1 解释 IoC

Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在 Java 开发中，Ioc 意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。如何理解好 Ioc 呢？理解好 Ioc 的关键是要明确“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”，那我们来深入分析一下：

●谁控制谁，控制什么：传统 Java SE 程序设计，我们直接在对象内部通过 new 进行创建对象，是程序主动去创建依赖对象；而 IoC 是有专门一个容器来创建这些对象，即由 Ioc 容器来控制对象的创建；谁控制谁？当然是 IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。

●为何是反转，哪些方面反转了：有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

2 IoC 的优势

IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了 IoC 容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

其实 IoC 对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在 IoC/DI 思想中，应用程序就变成被动的了，被动的等待 IoC 容器来创建并注入它所需要的资源了。

我们还是从 USB 的例子说起，使用 USB 外部设备比使用内置硬盘，到底带来什么好处？

第一、USB 设备作为电脑主机的外部设备，在插入主机之前，与电脑主机没有任何的关系，只有被我们连接在一起之后，两者才发生联系，具有相关性。所以，无论两者中的任何一方出现什么的问题，都不会影响另一方的运行。这种特性体现在软件工程中，就是可维护性比较好，非常便于进行单元测试，便于调试程序和诊断故障。代码中的每一个 Class 都可以单独测试，彼此之间互不影响，只要保证自身的功能无误即可，这就是组件之间低耦合或者无耦合带来的好处。

第二、USB 设备和电脑主机的之间无关性，还带来了另外一个好处，生产 USB 设备的厂商和生产电脑主机的厂商完全可以是互不相干的人，各干各事，他们之间唯一需要遵守的就是 USB 接口标准。这种特性体现在软件开发过程中，好处可是太大了。每个开发团队的成员都只需要关心实现自身的业务逻辑，完全不用去关心其它的人工作进展，因为你的任务跟别人没有任何关系，你的任务可以单独测试，你的任务也不用依赖于别人的组件，再也不用扯不清责任了。所以，在一个大中型项目中，团队成员分工明确、责任明晰，很容易将一个大的任务划分为细小的任务，开发效率和产品质量必将得到大幅度的提高。

第三、同一个 USB 外部设备可以插接到任何支持 USB 的设备，可以插接到电脑主机，也可以插接到 DV 机，USB 外部设备可以被反复利用。在软件工程中，这种特性就是可复用性好，我们可以把具有普遍性的常用组件独立出来，反复利用到项目中的其它部分，或者是其它项目，当然这也是面向对象的基本特征。显然，IOC 不仅更好地贯彻了这个原则，提高了模块的可复用性。符合接口标准的实现，都可以插接到支持此标准的模块中。

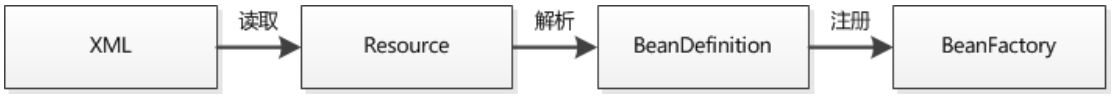
第四、同 USB 外部设备一样，模块具有热插拔特性。IOC 生成对象的方式转为外置方式，也就是把对象生成放在配置文件里进行定义，这样，当我们更换一个实现子类将会变得

很简单，只要修改配置文件就可以了，完全具有热插拔的特性。

3 Spring IOC 核心源码

1） 初始化

初始化的过程主要就是读取 XML 资源，并解析，最终注册到 Bean Factory 中：




**准备：**

**保存配置位置，并刷新**

在调用 ClassPathXmlApplicationContext 后，先会将配置位置信息保存到 configLocations，供后面解析使用，之后，会调用 AbstractApplicationContext 的 refresh 方法进行刷新

**创建载入 BeanFactory**

**创建 XMLBeanDefinitionReader**

**读取：**

**创建处理每一个 resource**

**处理 XML 每个元素**

**解析和注册 bean**

**解析：**

**处理每个 Bean 的元素**

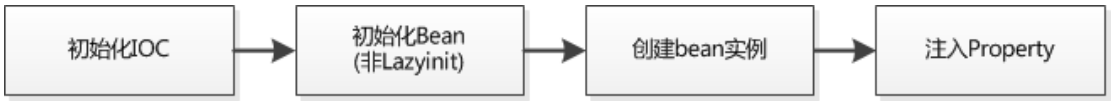
**处理属性的值**

**注册：**

注册过程中，最核心的一句就是：this.beanDefinitionMap.put(beanName, beanDefinition)，也就是说注册的实质就是以 beanName 为 key，以 beanDefinition 为 value，将其 put 到 HashMap 中。

2） 注入依赖

当完成初始化 IOC 容器后，如果 bean 没有设置 lazy-init(延迟加载)属性，那么 bean 的实例就会在初始化 IOC 完成之后，及时地进行初始化。初始化时会先建立实例，然后根据配置利用反射对实例进行进一步操作，具体流程如下所示：




