问题清单：

什么是Spring框架？Spring框架有哪些主要模块？

使用Spring框架有什么好处？

什么是控制反转（IOC）？什么是依赖注入？

请解释下Spring中的IOC？

BeanFactory和ApplicationContext有什么区别？

将Spring配置到你的应用中共有几种方法？

什么基于XML的配置？

什么基Java的配置？

怎样用注解的方式配置Spring？

描述Spring Bean的生命周期？

描述Spring中各种Bean的范围？

什么是Spring的嵌入beans？

Spring框架中的单例bean是否是线程安全的？

请举例说明如何用Spring注入一个Java的集合类？

请举例说明如何在Spring的Bean中注入一个java.util.Properties？

请解释Spring的Bean的自动生成原理？

请辨析自动生成Bean之间模块的区别？

如何开启基于基于注解的自动写入？

请举例说明@Required注解？

请举例说明@Autowired注解？

请举例说明@Qualifier注解？

请说明构造器注入和setter方法注入之间的区别？

Spring框架中不同类型event有什么区别？

FileSystemResource和ClassPathResource有何区别？

请列举Spring框架中用了哪些设计模式？

### 1、什么是Spring框架？Spring框架有哪些主要模块？

Spring框架是一个为Java应用程序的开发提供了综合、广泛的基础性支持的Java平台。Spring帮助开发者解决了开发中基础性的问题，使得开发人员可以专注于应用程序的开发。Spring框架本身亦是按照设计模式精心打造，这使得我们可以在开发环境中安心的集成Spring框架，不必担心Spring是如何在后台进行工作的。

Spring框架至今已集成了20多个模块。这些模块主要被分如下图所示的核心容器、数据访问/集成,、Web、AOP（面向切面编程）、工具、消息和测试模块。

更多信息：Spring 框架教程。

### 2、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：

Dependency Injection(DI) 方法使得构造器和JavaBean properties文件中的依赖关系一目了然。

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。

Spring并没有闭门造车，Spring利用了已有的技术比如ORM框架、logging框架、J2EE、Quartz和JDK Timer，以及其他视图技术。

Spring框架是按照模块的形式来组织的。由包和类的编号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。

要测试一项用Spring开发的应用程序十分简单，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。

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。

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）。

### 3、什么是控制反转(IOC)？什么是依赖注入？

控制反转是应用于软件工程领域中的，在运行时被装配器对象来绑定耦合对象的一种编程技巧，对象之间耦合关系在编译时通常是未知的。在传统的编程方式中，业务逻辑的流程是由应用程序中的早已被设定好关联关系的对象来决定的。在使用控制反转的情况下，业务逻辑的流程是由对象关系图来决定的，该对象关系图由装配器负责实例化，这种实现方式还可以将对象之间的关联关系的定义抽象化。而绑定的过程是通过“依赖注入”实现的。

控制反转是一种以给予应用程序中目标组件更多控制为目的设计范式，并在我们的实际工作中起到了有效的作用。

依赖注入是在编译阶段尚未知所需的功能是来自哪个的类的情况下，将其他对象所依赖的功能对象实例化的模式。这就需要一种机制用来激活相应的组件以提供特定的功能，所以依赖注入是控制反转的基础。否则如果在组件不受框架控制的情况下，框架又怎么知道要创建哪个组件？

在Java中依然注入有以下三种实现方式：

构造器注入

Setter方法注入

接口注入

### 4、请解释下Spring框架中的IoC？

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。

### 5、BeanFactory和ApplicationContext有什么区别？

BeanFactory 可以理解为含有bean集合的工厂类。BeanFactory 包含了种bean的定义，以便在接收到客户端请求时将对应的bean实例化。

BeanFactory还能在实例化对象的时生成协作类之间的关系。此举将bean自身与bean客户端的配置中解放出来。BeanFactory还包含了bean生命周期的控制，调用客户端的初始化方法（initialization methods）和销毁方法（destruction methods）。

从表面上看，application context如同bean factory一样具有bean定义、bean关联关系的设置，根据请求分发bean的功能。但application context在此基础上还提供了其他的功能。

提供了支持国际化的文本消息

统一的资源文件读取方式

已在监听器中注册的bean的事件

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。

1

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。

1

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

3、XmlWebApplicationContext：由Web应用的XML文件读取上下文。

### 6、Spring有几种配置方式？

将Spring配置到应用开发中有以下三种方式：

基于XML的配置

基于注解的配置

基于Java的配置

### 7、如何用基于XML配置的方式配置Spring？

在Spring框架中，依赖和服务需要在专门的配置文件来实现，我常用的XML格式的配置文件。这些配置文件的格式通常用<beans>开头，然后一系列的bean定义和专门的应用配置选项组成。

SpringXML配置的主要目的时候是使所有的Spring组件都可以用xml文件的形式来进行配置。这意味着不会出现其他的Spring配置类型（比如声明的方式或基于Java Class的配置方式）

Spring的XML配置方式是使用被Spring命名空间的所支持的一系列的XML标签来实现的。Spring有以下主要的命名空间：context、beans、jdbc、tx、aop、mvc和aso。

1

2

3

4

5

6

7

8

9

<beans>

    <!-- JSON Support -->

    <bean name="viewResolver" class="org.springframework.web.servlet.view.BeanNameViewResolver"/>

