Spring：

1. Spring 的特点？为什么需要采用 Spring 框架？它有什么优势？

   A. Spring 是一个开源的 Java EE 开发框架。Spring 框架的核心功能可以应用在任何 Java 应用程序中，但对 Java EE 平台上的 Web 应用程序有更好的扩展性。Spring 框架的目标是使得 Java EE 应用程序的开发更加简捷，通过使用 POJO 为基础的编程模型促进良好的编程风格。
   B. 轻量级 :Spring 在大小和透明性方面绝对属于轻量级的，基础版本的 Spring 框架大约只有 2MB。
   C. 控制反转(IOC) :Spring 使用控制反转技术实现了松耦合。依赖被注入到对象，而不是创建或寻找依赖对象。
   D. 面向切面编程(AOP)： Spring 支持面向切面编程，同时把应用的业务逻辑与系统的服务分离开来。
   E. 容器：Spring 包含并管理应用程序对象的配置及生命周期。
   F. MVC 框架：Spring 的 web 框架是一个设计优良的 web MVC 框架，很好的取代了一些 web 框架。
   G. 事务管理 :Spring 对下至本地业务上至全局业务(JAT)提供了统一的事务管理接口。
   H. 异常处理：Spring 提供一个方便的 API 将特定技术的异常(由 JDBC, Hibernate, 或 JDO 抛出)转化为一致的、Unchecked 异常。

2. Spring bean 的创建方式？注入方式？
3. 什么是 IoC?什么是 DI？它们有什么好处？怎样理解？

   a) 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。就是被调用类的实例由原先的调用类控制创建、销毁现在转变成由 Spring 的容器管理。
   b) 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？
   c) 在 Java 中依然注入有以下三种实现方式：
      1. 构造器注入
      2. Setter 方法注入
      3. 接口注入

4. Spring 容器的基类？常用实现类？内部的方法？
5. BeanFactory 和 ApplicationContext 有什么区别？

a) 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。
b) 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）。
c) 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。
    1. 提供了支持国际化的文本消息
    2. 统一的资源文件读取方式
    3. 已在监听器中注册的 bean 的事件
d) 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。

```
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。