**创建 bean 的实例**

**注入 bean 的属性**

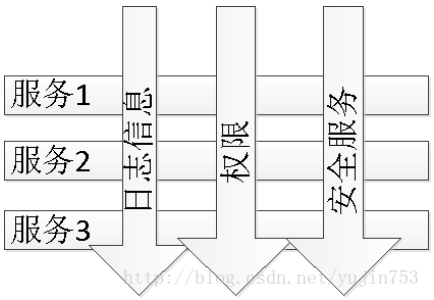
4 Spring AOP

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。

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；

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。

面向切面——Spring 提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如审计（auditing）和事务（transaction）管理）进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责（甚至是意识）其它的系统级关注点，例如日志或事务支持。




5 JDK 动态代理、CGLIB 动态代理讲解

1）JDK 动态代理

java.lang.reflect 包里的 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：

- proxy：被代理的类的实例
- method：调用被代理的类的方法
- args：该方法需要的参数

java.lang.reflect 包中的 Proxy 类的 newProxyInstance 方法：

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

其中的参数含义如下：

- loader：被代理的类的类加载器
- interfaces：被代理类的接口数组
- invocationHandler：就是刚刚介绍的调用处理器类的对象实例

例子：

接口

```
public interface Fruit {
    public void show();
}
```

接口的实现类

```
public class Apple implements Fruit{
    @Override
    public void show() {
        System.out.println("<<<<show method is invoked");
    }
}
```

代理类

```
public class DynamicAgent {

    //实现 InvocationHandler 接口，并且可以初始化被代理类的对象
    static class MyHandler implements InvocationHandler {
        private Object proxy;
        public MyHandler(Object proxy) {
            this.proxy = proxy;
        }

        //自定义 invoke 方法
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
            System.out.println(">>>>before invoking");
            //真正调用方法的地方
            Object ret = method.invoke(this.proxy, args);
            System.out.println(">>>>after invoking");
            return ret;
        }
    }

    //返回一个被修改过的对象
    public static Object agent(Class interfaceClazz, Object proxy) {
        return Proxy.newProxyInstance(interfaceClazz.getClassLoader(), new
Class[]{interfaceClazz},
                new MyHandler(proxy));
    }
}
```

测试

```
public class ReflectTest {
    public static void main(String[] args) throws InvocationTargetException,
IllegalAccessException {
        //注意一定要返回接口，不能返回实现类否则会报错
        Fruit fruit = (Fruit) DynamicAgent.agent(Fruit.class, new Apple());
        fruit.show();
    }
```

```
}
```

2）CGLIB 动态代理

```
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();
    }
}
```

6 Spring 事务管理

事务是逻辑上的一组操作，要么都执行，要么都不执行。事务的特性：

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。

Spring 事务管理接口：

- PlatformTransactionManager： （平台）事务管理器
- 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)
- TransactionStatus： 事务运行状态

Spring 并不直接管理事务，而是提供了多种事务管理器 ，他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：

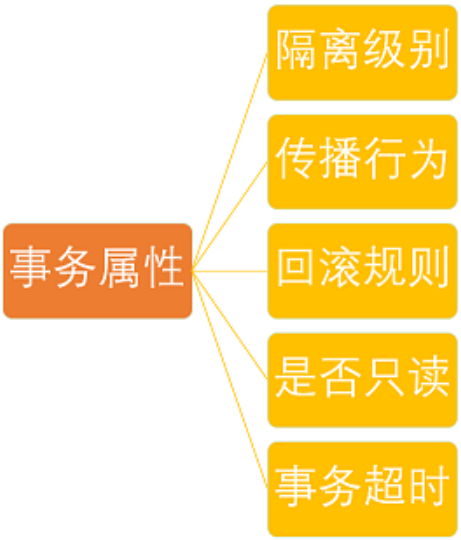
```
Public interface PlatformTransactionManager()...{
    // Return a currently active transaction or create a new one, according to the specified propagation behavior（根据指定的传播行为，返回当前活动的事务或创建一个新事务。）
    TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;
    // Commit the given transaction, with regard to its status（使用事务目前的状态提交事务）
    Void commit(TransactionStatus status) throws TransactionException;
    // Perform a rollback of the given transaction（对执行的事务进行回滚）
    Void rollback(TransactionStatus status) throws TransactionException;
    }
```

| 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 iBatis（就是 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>
```

TransactionDefinition 接口介绍




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。

```
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();
    // 返回事务必须在多少秒内完成
    //返回事务的名字
    String getName();
    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）：

- 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。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。

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

- 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.
- 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
- 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
- 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。
- 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。

(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（这一行为与 EJB 的回滚行为是一致的）。

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。

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.

7 使用 Spring 框架的好处是什么?

- 轻量：Spring 是轻量的，基本的版本大约 2MB。
- 控制反转：Spring 通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
- 面向切面的编程(AOP)：Spring 支持面向切面的编程，并且把应用业务逻辑和系统服务分开。
- 容器：Spring 包含并管理应用中对象的生命周期和配置。
- MVC 框架：Spring 的 WEB 框架是个精心设计的框架，是 Web 框架的一个很好的替代品。
- 事务管理：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。
- 异常处理：Spring 提供方便的 API 把具体技术相关的异常（比如由 JDBC，Hibernate or JDO 抛出的）转化为一致的 unchecked 异常。

8 Spring 由哪些模块组成

以下是 Spring 框架的基本模块:

Core module
Bean module
Context module
Expression Language module
JDBC module
ORM module
OXM module
Java Messaging Service(JMS) module
Transaction module
Web module
Web-Servlet module
Web-Struts module
Web-Portlet module