    <bean name="jsonTemplate" class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>

    <bean id="restTemplate" class="org.springframework.web.client.RestTemplate"/>

</beans>

下面这个web.xml仅仅配置了DispatcherServlet，这件最简单的配置便能满足应用程序配置运行时组件的需求。

<web-app>

  <display-name>Archetype Created Web Application</display-name>

  <servlet>

        <servlet-name>spring</servlet-name>

            <servlet-class>

                org.springframework.web.servlet.DispatcherServlet

            </servlet-class>

        <load-on-startup>1</load-on-startup>

    </servlet>

    <servlet-mapping>

        <servlet-name>spring</servlet-name>

        <url-pattern>/</url-pattern>

    </servlet-mapping>

</web-app>

### 8、如何用基于Java配置的方式配置Spring？

Spring对Java配置的支持是由@Configuration注解和@Bean注解来实现的。由@Bean注解的方法将会实例化、配置和初始化一个新对象，这个对象将由Spring的IoC容器来管理。@Bean声明所起到的作用与<bean/> 元素类似。被@Configuration所注解的类则表示这个类的主要目的是作为bean定义的资源。被@Configuration声明的类可以通过在同一个类的内部调用@bean方法来设置嵌入bean的依赖关系。

最简单的@Configuration 声明类请参考下面的代码：

@Configuration

public class AppConfig

{

    @Bean

    public MyService myService() {

        return new MyServiceImpl();

    }

}

对于上面的@Beans配置文件相同的XML配置文件如下：

<beans>

    <bean id="myService" class="com.howtodoinjava.services.MyServiceImpl"/>

</beans>

上述配置方式的实例化方式如下：利用AnnotationConfigApplicationContext 类进行实例化

public static void main(String[] args) {

    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

    MyService myService = ctx.getBean(MyService.class);

    myService.doStuff();

}

要使用组件组建扫描，仅需用@Configuration进行注解即可：

@Configuration

@ComponentScan(basePackages = "com.howtodoinjava")

public class AppConfig  {

    ...

}

在上面的例子中，com.acme包首先会被扫到，然后再容器内查找被@Component 声明的类，找到后将这些类按照Sring bean定义进行注册。

如果你要在你的web应用开发中选用上述的配置的方式的话，需要用AnnotationConfigWebApplicationContext 类来读取配置文件，可以用来配置Spring的Servlet监听器ContrextLoaderListener或者Spring MVC的DispatcherServlet。

<web-app>

    <!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext

        instead of the default XmlWebApplicationContext -->

    <context-param>

        <param-name>contextClass</param-name>

        <param-value>

            org.springframework.web.context.support.AnnotationConfigWebApplicationContext

        </param-value>

    </context-param>

    <!-- Configuration locations must consist of one or more comma- or space-delimited

        fully-qualified @Configuration classes. Fully-qualified packages may also be

        specified for component-scanning -->

    <context-param>

        <param-name>contextConfigLocation</param-name>

        <param-value>com.howtodoinjava.AppConfig</param-value>

    </context-param>

    <!-- Bootstrap the root application context as usual using ContextLoaderListener -->

    <listener>

        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

    </listener>

    <!-- Declare a Spring MVC DispatcherServlet as usual -->

    <servlet>

        <servlet-name>dispatcher</servlet-name>

        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

        <!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext

            instead of the default XmlWebApplicationContext -->

        <init-param>

            <param-name>contextClass</param-name>

            <param-value>

                org.springframework.web.context.support.AnnotationConfigWebApplicationContext

            </param-value>

        </init-param>

        <!-- Again, config locations must consist of one or more comma- or space-delimited

            and fully-qualified @Configuration classes -->

        <init-param>

            <param-name>contextConfigLocation</param-name>

            <param-value>com.howtodoinjava.web.MvcConfig</param-value>

        </init-param>

    </servlet>

    <!-- map all requests for /app/\* to the dispatcher servlet -->

    <servlet-mapping>

        <servlet-name>dispatcher</servlet-name>

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

    </servlet-mapping>

</web-app>

### 9、怎样用注解的方式配置Spring？

Spring在2.5版本以后开始支持用注解的方式来配置依赖注入。可以用注解的方式来替代XML方式的bean描述，可以将bean描述转移到组件类的内部，只需要在相关类上、方法上或者字段声明上使用注解即可。注解注入将会被容器在XML注入之前被处理，所以后者会覆盖掉前者对于同一个属性的处理结果。

注解装配在Spring中是默认关闭的。所以需要在Spring文件中配置一下才能使用基于注解的装配模式。如果你想要在你的应用程序中使用关于注解的方法的话，请参考如下的配置。

<beans>

   <context:annotation-config/>

   <!-- bean definitions go here -->

</beans>

在 <context:annotation-config/>标签配置完成以后，就可以用注解的方式在Spring中向属性、方法和构造方法中自动装配变量。

下面是几种比较重要的注解类型：

@Required：该注解应用于设值方法。

@Autowired：该注解应用于有值设值方法、非设值方法、构造方法和变量。

@Qualifier：该注解和@Autowired注解搭配使用，用于消除特定bean自动装配的歧义。

JSR-250 Annotations：Spring支持基于JSR-250 注解的以下注解，@Resource、@PostConstruct 和 @PreDestroy。