```
ApplicationContext context = new FileSystemXmlApplicationContext("bean.xml");
```

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

6. ApplicationContext 的事件机制？几个内置事件？怎样在 Bean 中获得 Spring 容器？
7. Spring 框架中的单例 Beans 是线程安全的么？

a) 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。
b) 最浅显的解决办法就是将多态 bean 的作用域由"singleton"变更为"prototype"

8. Spring 容器 bean 的作用域？Spring bean 的自动注入有几个原则(no、byName、byType、constructor、autodetect)？

Spring 框架支持如下五种不同的作用域：

A. singleton：在 Spring IOC 容器中仅存在一个 Bean 实例，Bean 以单实例的方式存在。
B. prototype：一个 bean 可以定义多个实例。

C. request：每次 HTTP 请求都会创建一个新的 Bean。该作用域仅适用于 WebApplicationContext 环境。
D. session：一个 HTTP Session 定义一个 Bean。该作用域仅适用于 WebApplicationContext 环境.
E. globalSession：同一个全局 HTTP Session 定义一个 Bean。该作用域同样仅适用于 WebApplicationContext 环境.

bean 默认的 scope 属性是'singleton'。

Spring bean 的自动注入有几个原则

a) no：默认的方式是不进行自动装配，通过手工设置 ref 属性来进行装配 bean。
b) byName：通过参数名自动装配，Spring 容器查找 beans 的属性，这些 beans 在 XML 配置文件中被设置为 byName。之后容器试图匹配、装配和该 bean 的属性具有相同名字的 bean。
c) byType：通过参数的数据类型自动自动装配，Spring 容器查找 beans 的属性，这些 beans 在 XML 配置文件中被设置为 byType。之后容器试图匹配和装配和该 bean 的属性类型一样的 bean。如果有多个 bean 符合条件，则抛出错误。
d) constructor：这个同 byType 类似，不过是应用于构造函数的参数。如果在 BeanFactory 中不是恰好有一个 bean 与构造函数参数相同类型，则抛出一个严重的错误。
e) autodetect：如果有默认的构造方法，通过 construct 的方式自动装配，否则使用 byType 的方式自动装配。

9. Spring bean 配置使用 abstract 和 parent 可以继承配置，但 depends-on(规定初始化顺序)、lazy-init、singleton、scope 和 autowired 不被继承。
10. FactoryBean 接口的作用？BeanNameAware 接口？
11. 如何管理 Bean 的生命周期？

A. Spring 根据 bean 的定义设置属性值。
B. 如果该 Bean 实现了 BeanNameAware 接口，Spring 将 bean 的 id 传递给 setBeanName()方法。
C. 如果该 Bean 实现了 BeanFactoryAware 接口，Spring 将 beanfactory 传递给 setBeanFactory()方法。
D. 如果任何 bean BeanPostProcessors 和该 bean 相关，Spring 调用 postProcessBeforeInitialization()方法。
E. 如果该 Bean 实现了 InitializingBean 接口，调用 Bean 中的 afterPropertiesSet 方法。如果 bean 有初始化函数声明，调用相应的初始化方法。
F. 如果任何 bean BeanPostProcessors 和该 bean 相关，调用 postProcessAfterInitialization()方法。
G. 如果该 bean 实现了 DisposableBean，调用 destroy()方法。

H. Spring 容器读取 XML 文有两个重要的 bean 生命周期方法。第一个是 setup 方法，该方法在容器加载 bean 的时候被调用。第二个是 teardown 方法，该方法在 bean 从容器中移除的时候调用。

I. bean 标签有两个重要的属性(init-method 和 destroy-method)，你可以通过这两个属性定义自己的初始化方法和析构方法。Spring 也有相应的注解：@PostConstruct 和 @PreDestroy。件中 bean 的定义并实例化 bean。

12. 如何优雅的关闭 Spring 容器？Web 环境下与 WebApplicationContext 同生命周期，非 Web 环境下则调用 AbstractApplicationContext 提供的 registerShutdownHook()方法即可。
13. 协调作用域不同步的 Bean->单例 Bean 内需要注入 prototype 的 bean，需要在使用单例 Bean 某个方法时，每次获得一个新的 Bean。解决方法：bean 属性中可以配置 lookup-method 标签，name 表示单例 bean 的某方法，bean 表示需要获取的 prototype 的 beanid。
14. 高级依赖关系配置：PropertyPathFactoryBean(targetBeanName, propertyPath)可以从一个 Bean 中通过 getter 方法获取属性对象。
    FieldRetrievingFactoryBean(targetClass, targetField)可以获取 Bean 中的字段。
    MethodInvokingFactoryBean(targetObject, targetMethod, arguments)可以获取某类的方法的返回值。
15. 优化 Spring 的配置？分不同的配置文件，使用 p 属性标签代替 property 标签，使用 c 标签代替 constructor-arg 标签，使用 util 命名空间代替 12 条表示的高级依赖关系。
16. spEL 的作用？
17. Spring 的两种后处理器？BeanPostProcesser(Bean 初始化前后)和 BeanFactoryPostProcesser(BeanFactory 前后)
18. Bean 生命周期方法的执行顺序？ 构造方法 -> 设值注入 -> Bean 后处理器初始化前 -> InitializingBean 接口 -> init 方法 -> Bean 后处理器初始化后 -> destory -> DisposableBean 接口。
19. Bean 后处理器的用处？经常用来生成 Bean 的代理，增强 Bean 的功能。
20. 常用的 BeanFactory 后处理器？PropertyPlaceholderConfigurer 后处理器，在 IoC 容器之后载入 Properties 文件。
21. Spring 的注解支持？Component Service Controller Repository。Bean 的属性对应注解：@PostConstruct @PreDestroy @Lazy @DependsOn @Resource @Autowired @Scope
22. Spring 的 Autowired 默认 byType，遇到同样类型的 Bean，注入会失败，此时可以使用@Qualify，限定 Bean id。@Resource 默认实例变量同名。
23. Spring 的资源访问，体现了**策略模式**，不同的资源对应不同的处理方法。Resource 接口？
24. classpath*:和 classpath？classpath*多个同名文件会合并。classpath 只会加载第一个符合条件的配置文件。
25. Spring 的 AOP 原理？分离不变和变化的代码，提取相同的代码，合并横切关注点。
26. Spring AOP 的实现？AspectJ 是 Java 语言的 AOP 实现。AOP 实现可以分为两类，一种是静态 AOP，在编译阶段就对程序进行修改，增强了目标类。如 AspectJ。一

种是动态 AOP，是 AOP 框架在运行阶段动态生成 AOP 代理(在内存中以 JDK 动态代理或 cglib 动态地生成 AOP 代理类)，以实现对目标的增强，如 Spring 的 AOP。

27. JDK 的动态代理和 CGLIB 的原理？JDK 生成代理类实现了目标类的接口，CGLIB 生成了目标类的子类。
28. AOP 的基本概念？切面 Aspect、连接点 JoinPoint、增强处理(通知)Advice、切入点 Pointcut、引入、目标对象、AOP 代理、织入 Weaving(编译时增强和运行时增强)。
29. Spring 的 AOP 默认采用 JDK 的动态代理，也可以使用 cglib，如果需要对成员变量的访问和更新也做为增强处理的连接点，则可以考虑使用 AspectJ。具体使用哪种方式生成由 AopProxyFactory 根据 AdvisedSupport 对象的配置来决定。默认的策略是如果目标类是接口，则使用 JDK 动态代理技术，否则使用 Cglib 来生成代理。
30. 配置 AOP 的过程？在配置文件中开启 aop:aspectj-autoproxy 注解支持，定义业务 Bean 和切面@Aspect，定义切入点@PointCut(execution(* org.*.*(..)))，定义增强处理(通知：前置@Before、后置@After、异常@AfterThrowing、返回@AfterReturning<可以注解 returning 属性获得返回值>、环绕@Around<形参必须使用 ProceedingJoinPoint 类型，执行目标方法为该对象.proceed()方法，该类型还可以得到方法传入的参数，目标对象和代理对象>)
31. 使用@Order 可以指定切面的优先级，越小优先级越高。两个类型相同的增强处理，随机进入。可以使用无返回值的空方法来定义 PointCut，然后在通知定义时使用方法名代替 PointCut，这样也可以跨类使用 PointCut。切入点指示符也有很多种，中间可以以|| && !连接。
32. Spring XML 方式配置 AOP，典型配置：