来自网上的资料:

# 什么是 Spring

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。

---

# Spring 有哪些优点

**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。

**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。

**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**。

**容器**：Spring 包含并管理应用程序对象的配置，依赖关系和生命周期。

**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。

**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。

**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。

---

# Spring 框架有哪些模块

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。

**核心容器模块**：是 spring 中最核心的模块。负责 Bean 的创建，配置和管理。主要包括：beans,core,context,expression 等模块。

**Spring 的 AOP 模块**：主要负责对面向切面编程的支持，帮助应用对象解耦。

**数据访问和集成模块**：包括 JDBC，ORM，OXM，JMS 和事务处理模块，其细节如下： JDBC 模块提供了不再需要冗长的 JDBC 编码相关了 JDBC 的抽象层。 ORM 模块提供的集成层。流行的对象关系映射 API，包括 JPA，JDO，Hibernate 和 iBatis。 OXM 模块提供了一个支持对象/ XML 映射实现对 JAXB，Castor，使用 XMLBeans，JiBX 和 XStream 的抽象层。 Java 消息服务 JMS 模块包含的功能为生产和消费的信息。 事务模块支持编程和声明式事务管理实现特殊接口类，并为所有的 POJO。

**Web 和远程调用**：包括 web,servlet,struts,portlet 模块。

**测试模块**：test

**工具模块消息模块**

---

# 什么是控制反转(IOC)？什么是依赖注入？

传统模式中对象的调用者需要创建被调用对象，两个对象过于耦合，不利于变化和拓展．在 spring 中，直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理，从而实现对象之间的**松耦合**。所谓的“控制反转”概念就是对组件对象控制权的转移，从程序代码本身转移到了外部容器。

**依赖注入**：对象无需自行创建或管理它们的依赖关系，IoC 容器在运行期间，动态地将某种依赖关系注入到对象之中。依赖注入能让相互协作的软件组件保持松散耦合。

---

# BeanFactory 和 ApplicationContext 有什么区别？

Bean 工厂(BeanFactory)是 Spring 框架最核心的接口，提供了高级 Ioc 的配置机制.

应用上下文(ApplicationContext)建立在 BeanFacotry 基础之上，提供了更多面向应用的功能，如果国际化，属性编辑器，事件等等.

beanFactory 是 spring 框架的基础设施，是面向 spring 本身，ApplicationContext 是面向使用 Spring 框架的开发者，几乎所有场合都会用到 ApplicationContext.

---

# 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：** 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

**基于 Java 的配置：** 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。

---

# Spring Bean 的生命周期

Bean 在 Spring 中的生命周期如下：

**实例化**。Spring 通过 new 关键字将一个 Bean 进行实例化，JavaBean 都有默认的构造函数，因此不需要提供构造参数。

**填入属性**。Spring 根据 xml 文件中的配置通过调用 Bean 中的 setXXX 方法填入对应的属性。 事件通知。Spring 依次检查 Bean 是否实现了 BeanNameAware、BeanFactoryAware、ApplicationContextAware、BeanPostProcessor、InitializingBean 接口，如果有的话，依次调用这些接口。

**使用**。应用程序可以正常使用这个 Bean 了。

**销毁**。如果 Bean 实现了 DisposableBean 接口，就调用其 destroy 方法。

**---加载过程---**

1.容器寻找 Bean 的定义信息并且将其实例化。

2.如果允许提前暴露工厂，则提前暴露这个 bean 的工厂，这个工厂主要是返回该未完全处理的 bean. 主要是用于避免单例属性循环依赖问题.

3.受用**依赖注入**，Spring 按照 Bean 定义信息配置 Bean 的所有属性。

4.如果 Bean 实现了 **BeanNameAware** 接口，工厂调用 Bean 的 **setBeanName()**方法传递 Bean 的 ID。

5.如果 Bean 实现了 **BeanFactoryAware** 接口，工厂调用 **setBeanFactory()**方法传入工厂自身。

6.如果 **BeanPostProcessor** 和 Bean 关联，那么它们的 **postProcessBeforeInitialzation()**方法将被调用。

7.如果 Bean 指定了 init-method 方法，它将被调用。

8.如果有 **BeanPostProcessor** 和 Bean 关联，那么它们的 postProcessAfterInitialization()方法将被调用

9.最后如果配置了 destroy-method 方法则注册 **DisposableBean.**

到这个时候，Bean 已经可以被应用系统使用了，并且将被保留在 Bean Factory 中知道它不再需要。 有两种方法可以把它从 Bean Factory 中删除掉。

1.如果 Bean 实现了 DisposableBean 接口，destory()方法被调用。

2.如果指定了订制的销毁方法，就调用这个方法。

---

# Spring Bean 的作用域之间有什么区别

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。

---

# 请解释自动装配模式的区别

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。

---

# Spring 框架中都用到了哪些设计模式

**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—Srping 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。

**Builder 模式**- 自定义配置文件的解析 bean 是时采用 builder 模式，一步一步地构建一个 beanDefinition

**策略模式**：Spring 中策略模式使用有多个地方，如 Bean 定义对象的创建以及代理对象的创建等。这里主要看一下代理对象创建的策略模式的实现。 前面已经了解 Spring 的代理方式有两个 Jdk 动态代理和 CGLIB 代理。这两个代理方式的使用正是使用了策略模式。