### 10、请解释Spring Bean的生命周期？

Spring一共有五类生命类型，singleton，prototype，request，session，global session

1、singleton指的是bean在整个spring container中有且只有一个，singleton bean默认与spring container一个生命周期级别。

2、prototype bean每次调用都会得到一个新的实例。

3、request，session，global session只有引入spring applicationContext才存在，生命周期大致对应于HTTP中的request，session和application。

Spring Bean的生命周期简单易懂。在一个bean实例被初始化时，需要执行一系列的初始化操作以达到可用的状态。同样的，当一个bean不在被调用时需要进行相关的析构操作，并从bean容器中移除。

Spring bean factory 负责管理在spring容器中被创建的bean的生命周期。Bean的生命周期由两组回调（call back）方法组成。

初始化之后调用的回调方法。

销毁之前调用的回调方法。

Spring框架提供了以下四种方式来管理bean的生命周期事件：

InitializingBean和DisposableBean回调接口

针对特殊行为的其他Aware接口

Bean配置文件中的Custom init()方法和destroy()方法

@PostConstruct和@PreDestroy注解方式

使用customInit()和 customDestroy()方法管理bean生命周期的代码样例如下：

<beans>

    <bean id="demoBean" class="com.howtodoinjava.task.DemoBean"

            init-method="customInit" destroy-method="customDestroy"></bean>

</beans>

更多内容请参考：Spring生命周期Spring Bean Life Cycle。

### 11、Spring Bean的作用域之间有什么区别？

Spring容器中的bean可以分为5个范围。所有范围的名称都是自说明的，但是为了避免混淆，还是让我们来解释一下：

singleton：这种bean范围是默认的，这种范围确保不管接受到多少个请求，每个容器中只有一个bean的实例，单例的模式由bean factory自身来维护。

prototype：原形范围与单例范围相反，为每一个bean请求提供一个实例。

request：在请求bean范围内会每一个来自客户端的网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。

Session：与请求范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。

global-session：global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。

全局作用域与Servlet中的session作用域效果相同。

更多内容请参考 : Spring Bean Scopes。

### 12、什么是Spring inner beans？

在Spring框架中，无论何时bean被使用时，当仅被调用了一个属性。一个明智的做法是将这个bean声明为内部bean。内部bean可以用setter注入“属性”和构造方法注入“构造参数”的方式来实现。

比如，在我们的应用程序中，一个Customer类引用了一个Person类，我们的要做的是创建一个Person的实例，然后在Customer内部使用。

public class Customer

{

    private Person person;

    //Setters and Getters

}

public class Person

{

    private String name;

    private String address;

    private int age;

    //Setters and Getters

}

内部bean的声明方式如下：

<bean id="CustomerBean" class="com.howtodoinjava.common.Customer">

    <property name="person">

        <!-- This is inner bean -->

        <bean class="com.howtodoinjava.common.Person">

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

            <property name="address" value="India" />

            <property name="age" value="34" />

        </bean>

    </property>

</bean>

### 13、Spring框架中的单例Beans是线程安全的么？

Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。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（比如 View Model 对象），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的作用域由“singleton”变更为“prototype”。

线程安全问题都是由全局变量及静态变量引起的。

单例模式的意思就是只有一个实例。单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。

当多用户同时请求一个服务时，容器会给每一个请求分配一个线程，这是多个线程会并发执行该请求多对应的业务逻辑（成员方法），此时就要注意了，如果该处理逻辑中有对该单列状态的修改（体现为该单列的成员属性），则必须考虑线程同步问题

**线程同步解决：**

1、在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。可以用synchronized或Lock加锁实现，或者用wait(),notify()线程互斥。

2、而ThreadLocal则从另一个角度来解决多线程的并发访问。ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