```
<!-- 配置原始 Bean -->
<bean id="atithmeticCaculatorImpl" class="com.bestsonic.spring.aop.xml.AtithmeticCaculatorImpl"></bean>
<!-- 配置切面 Bean -->
<bean id="loggingAspect" class="com.bestsonic.spring.aop.xml.LoggingAspect" />

<!-- 配置 AOP -->
<aop:config>
    <!-- 配置切点表达式 -->
    <aop:pointcut expression="execution(* *.*(..))" id="pointcut" />
    <!-- 配置切面和通知 -->
    <aop:aspect ref="loggingAspect" order="2">
        <!-- 环绕通知 -->
        <aop:around method="aroundMethod" pointcut-ref="pointcut"  />
        <aop:before method="preMethod" pointcut-ref="pointcut" />
    </aop:aspect>
</aop:config>
```

33. Spring 的缓存？类级别的缓存，默认以所有方法参数作为 key 来缓存方法返回的数据。也可以使用@Cacheable 指定方法级别的缓存。不推荐使用 Spring 默认的 SimpleCacheManager 实现，底层是 ConcurrentMap 实现。可以使用 EHCACHE。

34. Spring 的事务？PlatformTransactionManager 是典型的策略模式实现机制，不知道底层如何管理事务，只知道提供 getTransaction()，commit()和 rollback()方法。
35. 事务的隔离级别？事务的传播特性？具体配置？

```
<bean id="bookShopService"
    class="org.simpleit.transaction.BookShopServiceImpl">
    <property name="bookShopDAO" ref="bookDAO"/>
</bean>

<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<tx:advice id="bookShopTxAdvice"
    transaction-manager="transactionManager">
</tx:advice>