---

# AOP 是怎么实现的

实现 AOP 的技术，主要分为两大类：

一是采用**动态代理技术**，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；

二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

Spring AOP 的实现原理其实很简单：AOP 框架负责动态地生成 AOP 代理类，这个代理类的方法则由 Advice 和回调目标对象的方法所组成,并将该对象可作为目标对象使用。AOP 代理包含了目标对象的全部方法，但 AOP 代理中的方法与目标对象的方法存在差异，AOP 方法在特定切入点添加了增强处理，并回调了目标对象的方法。

Spring AOP 使用动态代理技术在运行期织入增强代码。使用两种代理机制：

基于 **JDK 的动态代理**（JDK 本身只提供接口的代理）；

**基于 CGlib 的动态代理**。

1)JDK 的动态代理主要涉及 java.lang.reflect 包中的两个类：Proxy 和 InvocationHandler。其中 InvocationHandler 只是一个接口，可以通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态的将横切逻辑与业务逻辑织在一起。而 Proxy 利用 InvocationHandler 动态创建一个符合某一接口的实例，生成目标类的代理对象。 其代理对象**必须是某个接口的实现**,它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理.只能实现接口的类生成代理,而**不能针对类**

2)CGLib 采用底层的字节码技术，为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类的调用方法，并顺势织入横切逻辑.它运行期间生成的代理对象是目标类的扩展子类.所以无法通知 final 的方法,因为它们不能被覆写.是针对类实现代理,主要是为指定的类生成一个子类,覆盖其中方法.

在 spring 中默认情况下使用 JDK 动态代理实现 AOP,如果 proxy-target-class 设置为 true 或者使用了优化策略那么会使用 CGLIB 来创建动态代理.Spring AOP 在这两种方式的实现上基本一样. 以 JDK 代理为例，会使用 JdkDynamicAopProxy 来创建代理，在 invoke()方法首先需要织入到当前类的增强器封装到拦截器链中，然后递归的调用这些拦截器完成功能的织入. 最终返回代理对象.

http://zhengjianglong.cn/2015/12/12/Spring/spring-source-aop/

---

介绍 spring 的 IOC 实现

Spring　IOC 主要负责创建和管理 bean 及 bean 之间的依赖关系.

Spring　IOC 的可分为:**IOC 容器的初始化和 bean 的加载.**

在 IOC 容器阶段主要是完成资源的加载(如定义 bean 的 xml 文件)，bean 的解析及对解析后得到的 beanDefinition 的进行注册. 以 xmlBeanFactory 为例，XmlBeanFactory 继承了 DefaultListableBeanFactory，XmlBeanFactory 将读取 xml 配置文件，解析 bean 和注册解析后的 beanDefinition 工作交给 XmlBeanDefinitionReader(是 BeanDefinitionReader 接口的一个个性化实现)来执行.spring 中定义了一套资源类，将文件，class 等都看做资源.

1)所以首先是将 xml 文件转化为资源然后用 EncodeResouce 来封装，该功能主要时考虑 Resource 可能存在编码要求的情况，如 UTF-8 等.

2)然后根据 xml 文件判断 xml 的约束模式，是 DTD 还是 Schema,以及寻找模式文档(验证文件)的方法(EntityResolver，这部分采用了代理模式和策略模式). 完成了前面所有的准备工作以后就可以正式的加载配置文件，获取 Docoment 和解析注册 BeanDefinition. Docoment 的获取以及 BeanDefinition 的解析注册并不是由 XmlBeanDefinitionReader 完成，XmlBeanDefinitionReader 只是将前面的工作完成以后文档加载交给 DefaultDocumentLoader 类来完成. 而解析交给了 DefaultBeanDefinitionDocumentReader 来处理.bean 标签可以分为两种，一种是 spring 自带的默认标签，另一种就是用户自定义的标签. 所以 spring 针对这两种情况，提供了不同的解析方式. 每种 bean 的解析完成后都会先注册到容器中然后最后发出响应事件，通知相关的监听器这个 bean 已经注册完成了.

**bean 的加载** http://zhengjianglong.cn/2015/12/06/Spring/spring-source-ioc-bean-parse/

---

# springMVC 流程具体叙述下

当应用启动时,容器会加载 servlet 类并调用 init 方法. 在这个阶段,DispatcherServlet 在 init()完成初始化参数 init-param 的解析和封装,相关配置,spring 的 WebApplicationContext 的初始化即完成 xml 文件的加载,bean 的解析和注册等工作,另外为 servlet 功能所用的变量进行初始化,如:handlerMapping,viewResolvers 等.

当用户发送一个请求时，首先根据请求的类型调用 DispatcherServlet 不同的方法，这些方法都会转发到 doService()中执行. 在该方法内部完成以下工作：

1)spring 首先考虑 multipart 的处理,如果是 MultipartContent 类型的 request,则将该请求转换成 MultipartHttpServletRequest 类型的 request.

2)根据 request 信息获取对应的 Handler. 首先根据 request 获取访问路径,然后根据该路径可以选择直接匹配或通用匹配的方式寻找 Handler,即用户定义的 controller. Handler 在 init()方法时已经完成加载且保存到 Map 中了,只要根据路径就可以得到对应的 Handler. 如果不存在则尝试使用默认的 Handler. 如果还是没有找到那么就通过 response 向用户返回错误信息.找到 handler 后会将其包装在一个执行链中,然后将所有的拦截器也加入到该链中.