3、用volatile共享变量不取缓存，轻量级锁

### 14、请举例说明如何在Spring中注入一个Java Collection？

Spring提供了以下四种集合类的配置元素：

<list> :   该标签用来装配可重复的list值。

<set> :    该标签用来装配没有重复的set值。

<map>:   该标签可用来注入键和值可以为任何类型的键值对。

<props> : 该标签支持注入键和值都是字符串类型的键值对。

下面看一下具体的例子：

<beans>

   <!-- Definition for javaCollection -->

   <bean id="javaCollection" class="com.howtodoinjava.JavaCollection">

      <!-- java.util.List -->

      <property name="customList">

        <list>

           <value>INDIA</value>

           <value>Pakistan</value>

           <value>USA</value>

           <value>UK</value>

        </list>

      </property>

     <!-- java.util.Set -->

     <property name="customSet">

        <set>

           <value>INDIA</value>

           <value>Pakistan</value>

           <value>USA</value>

           <value>UK</value>

        </set>

      </property>

     <!-- java.util.Map -->

     <property name="customMap">

        <map>

           <entry key="1" value="INDIA"/>

           <entry key="2" value="Pakistan"/>

           <entry key="3" value="USA"/>

           <entry key="4" value="UK"/>

        </map>

      </property>

      <!-- java.util.Properties -->

    <property name="customProperies">

        <props>

            <prop key="admin">admin@nospam.com</prop>

            <prop key="support">support@nospam.com</prop>

        </props>

    </property>

   </bean>

</beans>

### 15、如何向Spring Bean中注入一个Java.util.Properties？

第一种方法是使用如下面代码所示的<props> 标签：

<bean id="adminUser" class="com.howtodoinjava.common.Customer">

    <!-- java.util.Properties -->

    <property name="emails">

        <props>

            <prop key="admin">admin@nospam.com</prop>

            <prop key="support">support@nospam.com</prop>

        </props>

    </property>

</bean>

也可用”util:”命名空间来从properties文件中创建出一个propertiesbean，然后利用setter方法注入bean的引用。

### 16、请解释Spring Bean的自动装配？

在Spring框架中，在配置文件中设定bean的依赖关系是一个很好的机制，Spring容器还可以自动装配合作关系bean之间的关联关系。这意味着Spring可以通过向Bean Factory中注入的方式自动搞定bean之间的依赖关系。自动装配可以设置在每个bean上，也可以设定在特定的bean上。

下面的XML配置文件表明了如何根据名称将一个bean设置为自动装配：

1

<bean id="employeeDAO" class="com.howtodoinjava.EmployeeDAOImpl" autowire="byName" />

除了bean配置文件中提供的自动装配模式，还可以使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在按照如下的配置方式在Spring配置文件进行配置才可以使用。

1

<context:annotation-config />

也可以通过在配置文件中配置AutowiredAnnotationBeanPostProcessor 达到相同的效果。

1

<bean class ="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

配置好以后就可以使用@Autowired来标注了。

@Autowired

public EmployeeDAOImpl ( EmployeeManager manager ) {

    this.manager = manager;

}

### 17、请解释自动装配模式的区别？

在Spring框架中共有5种自动装配，让我们逐一分析。

no：这是Spring框架的默认设置，在该设置下自动装配是关闭的，开发者需要自行在bean定义中用标签明确的设置依赖关系。

byName：该选项可以根据bean名称设置依赖关系。当向一个bean中自动装配一个属性时，容器将根据bean的名称自动在在配置文件中查询一个匹配的bean。如果找到的话，就装配这个属性，如果没找到的话就报错。

byType：该选项可以根据bean类型设置依赖关系。当向一个bean中自动装配一个属性时，容器将根据bean的类型自动在在配置文件中查询一个匹配的bean。如果找到的话，就装配这个属性，如果没找到的话就报错。

constructor：造器的自动装配和byType模式类似，但是仅仅适用于与有构造器相同参数的bean，如果在容器中没有找到与构造器参数类型一致的bean，那么将会抛出异常。

autodetect：该模式自动探测使用构造器自动装配或者byType自动装配。首先，首先会尝试找合适的带参数的构造器，如果找到的话就是用构造器自动装配，如果在bean内部没有找到相应的构造器或者是无参构造器，容器就会自动选择byTpe的自动装配方式。

### 18、如何开启基于注解的自动装配？

要使用 @Autowired，需要注册 AutowiredAnnotationBeanPostProcessor，可以有以下两种方式来实现：

1、引入配置文件中的<bean>下引入 <context:annotation-config>

<beans>

    <context:annotation-config />

</beans>

2、在bean配置文件中直接引入AutowiredAnnotationBeanPostProcessor

<beans>

    <bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

</beans>

### 19、请举例解释@Required注解？

在产品级别的应用中，IoC容器可能声明了数十万了bean，bean与bean之间有着复杂的依赖关系。设值注解方法的短板之一就是验证所有的属性是否被注解是一项十分困难的操作。可以通过在<bean>中设置“dependency-check”来解决这个问题。

在应用程序的生命周期中，你可能不大愿意花时间在验证所有bean的属性是否按照上下文文件正确配置。或者你宁可验证某个bean的特定属性是否被正确的设置。即使是用“dependency-check”属性也不能很好的解决这个问题，在这种情况下，你需要使用@Required 注解。

需要用如下的方式使用来标明bean的设值方法。

public class EmployeeFactoryBean extends AbstractFactoryBean<Object>

{

    private String designation;

    public String getDesignation() {

        return designation;

    }

    @Required

    public void setDesignation(String designation) {

        this.designation = designation;

    }

    //more code here

}

RequiredAnnotationBeanPostProcessor是Spring中的后置处理用来验证被@Required 注解的bean属性是否被正确的设置了。在使用RequiredAnnotationBeanPostProcesso来验证bean属性之前，首先要在IoC容器中对其进行注册：

<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor" />

但是如果没有属性被用 @Required 注解过的话，后置处理器会抛出一个BeanInitializationException 异常。

### 20、请举例解释@Autowired注解？

@Autowired注解对自动装配何时何处被实现提供了更多细粒度的控制。@Autowired注解可以像@Required注解、构造器一样被用于在bean的设值方法上自动装配bean的属性，一个参数或者带有任意名称或带有多个参数的方法。

比如，可以在设值方法上使用@Autowired注解来替代配置文件中的 <property>元素。当Spring容器在setter方法上找到@Autowired注解时，会尝试用byType 自动装配。

当然我们也可以在构造方法上使用@Autowired 注解。带有@Autowired 注解的构造方法意味着在创建一个bean时将会被自动装配，即便在配置文件中使用<constructor-arg> 元素。

public class TextEditor {

   private SpellChecker spellChecker;

   @Autowired

   public TextEditor(SpellChecker spellChecker){

      System.out.println("Inside TextEditor constructor." );

      this.spellChecker = spellChecker;

