml 配置通常如下：

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

4.2.2 TransactionDefinition 接口介绍

事务管理器接口 PlatformTransactionManager 通过 getTransaction(TransactionDefinition definition) 方法来得到一个事务，这个方法里面的参数是 TransactionDefinition 类，这个类就定义了一些基本的事务属性。

- **那么什么是事务属性呢?**

事务属性可以理解成事务的一些基本配置，描述了事务策略如何应用到方法上。事务属性包含了 5 个方面。

- **TransactionDefinition 接口中的方法如下:**

TransactionDefinition 接口中定义了 5 个方法以及一些表示事务属性的常量。比如：隔离级别、传播行为等等的常量。我下面只是列出了 TransactionDefinition 接口中的方法而没有给出接口中定义的常量，该接口中的常量信息会在后面依次介绍到。

```
public interface TransactionDefinition {
        // 返回事务的传播行为
        int getPropagationBehavior();

        // 返回事务的隔离级别，事务管理器根据它来控制另外一个事务可以看到本事务内的哪些数据
        int getIsolationLevel();

        // 返回事务的名字
        String getName();

        // 返回事务必须在多少秒内完成
        int getTimeout();

        // 返回是否优化为只读事务
        boolean isReadOnly();
}
```

(1) 事务隔离级别（定义了一个事务可能受其他并发事务影响的程度）

我们先来看一下**并发事务**带来的问题，然后再来介绍一下 TransactionDefinition 接口 中定义了五个表示隔离级别的常量。

- **并发事务带来的问题：**

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务（多个用户对统一数据进行操作）。并发虽然是必须的，但可能会导致一下的问题。

脏读（Dirty read）：当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。

丢失修改（Lost to modify）：指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。例如：事务 1 读取某表中的数据 A=20，事务 2 也读取 A=20，事务 1 修改 A=A-1，事务 2 也修改A=A-1，最终结果 A=19，事务 1 的修改被丢失。

不可重复读（Unrepeatableread）：指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。

幻读（Phantom read）：幻读与不可重复读类似（不可重复度和幻读区别：不可重复读的重点是修改，幻读的重点在于新增或者删除）。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

- **隔离级别**

TransactionDefinition 接口中定义了五个表示隔离级别的常量：

1、**TransactionDefinition.ISOLATION_DEFAULT**：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；

2、**TransactionDefinition.ISOLATION_READ_UNCOMMITTED**：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；

3、**TransactionDefinition.ISOLATION_READ_COMMITTED**：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；

4、**TransactionDefinition.ISOLATION_REPEATABLE_READ**：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；

5、**TransactionDefinition.ISOLATION_SERIALIZABLE**：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。

（2）事务传播行为（为了解决业务层方法之间互相调用的事务问题）

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。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。在 TransactionDefinition 定义中包括了如下几个表示传播行为的常量：

- **支持当前事务的情况：**

TransactionDefinition.PROPAGATION_REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务；

TransactionDefinition.PROPAGATION_SUPPORTS： 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行；

TransactionDefinition.PROPAGATION_MANDATORY： 如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。（mandatory：强制性）

- **不支持当前事务的情况：**

TransactionDefinition.PROPAGATION_REQUIRES_NEW： 创建一个新的事务，如果当前存在事务，则把当前事务挂起；

TransactionDefinition.PROPAGATION_NOT_SUPPORTED： 以非事务方式运行，如果当前存在事务，则把当前事务挂起。

TransactionDefinition.PROPAGATION_NEVER： 以非事务方式运行，如果当前存在事务，则抛出异常。

- **其他情况：**

TransactionDefinition.PROPAGATION_NESTED： 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，则该取值等价于 TransactionDefinition.PROPAGATION_REQUIRED。

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，它们共享相同的概念。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。

(3) 事务超时属性(一个事务允许执行的最长时间)

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。

(4) 事务只读属性（对事物资源是否执行只读操作）

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

(5) 回滚规则（定义事务回滚规则）

这些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。** 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。

4.2.3 TransactionStatus 接口介绍

TransactionStatus 接口用来记录事务的状态。该接口定义了一组方法，用来获取或判断事务的相应状态信息。

PlatformTransactionManager.getTransaction(...) 方法返回一个 TransactionStatus 对象。返回的TransactionStatus 对象可能代表一个新的或已经存在的事务（如果在当前调用堆栈有一个符合条件的事务）。

```
public interface TransactionStatus{
        boolean isNewTransaction();  // 是否是新的事物
        boolean hasSavepoint();       // 是否有恢复点
        void setRollbackOnly();        // 设置为只回滚
        boolean isRollbackOnly();    // 是否为只回滚
        boolean isCompleted;        // 是否已完成
}
```

4.3 Spring 编程式和声明式事务实例讲解

五、Spring 框架中用到了哪些设计模式?

https://mp.weixin.qq.com/s?__biz=Mzg2OTA0Njk0OA==&mid=2247485303&idx=1&sn=9e4626a1e3f001f9b0d84a6fa0cff04a&chksm=cea248bcf9d5c1aaf48b67cc52bac74eb29d603

1、工厂设计模式 : Spring 使用工厂模式通过 BeanFactory、ApplicationContext 创建 bean 对象；

2、代理设计模式 : Spring AOP 功能的实现；

3、单例设计模式 : Spring 中的 Bean 默认都是单例的；

4、模板方法模式 : Spring 中 jdbcTemplate、hibernateTemplate 等以 Template 结尾的对数据库操作的类，它们就使用到了模板模式；

5、包装器设计模式 : 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。这种模式让我们可以根据客户的需求能够动态切换不同的数据源；

6、观察者模式：Spring 事件驱动模型就是观察者模式很经典的一个应用；

7、适配器模式：Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配 Controller。