4)如果存在 handler 则根据当前的 handler 寻找对应的 HandlerAdapter. 通过遍历所有适配器来选择合适的适配器.

5)SpringMVC 允许你通过处理拦截器 Web 请求,进行前置处理和后置处理.所以在正式调用 Handler 的逻辑方法时,先执行所有拦截器的 preHandle()方法.

6)正式执行 handle 的业务逻辑方法 handle(),返回 ModelAndView.逻辑处理是通过适配器调用 handle 并返回视图.这过程其实是调用用户 controller 的业务逻辑.

8)调用拦截器的 postHandle()方法,完成后置处理.

9)根据视图进行页面跳转.该过程首先会根据视图名字解析得到视图,该过程支持缓存,如果缓存中存在则直接获取,否则创建新的视图并在支持缓存的情况下保存到缓冲中.

10)过程完成了像添加前缀后缀,设置必须的属性等工作.最后就是进行页面跳转处理.

11)调用拦截器的 afterComplection()

AOP 相关概念

**方面（Aspect）**：一个关注点的模块化，这个关注点实现可能另外横切多个对象。事务管理是 J2EE 应用中一个很好的横切关注点例子。方面用 Spring 的 Advisor 或拦截器实现。

**连接点（Joinpoint）**：程序执行过程中明确的点，如方法的调用或特定的异常被抛出。

**通知（Advice）**:在特定的连接点，AOP 框架执行的动作。各种类型的通知包括“around”、“before”和“throws”通知。通知类型将在下面讨论。许多 AOP 框架包括 Spring 都是以拦截器做通知模型，维护一个“围绕”连接点的拦截器链。Spring 中定义了 4 个 advice.Interception Around(MethodInterceptor)、Before(MethodBeforeAdvice)、After Returning(AfterReturningAdvice)、After(AfterAdvice)。

**切入点（Pointcut）**：一系列连接点的集合。AOP 框架必须允许开发者指定切入点：例如，使用正则表达式。 Spring 定义了 Pointcut 接口，用来组合 MethodMatcher 和 ClassFilter，可以通过名字很清楚的理解， MethodMatcher 是用来检查目标类的方法是否可以被应用此通知，而 ClassFilter 是用来检查 Pointcut 是否应该应用到目标类上

**引入（Introduction）**：添加方法或字段到被通知的类。 Spring 允许引入新的接口到任何被通知的对象。例如，你可以使用一个引入使任何对象实现 IsModified 接口，来简化缓存。Spring 中要使用 Introduction, 可有通过 DelegatingIntroductionInterceptor 来实现通知，通过 DefaultIntroductionAdvisor 来配置 Advice 和代理类要实现的接口

**目标对象（Target Object）**：包含连接点的对象。也被称作被通知或被代理对象。

POJO**AOP 代理（AOP Proxy）**：AOP 框架创建的对象，包含通知。 在 Spring 中，AOP 代理可以是 JDK 动态代理或者 CGLIB 代理。

**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。

---

# 过滤器与监听器的区别

Filter 可认为是 Servlet 的一种“变种”，它主要用于对用户请求进行预处理，也可以对 HttpServletResponse 进行后处理，是个典型的处理链。它与

Servlet 的区别在于：它不能直接向用户生成响应。完整的流程是：Filter 对用户请求进行预处理，接着将请求交给 Servlet 进行处理并生成响应，最后 Filter 再对服务器响应进行后处理。 Java 中的 Filter 并不是一个标准的 Servlet ，它不能处理用户请求，也不能对客户端生成响应。 主要用于对 HttpServletRequest 进行预处理，也可以对 HttpServletResponse 进行后处理，是个典型的处理链。优点：过滤链的好处是，执行过程中任何时候都可以打断，只要不执行 chain.doFilter()就不会再执行后面的过滤器和请求的内容。而在实际使用时，就要特别注意过滤链的执行顺序问题
http://blog.csdn.net/sd0902/article/details/8395641

Servlet,Filter 都是针对 url 之类的，而 Listener 是针对对象的操作的，如 session 的创建，session.setAttribute 的发生，或者在启动服务器的时候将你需要的数据加载到缓存等，在这样的事件发生时做一些事情。
http://www.tuicool.com/articles/bmqMjm

---

# 请描述一下 java 事件监听机制。

(1) Java 的事件监听机制涉及到三个组件：事件源、事件监听器、事件对象

(2) 当事件源上发生操作时，它将会调用事件监听器的一个方法，并在调用这个方法时，会传递事件对象过来

(3) 事件监听器由开发人员编写，开发人员在事件监听器中，通过事件对象可以拿到事件源，从而对事件源上的操作进行处理。

---

# 解释核心容器(应用上下文)模块

这是 Spring 的基本模块，它提供了 Spring 框架的基本功能。BeanFactory 是所有 Spring 应用的核心。Spring 框架是建立在这个模块之上的，这也使得 Spring 成为一个容器。

---

# BeanFactory - BeanFactory 实例

BeanFactory 是工厂模式的一种实现，它使用控制反转将应用的配置和依赖与实际的应用代码分离开来。最常用的 BeanFactory 实现是 XmlBeanFactory 类。