   }

   public void spellCheck(){

      spellChecker.checkSpelling();

   }

}

下面是没有构造参数的配置方式：

<beans>

   <context:annotation-config/>

   <!-- Definition for textEditor bean without constructor-arg  -->

   <bean id="textEditor" class="com.howtodoinjava.TextEditor">

   </bean>

   <!-- Definition for spellChecker bean -->

   <bean id="spellChecker" class="com.howtodoinjava.SpellChecker">

   </bean>

</beans>

### 21、请举例说明@Qualifier注解？

@Qualifier注解意味着可以在被标注bean的字段上可以自动装配。Qualifier注解可以用来取消Spring不能取消的bean应用。

下面的示例将会在Customer的person属性中自动装配person的值。

public class Customer

{

    @Autowired

    private Person person;

}

下面我们要在配置文件中来配置Person类。

<bean id="customer" class="com.howtodoinjava.common.Customer" />

<bean id="personA" class="com.howtodoinjava.common.Person" >

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

</bean>

<bean id="personB" class="com.howtodoinjava.common.Person" >

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

</bean>

Spring会知道要自动装配哪个person bean么？不会的，但是运行上面的示例时，会抛出下面的异常：

Caused by: org.springframework.beans.factory.NoSuchBeanDefinitionException:

    No unique bean of type [com.howtodoinjava.common.Person] is defined:

        expected single matching bean but found 2: [personA, personB]

要解决上面的问题，需要使用 @Quanlifier注解来告诉Spring容器要装配哪个bean：

public class Customer

{

    @Autowired

    @Qualifier("personA")

    private Person person;

}

### 22、构造方法注入和设值注入有什么区别？

请注意以下明显的区别：

在设值注入方法支持大部分的依赖注入，如果我们仅需要注入int、string和long型的变量，我们不要用设值的方法注入。对于基本类型，如果我们没有注入的话，可以为基本类型设置默认值。在构造方法注入不支持大部分的依赖注入，因为在调用构造方法中必须传入正确的构造参数，否则的话为报错。

设值注入不会重写构造方法的值。如果我们对同一个变量同时使用了构造方法注入又使用了设置方法注入的话，那么构造方法将不能覆盖由设值方法注入的值。很明显，因为构造方法尽在对象被创建时调用。

在使用设值注入时有可能还不能保证某种依赖是否已经被注入，也就是说这时对象的依赖关系有可能是不完整的。而在另一种情况下，构造器注入则不允许生成依赖关系不完整的对象。

在设值注入时如果对象A和对象B互相依赖，在创建对象A时Spring会抛出sObjectCurrentlyInCreationException异常，因为在B对象被创建之前A对象是不能被创建的，反之亦然。所以Spring用设值注入的方法解决了循环依赖的问题，因对象的设值方法是在对象被创建之前被调用的。

### 23、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。ApplicationEvent类和在ApplicationContext接口中处理的事件，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。

public class AllApplicationEventListener implements ApplicationListener < ApplicationEvent >

{

    @Override

    public void onApplicationEvent(ApplicationEvent applicationEvent)

    {

        //process event

    }

}

Spring 提供了以下5中标准的事件：

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。

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。

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。

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。

请求处理事件（RequestHandledEvent）：在Web应用中，当一个http请求（request）结束触发该事件。

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。

public class CustomApplicationEvent extends ApplicationEvent

{

    public CustomApplicationEvent ( Object source, final String msg )

    {

        super(source);

        System.out.println("Created a Custom event");

    }

}

为了监听这个事件，还需要创建一个监听器：

public class CustomEventListener implements ApplicationListener < CustomApplicationEvent >

{

    @Override

    public void onApplicationEvent(CustomApplicationEvent applicationEvent) {

        //handle event

    }

}

之后通过applicationContext接口的publishEvent()方法来发布自定义事件。

CustomApplicationEvent customEvent = new CustomApplicationEvent(applicationContext, "Test message");

applicationContext.publishEvent(customEvent);

### 24、FileSystemResource和ClassPathResource有何区别？

在FileSystemResource 中需要给出spring-config.xml文件在你项目中的相对路径或者绝对路径。在ClassPathResource中spring会在ClassPath中自动搜寻配置文件，所以要把ClassPathResource 文件放在ClassPath下。

如果将spring-config.xml保存在了src文件夹下的话，只需给出配置文件的名称即可，因为src文件夹是默认。

简而言之，ClassPathResource在环境变量中读取配置文件，FileSystemResource在配置文件中读取配置文件。

### 25、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, JmsTemplate, JpaTemplate。

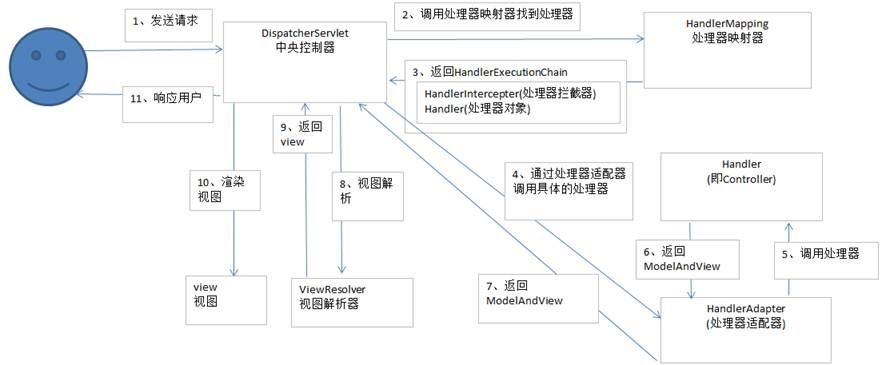
前端控制器—Spring提供了DispatcherServlet来对请求进行分发。