<aop:config>
    <aop:pointcut expression="execution(* *.BookShopService.*(..))"
        id="bookShopOperation"/>
    <aop:advisor advice-ref="bookShopTxAdvice"
        pointcut-ref="bookShopOperation"/>
</aop:config>
```

声明事务管理器

声明事务通知

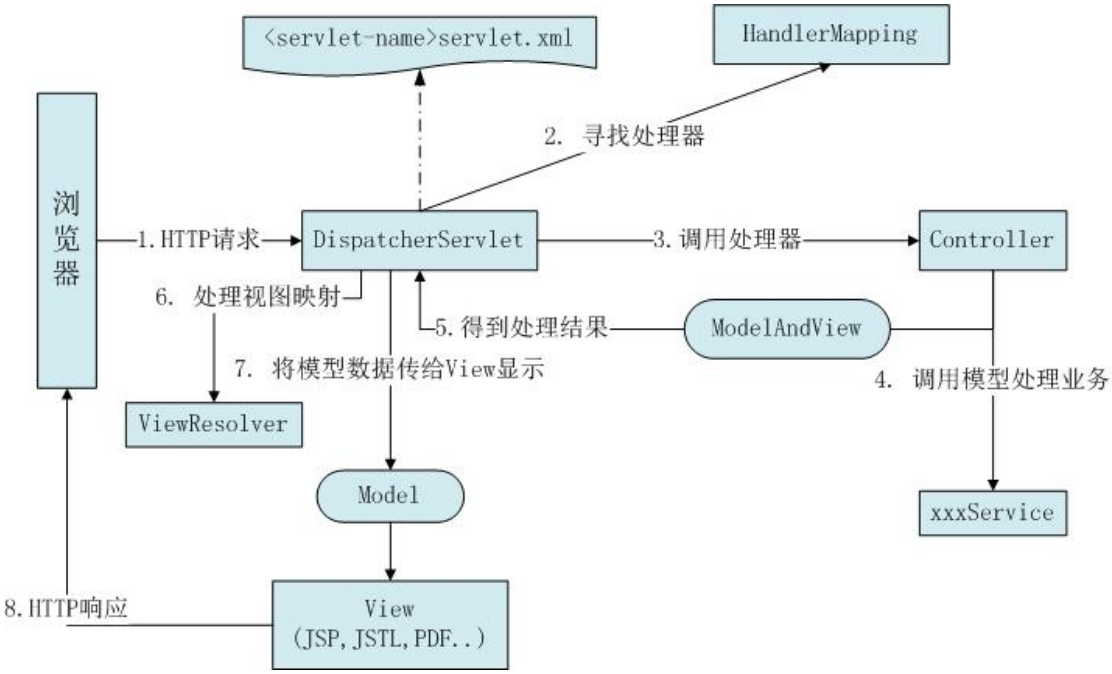
声明 事务通知需要通知方法(即需要进行事务管理的方法)

36. Spring 事务的传播特性？

Spring 提供了多种传播规则供选择

传播行为	含义
PROPAGATION_MANDATORY	表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常
PROPAGATION_NESTED	表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与 PROPAGATION_REQUIRED 一样。注意各厂商对这种传播行为的支持是有所差异的，可以参考资源管理器的文档来确定它们是否支持嵌套式事务
PROPAGATION_NEVER	表示当前方法不应该运行在事务上下文中，如果当前正有一个事务在运行，则会抛出异常
PROPAGATION_NOT_SUPPORTED	表示该方法不应该运行在事务中，如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用 JTATransactionManager 的话，则需要访问 TransactionManager
PROPAGATION_REQUIRED	表示当前方法必须运行在事务中，如果当前事务存在，方法将会在该事务中运行，否则，会启动一个新的事务
PROPAGATION_REQUIRES_NEW	表示当前方法必须运行在它自己的事务中，一个新的事务将被启动，如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用 JTATransactionManager，则需要访问 TransactionManager
PROPAGATION_SUPPORTS	表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行

37. SpringMVC 的运行流程



- DispatcherServlet：Spring 提供的前端控制器，所有的请求都有经过它来统一分发。在 DispatcherServlet 将请求分发给 Spring Controller 之前，需要借助于 Spring 提供的 HandlerMapping 定位到具体的 Controller。
- HandlerMapping：能够完成客户请求到 Controller 映射。
- Controller：需要为并发用户处理上述请求，因此实现 Controller 接口时，必须保证线程安全并且可重用。Controller 将处理用户请求，这和 Struts Action 扮演的角色是一致的。一旦 Controller 处理完用户请求，则返回 ModelAndView 对象给 DispatcherServlet 前端控制器，ModelAndView 中包含了模型（Model）和视图（View）。从宏观角度考虑，DispatcherServlet 是整个 Web 应用的控制器；从微观考虑，Controller 是单个 Http 请求处理过程中的控制器，而 ModelAndView 是 Http 请求过程中返回的模型（Model）和视图（View）。
- ViewResolver：Spring 提供的视图解析器（ViewResolver）在 Web 应用中查找 View 对象，从而将相应结果渲染给客户。

a) 客户端请求提交到 DispatcherServlet
b) 由 DispatcherServlet 控制器查询一个或多个 HandlerMapping，找到处理请求的 Controller
c) DispatcherServlet 将请求提交到 Controller
d) Controller 调用业务逻辑处理后，返回 ModelAndView
e) DispatcherServlet 查询一个或多个 ViewResoler 视图解析器，找到 ModelAndView 指定的视图
f) 视图负责将结果显示到客户端

38. servlet 生命周期?

a) 读取 Servlet 类
b) 创建 Servlet 实例
c) Web 容器调用 Servlet 的 init() 方法

d) 响应客户端请求通过 Servlet 中 service()方法中相应的 doXXX()方法
e) 调用 Servlet 的 destroy()