XmlBeanFactory

最常用的就是 org.springframework.beans.factory.xml.XmlBeanFactory，它根据 XML 文件中定义的内容加载 beans。该容器从 XML 文件中读取配置元数据，并用它来创建一个完备的系统或应用。

---

# 解释 AOP 模块

AOP 模块用来开发 Spring 应用程序中具有切面性质的部分。该模块的大部分服务由 AOP Aliance 提供，这就保证了 Spring 框架和其他 AOP 框架之间的互操作性。另外，该模块将元数据编程引入到了 Spring。

---

# 解释抽象 JDBC 和 DAO 模块

通过使用抽象 JDBC 和 DAO 模块保证了与数据库连接代码的整洁与简单，同时避免了由于未能关闭数据库资源引起的问题。它在多种数据库服务器的错误信息之上提供了一个很重要的异常层。它还利用 Spring 的 AOP 模块为 Spring 应用程序中的对象提供事务管理服务。

---

# 解释对象/关系映射集成模块

Spring 通过提供 ORM 模块在 JDBC 的基础上支持对象关系映射工具。这样的支持使得 Spring 可以集成主流的 ORM 框架，包括 Hibernate, JDO, 及 iBATIS SQL Maps。Spring 的事务管理可以同时支持以上某种框架和 JDBC。

---

# 解释 web 模块

Spring 的 web 模块建立在应用上下文(application context)模块之上，提供了一个适合基于 web 应用程序的上下文环境。该模块还支持了几个面向 web 的任务，如透明的处理多文件上传请求及将请求参数同业务对象绑定起来。

---

# 解释 Spring MVC 模块

Spring 提供 MVC 框架构建 web 应用程序。Spring 可以很轻松的同其他 MVC 框架结合，但 Spring 的 MVC 是个更好的选择，因为它通过控制反转将控制逻辑和业务对象完全分离开来。

---

# ContextLoaderListener 是监听什么事件

ContextLoaderListener 的作用就是启动 Web 容器时，自动装配 ApplicationContext 的配置信息。因为它实现了 ServletContextListener 这个接口，在 web.xml 配置这个监听器，启动容器时，就会默认执行它实现的方法。

---

# Spring IoC 容器

Spring IOC 负责创建对象、管理对象(通过依赖注入)、整合对象、配置对象以及管理这些对象的生命周期。

---

# IOC 有什么优点？

IOC 或依赖注入减少了应用程序的代码量。它使得应用程序的**测试很简单**，因为在单元测试中不再需要单例或 JNDI 查找机制。简单的实现以及较少的干扰机制使得**松耦合**得以实现。IOC 容器支持勤性单例及延迟加载服务。

---

# 应用上下文是如何实现的？

ClassPathXmlApplicationContext 容器加载 XML 文件中 beans 的定义。XML Bean 配置文件的完整路径必须传递给构造器。
FileSystemXmlApplicationContext 容器也加载 XML 文件中 beans 的定义。注意，你需要正确的设置 CLASSPATH，因为该容器会在 CLASSPATH 中查看 bean 的 XML 配置文件。 WebXmlApplicationContext：该容器加载 xml 文件，这些文件定义了 web 应用中所有的 beans。

---

# 有哪些不同类型的 IOC(依赖注入)

**接口注入**:接口注入的意思是通过接口来实现信息的注入，而其它的类要实现该接口时，就可以实现了注入 构造器依赖注入：构造器依赖注入在容器触发构造器的时候完成，该构造器有一系列的参数，每个参数代表注入的对象。 Setter 方法依赖注入：首先容器会触发一个无参构造函数或无参静态工厂方法实例化对象，之后容器调用 bean 中的 setter 方法完成 Setter 方法依赖注入。

你推荐哪种依赖注入？构造器依赖注入还是 Setter 方法依赖注入？

你可以同时使用两种方式的依赖注入，最好的选择是使用构造器参数实现强制依赖注入，使用 setter 方法实现可选的依赖关系。

---

# 什么是 Spring Beans

Spring Beans 是构成 Spring 应用核心的 Java 对象。这些对象由 Spring IOC 容器实例化、组装、管理。这些对象通过容器中配置的元数据创建，例如，使用 XML 文件中定义的创建。 在 Spring 中创建的 beans 都是单例的 beans。在 bean 标签中有一个属性为”singleton”,如果设为 true，该 bean 是单例的，如果设为 false，该 bean 是原型 bean。Singleton 属性默认设置为 true。因此，spring 框架中所有的 bean 都默认为单例 bean。

---

# Spring Bean 中定义了什么内容？

Spring Bean 中定义了所有的配置元数据，这些配置信息告知容器如何创建它，它的生命周期是什么以及它的依赖关系。

---

# 如何向 Spring 容器提供配置元数据

有三种方式向 Spring 容器提供元数据: XML 配置文件 基于注解配置 基于 Java 的配置

---

# 你如何定义 bean 的作用域

在 Spring 中创建一个 bean 的时候，我们可以声明它的作用域。只需要在 bean 定义的时候通过’scope’属性定义即可。例如，当 Spring 需要产生每次一个新的 bean 实例时，应该声明 bean 的 scope 属性为 prototype。如果每次你希望 Spring 返回一个实例，应该声明 bean 的 scope 属性为 singleton。