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。

### 26、简单的谈一下SpringMVC的工作流程？



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，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

### 27、Spring 事务属性分析

#### 事务隔离级别

隔离级别是指若干个并发的事务之间的隔离程度。TransactionDefinition 接口中定义了五个表示隔离级别的常量：

TransactionDefinition.ISOLATION\_DEFAULT：这是默认值，表示使用底层数据库的默认隔离级别。对大部分数据库而言，通常这值就是TransactionDefinition.ISOLATION\_READ\_COMMITTED。

TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED：该隔离级别表示一个事务可以读取另一个事务修改但还没有提交的数据。该级别不能防止脏读和不可重复读，因此很少使用该隔离级别。

TransactionDefinition.ISOLATION\_READ\_COMMITTED：该隔离级别表示一个事务只能读取另一个事务已经提交的数据。该级别可以防止脏读，这也是大多数情况下的推荐值。

TransactionDefinition.ISOLATION\_REPEATABLE\_READ：该隔离级别表示一个事务在整个过程中可以多次重复执行某个查询，并且每次返回的记录都相同。即使在多次查询之间有新增的数据满足该查询，这些新增的记录也会被忽略。该级别可以防止脏读和不可重复读。

TransactionDefinition.ISOLATION\_SERIALIZABLE：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。

#### 事务传播行为

所谓事务的传播行为是指，如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。在TransactionDefinition定义中包括了如下几个表示传播行为的常量：

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\_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\_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\_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。

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。

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，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。

事务超时

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。

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

#### 事务的回滚规则

通常情况下，如果在事务中抛出了未检查异常（继承自 RuntimeException 的异常），则默认将回滚事务。如果没有抛出任何异常，或者抛出了已检查异常，则仍然提交事务。这通常也是大多数开发者希望的处理方式，也是 EJB 中的默认处理方式。但是，我们可以根据需要人为控制事务在抛出某些未检查异常时任然提交事务，或者在抛出某些已检查异常时回滚事务。

### 28、Spring AOP

AOP称为面向切面编程，在程序开发中主要用来解决一些系统层面上的问题，比如日志，事务，权限等待，Struts2的拦截器设计就是基于AOP的思想，是个比较经典的例子。AOP代理主要分为静态代理和动态代理，静态代理的代表为AspectJ；而动态代理则以Spring AOP为代表。AspectJ是静态代理的增强，所谓的静态代理就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强。相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理。

#### 一 AOP的基本概念

(1)Aspect(切面):通常是一个类，里面可以定义切入点和通知

(2)JointPoint(连接点):程序执行过程中明确的点，一般是方法的调用

(3)Advice(通知):AOP在特定的切入点上执行的增强处理，有before,after,afterReturning,afterThrowing,around

(4)Pointcut(切入点):就是带有通知的连接点，在程序中主要体现为书写切入点表达式

(5)AOP代理：AOP框架创建的对象，代理就是目标对象的加强。Spring中的AOP代理可以使JDK动态代理，也可以是CGLIB代理，前者基于接口，后者基于子类

#### 二 Spring AOP

Spring中的AOP代理还是离不开Spring的IOC容器，代理的生成，管理及其依赖关系都是由IOC容器负责，Spring默认使用JDK动态代理，在需要代理类而不是代理接口的时候，Spring会自动切换为使用CGLIB代理，不过现在的项目都是面向接口编程，所以JDK动态代理相对来说用的还是多一些。

#### 三 基于注解的AOP配置方式

1.启用@AsjectJ支持

在applicationContext.xml中配置下面一句:

<aop:aspectj-autoproxy />

2.通知类型介绍

(1)Before:在目标方法被调用之前做增强处理,@Before只需要指定切入点表达式即可

(2)AfterReturning:在目标方法正常完成后做增强,@AfterReturning除了指定切入点表达式后，还可以指定一个返回值形参名returning,代表目标方法的返回值

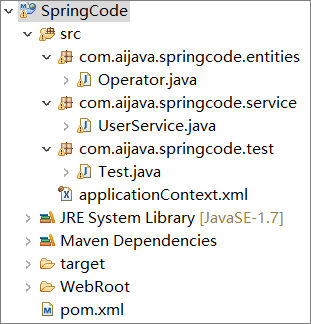
(3)AfterThrowing:主要用来处理程序中未处理的异常,@AfterThrowing除了指定切入点表达式后，还可以指定一个throwing的返回值形参名,可以通过该形参名

来访问目标方法中所抛出的异常对象

(4)After:在目标方法完成之后做增强，无论目标方法时候成功完成。@After可以指定一个切入点表达式

(5)Around:环绕通知,在目标方法完成前后做增强处理,环绕通知是最重要的通知类型,像事务,日志等都是环绕通知,注意编程中核心是一个ProceedingJoinPoint

3.例子：



(1)Operator.java --> 切面类

[复制代码](javascript:void(0);)