---

# Spring 框架中单例 beans 是线程安全的吗？

不是，Spring 框架中的单例 beans 不是线程安全的。

---

# 哪些是最重要的 bean 生命周期方法？能重写它们吗？

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。 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。

---

# 什么是 Spring 的内部 bean

当一个 bean 被用作另一个 bean 的属性时，这个 bean 可以被声明为内部 bean。在基于 XML 的配置元数据中，可以通过把元素定义在 或元素内部实现定义内部 bean。内部 bean 总是匿名的并且它们的 scope 总是 prototype。

---

# 如何在 Spring 中注入 Java 集合类

Spring 提供如下几种类型的集合配置元素： list 元素用来注入一系列的值，允许有相同的值。 set 元素用来注入一些列的值，不允许有相同的值。 map 用来注入一组”键-值”对，键、值可以是任何类型的。 props 也可以用来注入一组”键-值”对，这里的键、值都字符串类型。

# 什么是 bean wiring？

Wiring，或者说 bean Wiring 是指 beans 在 Spring 容器中结合在一起的情况。当装配 bean 的时候，Spring 容器需要知道需要哪些 beans 以及如何使用依赖注入将它们结合起来。

# 什么是 bean 自动装配？

Spring 容器可以自动配置相互协作 beans 之间的关联关系。这意味着 Spring 可以自动配置一个 bean 和其他协作 bean 之间的关系，通过检查 BeanFactory 的内容里没有使用和< property>元素。

# 解释自动装配的各种模式

自动装配提供五种不同的模式供 Spring 容器用来自动装配 beans 之间的依赖注入: 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。 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。 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。 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。 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。

# 自动装配有哪些局限性？

自动装配有如下局限性： 重写：你仍然需要使用 和< property>设置指明依赖，这意味着总要重写自动装配。 原生数据类型:你不能自动装配简单的属性，如原生类型、字符串和类。 模糊特性：自动装配总是没有自定义装配精确，因此，如果可能尽量使用自定义装配。

# 你可以在 Spring 中注入 null 或空字符串吗

完全可以。

---

# 什么是 Spring 基于 Java 的配置？给出一些注解的例子

基于 Java 的配置允许你使用 Java 的注解进行 Spring 的大部分配置而非通过传统的 XML 文件配置。以注解@Configuration 为例，它用来标记类，说明作为 beans 的定义，可以被 Spring IOC 容器使用。另一个例子是@Bean 注解，它表示该方法定义的 Bean 要被注册进 Spring 应用上下文中。

---

# 什么是基于注解的容器配置

另外一种替代 XML 配置的方式为基于注解的配置，这种方式通过字节元数据装配组件而非使用尖括号声明。开发人员将直接在类中进行配置，通过注解标记相关的类、方法或字段声明，而不再使用 XML 描述 bean 之间的连线关系。

---

# 如何开启注解装配？

注解装配默认情况下在 Spring 容器中是不开启的。如果想要开启基于注解的装配只需在 Spring 配置文件中配置元素即可。

---

# @Required 注解

@Required 表明 bean 的属性必须在配置时设置，可以在 bean 的定义中明确指定也可通过自动装配设置。如果 bean 的属性未设置，则抛出 BeanInitializationException 异常。

---

# @Autowired 注解

@Autowired 注解提供更加精细的控制，包括自动装配在何处完成以及如何完成。它可以像@Required 一样自动装配 setter 方法、构造器、属性或者具有任意名称和/或多个参数的 PN 方法。

---

# @Qualifier 注解

当有多个相同类型的 bean 而只有其中的一个需要自动装配时，将@Qualifier 注解和@Autowire 注解结合使用消除这种混淆，指明需要装配的 bean。
Spring 数据访问

---

# 在 Spring 框架中如何更有效的使用 JDBC?

使用 Spring JDBC 框架，资源管理以及错误处理的代价都会减轻。开发人员只需通过 statements 和 queries 语句从数据库中存取数据。Spring 框架中通过使用模板类能更有效的使用 JDBC，也就是所谓的 JdbcTemplate。

---

# JdbcTemplate

JdbcTemplate 类提供了许多方法，为我们与数据库的交互提供了便利。例如，它可以将数据库的数据转化为原生类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据库错误处理功能。

---

# Spring 对 DAO 的支持

Spring 对数据访问对象(DAO)的支持旨在使它可以与数据访问技术(如 JDBC, Hibernate 及 JDO)方便的结合起来工作。这使得我们可以很容易在的不同的持久层技术间切换，编码时也无需担心会抛出特定技术的异常。

---

# 使用 Spring 可以通过什么方式访问 Hibernate？

使用 Spring 有两种方式访问 Hibernate： 使用 Hibernate Template 的反转控制以及回调方法 继承 HibernateDAOSupport，并申请一个 AOP 拦截器节点

---

# Spring 支持的 ORM

Spring 支持一下 ORM： Hibernate iBatis JPA (Java -Persistence API) TopLink JDO (Java Data Objects) OJB

---

# 如何通过 HibernateDaoSupport 将 Spring 和 Hibernate 结合起来？

使用 Spring 的 SessionFactory 调用 LocalSessionFactory。结合过程分为以下三步： 配置 Hibernate SessionFactory 继承 HibernateDaoSupport 实现一个 DAO 使用 AOP 装载事务支持

---

# Spring 支持的事务管理类型

Spring 支持如下两种方式的事务管理： 编码式事务管理：sping 对编码式事务的支持与 EJB 有很大区别，不像 EJB 与 java 事务 API 耦合在一起. spring 通过回调机制将实际的事务实现从事务性代码中抽象出来. 你能够精确控制事务的边界，它们的开始和结束完全取决于你. 声明式事务管理：这种方式意味着你可以将事务管理和业务代码分离。你只需要通过注解或者 XML 配置管理事务。通过传播行为，隔离级别，回滚规则，事务超时，只读提示来定义.