@Component

@Aspect

public class Operator {

@Pointcut("execution(\* com.aijava.springcode.service..\*.\*(..))")

public void pointCut(){}

@Before("pointCut()")

public void doBefore(JoinPoint joinPoint){

System.out.println("AOP Before Advice...");

}

@After("pointCut()")

public void doAfter(JoinPoint joinPoint){

System.out.println("AOP After Advice...");

}

@AfterReturning(pointcut="pointCut()",returning="returnVal")

public void afterReturn(JoinPoint joinPoint,Object returnVal){

System.out.println("AOP AfterReturning Advice:" + returnVal);

}

@AfterThrowing(pointcut="pointCut()",throwing="error")

public void afterThrowing(JoinPoint joinPoint,Throwable error){

System.out.println("AOP AfterThrowing Advice..." + error);

System.out.println("AfterThrowing...");

}

@Around("pointCut()")

public void around(ProceedingJoinPoint pjp){

System.out.println("AOP Aronud before...");

try {

pjp.proceed();

} catch (Throwable e) {

e.printStackTrace();

}

System.out.println("AOP Aronud after...");

}

}

[复制代码](javascript:void(0);)

(2)UserService.java --> 定义一些目标方法

[复制代码](javascript:void(0);)

@Service

public class UserService {

public void add(){

System.out.println("UserService add()");

}

public boolean delete(){

System.out.println("UserService delete()");

return true;

}

public void edit(){

System.out.println("UserService edit()");

int i = 5/0;

}

}

[复制代码](javascript:void(0);)

(3).applicationContext.xml

<context:component-scan base-package="com.aijava.springcode"/>

<aop:aspectj-autoproxy />

(4).Test.java

[复制代码](javascript:void(0);)

public class Test {

public static void main(String[] args) {

ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:applicationContext.xml");

UserService userService = (UserService) ctx.getBean("userService");

userService.add();

}

}

[复制代码](javascript:void(0);)

上面是一个比较简单的测试，基本涵盖了各种增强定义。注意:做环绕通知的时候，调用ProceedingJoinPoint的proceed()方法才会执行目标方法。

4.通知执行的优先级

进入目标方法时,先织入Around,再织入Before，退出目标方法时，先织入Around,再织入AfterReturning,最后才织入After。

注意:Spring AOP的环绕通知会影响到AfterThrowing通知的运行,不要同时使用!同时使用也没啥意义。

5.切入点的定义和表达式

切入点表达式的定义算是整个AOP中的核心，有一套自己的规范

Spring AOP支持的切入点指示符：

(1)execution:用来匹配执行方法的连接点

A:@Pointcut("execution(\* com.aijava.springcode.service..\*.\*(..))")

第一个\*表示匹配任意的方法返回值，..(两个点)表示零个或多个，上面的第一个..表示service包及其子包,第二个\*表示所有类,第三个\*表示所有方法，第二个..表示

方法的任意参数个数

B:@Pointcut("within(com.aijava.springcode.service.\*)")

within限定匹配方法的连接点,上面的就是表示匹配service包下的任意连接点

C:@Pointcut("this(com.aijava.springcode.service.UserService)")

this用来限定AOP代理必须是指定类型的实例，如上，指定了一个特定的实例，就是UserService

D:@Pointcut("bean(userService)")

bean也是非常常用的,bean可以指定IOC容器中的bean的名称

6.基于XML形式的配置方式

开发中如果选用XML配置方式，通常就是POJO+XML来开发AOP,大同小异，无非就是在XML文件中写切入点表达式和通知类型

例子：

(1)Log.java

[复制代码](javascript:void(0);)

public class Log {

private Integer id;

//操作名称，方法名

private String operName;

//操作人

private String operator;

//操作参数

private String operParams;

//操作结果 成功/失败

private String operResult;

//结果消息

private String resultMsg;

//操作时间

private Date operTime = new Date();

setter,getter

}

[复制代码](javascript:void(0);)

(2).Logger.java

[复制代码](javascript:void(0);)

/\*\*

\* 日志记录器 （AOP日志通知）

\*/

public class Logger {

@Resource

private LogService logService;

public Object record(ProceedingJoinPoint pjp){

Log log = new Log();

try {

log.setOperator("admin");

String mname = pjp.getSignature().getName();

log.setOperName(mname);

//方法参数,本例中是User user

Object[] args = pjp.getArgs();

log.setOperParams(Arrays.toString(args));

//执行目标方法，返回的是目标方法的返回值，本例中 void

Object obj = pjp.proceed();

if(obj != null){

log.setResultMsg(obj.toString());

}else{

log.setResultMsg(null);

}

log.setOperResult("success");

log.setOperTime(new Date());

return obj;

} catch (Throwable e) {

log.setOperResult("failure");

log.setResultMsg(e.getMessage());

} finally{

logService.saveLog(log);

}

return null;

}

}

[复制代码](javascript:void(0);)

(3).applicationContext.xml

[复制代码](javascript:void(0);)

<aop:config>

<aop:aspect id="loggerAspect" ref="logger">

<aop:around method="record" pointcut="(execution(\* com.aijava.distributed.ssh.service..\*.add\*(..))

or execution(\* com.aijava.distributed.ssh.service..\*.update\*(..))