---

# Spring 框架的事务管理有哪些优点

它为不同的事务 API(如 JTA, JDBC, Hibernate, JPA, 和 JDO)提供了统一的编程模型。 它为编程式事务管理提供了一个简单的 API 而非一系列复杂的事务

API(如 JTA). 它支持声明式事务管理。 它可以和 Spring 的多种数据访问技术很好的融合。

---

ACID

**原子性(Atomic)**:一个操作要么成功，要么全部不执行.

**一致性(Consistent)**: 一旦事务完成，系统必须确保它所建模业务处于一致的状态

**隔离性(Isolated)**: 事务允许多个用户对相同的数据进行操作，每个用户用户的操作相互隔离互补影响.

**持久性(Durable)**: 一旦事务完成，事务的结果应该持久化.

---

# spring 事务定义的传播规则

PROPAGATION_REQUIRED - 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

PROPAGATION_SUPPORTS - 支持当前事务，如果当前没有事务，就以非事务方式执行。

PROPAGATION_MANDATORY - 支持当前事务，如果当前没有事务，就抛出异常。

PROPAGATION_REQUIRES_NEW - 新建事务，如果当前存在事务，把当前事务挂起。

PROPAGATION_NOT_SUPPORTED - 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

PROPAGATION_NEVER - 以非事务方式执行，如果当前存在事务，则抛出异常。

PROPAGATION_NESTED - 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与 PROPAGATION_REQUIRED 类似的操作。

---

# spring 事务支持的隔离级别

并发会导致以下问题：

藏读：发生在一个事务读取了另一个事务改写但尚未提交的数据.

不可重复读：在一个事务执行相同的查询两次或两次以上，每次得到的数据不同.

幻读：与不可重复读类似，发生在一个事务读取多行数据，接着另一个并发事务插入一些数据，随后查询中，第一个事务发现多了一些原本不存在的数据.

spring 事务上提供以下的隔离级别:

ISOLATION_DEFAULT: 使用后端数据库默认的隔离级别

ISOLATION_READ_UNCOMMITTED ： 允许读取未提交的数据变更，可能会导致脏读，幻读或不可重复读

ISOLATION_READ_COMMITTD ： 允许读取为提交数据,可以阻止脏读，当时幻读或不可重复读仍可能发生

ISOLATION_REPEATABLE_READ: 对统一字段多次读取结果是一致的，除非数据是被本事务自己修改. 可以阻止脏读，不可重复读，但幻读可能发生

ISOLATION_SERIALIZABLE ： 完全服从 ACID

---

# 你更推荐那种类型的事务管理？

许多 Spring 框架的用户选择声明式事务管理，因为这种方式和应用程序的关联较少，因此更加符合轻量级容器的概念。声明式事务管理要优于编程式事务管理，尽管在灵活性方面它弱于编程式事务管理(这种方式允许你通过代码控制业务)。

---

# 有几种不同类型的自动代理？

BeanNameAutoProxyCreator：bean 名称自动代理创建器
DefaultAdvisorAutoProxyCreator：默认通知者自动代理创建器 Metadata autoproxying：元数据自动代理

---

# 什么是织入？什么是织入应用的不同点？

织入是将切面和其他应用类型或对象连接起来创建一个通知对象的过程。织入可以在编译、加载或运行时完成。

---

# 什么是 Spring 的 MVC 框架？

Spring 提供了一个功能齐全的 MVC 框架用于构建 Web 应用程序。Spring 框架可以很容易的和其他的 MVC 框架融合(如 Struts)，该框架使用控制反转(IOC)将控制器逻辑和业务对象分离开来。它也允许以声明的方式绑定请求参数到业务对象上。

---

# DispatcherServlet

Spring 的 MVC 框架围绕 DispatcherServlet 来设计的，它用来处理所有的 HTTP 请求和响应。

---

# WebApplicationContext

WebApplicationContext 继承了 ApplicationContext，并添加了一些 web 应用程序需要的功能。和普通的 ApplicationContext 不同，WebApplicationContext 可以用来处理主题样式，它也知道如何找到相应的 servlet。

---

# 什么是 Spring MVC 框架的控制器？

控制器提供对应用程序行为的访问，通常通过服务接口实现。控制器解析用户的输入，并将其转换为一个由视图呈现给用户的模型。Spring 通过一种极其抽象的方式实现控制器，它允许用户创建多种类型的控制器。

@Controller annotation

@Controller 注解表示该类扮演控制器的角色。Spring 不需要继承任何控制器基类或应用 Servlet API。

@RequestMapping annotation

@RequestMapping 注解用于将 URL 映射到任何一个类或者一个特定的处理方法上。