or execution(\* com.aijava.distributed.ssh.service..\*.delete\*(..)))

and !bean(logService)"/>

</aop:aspect>

</aop:config>

[复制代码](javascript:void(0);)

注意切入点表达式,!bean(logService) 做日志通知的时候，不要给日志本身做日志，否则会造成无限循环！

有关更详细的Spring AOP知识，可以查看Spring官方文档第9章Aspect Oriented Programming with Spring

#### 四 Spring动态代理原理

1.JDK动态代理示例

例子:

(1)UserService.java

public interface UserService {

public void add();

}

(2)UserServiceImpl.java

[复制代码](javascript:void(0);)

public class UserServiceImpl implements UserService{

public void add() {

System.out.println("User add()...");

}

}

[复制代码](javascript:void(0);)

(3)ProxyUtils.java

[复制代码](javascript:void(0);)

public class ProxyUtils implements InvocationHandler{

private Object target;

public ProxyUtils(Object target){

this.target = target;

}

public Object getTarget() {

return target;

}

public void setTarget(Object target) {

this.target = target;

}

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("JDK动态代理，监听开始！");

method.invoke(target, args);

System.out.println("JDK动态代理，监听结束！");

return null;

}

}

[复制代码](javascript:void(0);)

(4)Test.java

[复制代码](javascript:void(0);)

public class Test {

public static void main(String[] args) {

UserService userService = new UserServiceImpl();

ProxyUtils proxyUtils = new ProxyUtils(userService);

UserService proxyObject = (UserService) Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),UserServiceImpl.class.getInterfaces(), proxyUtils);

proxyObject.add();

}

}

[复制代码](javascript:void(0);)

JDK动态代理运行结果

JDK动态代理，监听开始！

User add()...

JDK动态代理，监听结束！

2.CGLIB静态代理示例

Cglib动态代理（需要导入两个jar包，asm-5.2.jar,cglib-3.2.5.jar。版本自行选择）

package com.lf.shejimoshi.proxy.cglib;

import java.lang.reflect.Method;

import com.lf.shejimoshi.proxy.entity.UserManager;

import com.lf.shejimoshi.proxy.entity.UserManagerImpl;

import net.sf.cglib.proxy.Enhancer;

import net.sf.cglib.proxy.MethodInterceptor;

import net.sf.cglib.proxy.MethodProxy;

//Cglib动态代理，实现MethodInterceptor接口

public class CglibProxy implements MethodInterceptor {

private Object target;//需要代理的目标对象

//重写拦截方法

@Override

public Object intercept(Object obj, Method method, Object[] arr, MethodProxy proxy) throws Throwable {

System.out.println("Cglib动态代理，监听开始！");

Object invoke = method.invoke(target, arr);//方法执行，参数：target 目标对象 arr参数数组

System.out.println("Cglib动态代理，监听结束！");

return invoke;

}

//定义获取代理对象方法

public Object getCglibProxy(Object objectTarget){

//为目标对象target赋值

this.target = objectTarget;

Enhancer enhancer = new Enhancer();

//设置父类,因为Cglib是针对指定的类生成一个子类，所以需要指定父类

enhancer.setSuperclass(objectTarget.getClass());

enhancer.setCallback(this);// 设置回调

Object result = enhancer.create();//创建并返回代理对象

return result;

}

public static void main(String[] args) {

CglibProxy cglib = new CglibProxy();//实例化CglibProxy对象

UserService user = (UserService) cglib.getCglibProxy(new UserServiceImpl());//获取代理对象

user.add();//执行删除方法

}

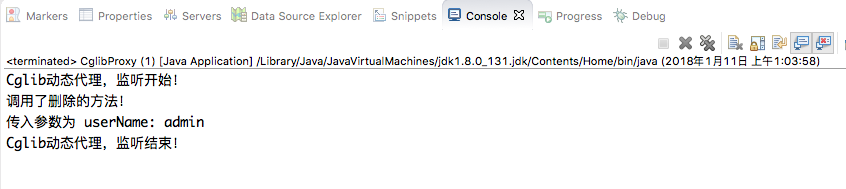
}

Cglib动态代理运行结果

Cglib动态代理，监听开始！

User add()...

Cglib动态代理，监听结束！



3.动态代理和静态代理两者的区别

1、java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理，而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

2、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP   
3、如果目标对象实现了接口，可以强制使用CGLIB实现AOP

4、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

(如何强制使用CGLIB实现AOP？  
 （1）添加CGLIB库，SPRING\_HOME/cglib/\*.jar  
 （2）在spring配置文件中加入<aop:aspectj-autoproxy proxy-target-class="true"/>)

5、JDK动态代理和CGLIB字节码生成的区别？  
 （1）JDK动态代理只能对实现了接口的类生成代理，而不能针对类  
 （2）CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法  
   因为是继承，所以该类或方法最好不要声明成final

6、JDK代理是不需要以来第三方的库，只要要JDK环境就可以进行代理，它有几个要求  
\* 实现InvocationHandler   
\* 使用Proxy.newProxyInstance产生代理对象  
\* 被代理的对象必须要实现接口

CGLib 必须依赖于CGLib的类库，采用非常底层的字节码技术，可以为一个类创建子类，并在子类中采用方法拦截的结束拦截所有父类方法的调用，并顺势织入横切逻辑。是一种继承。