Spring的优势

- **方便解耦，简化开发**

  通过Spring提供的IoC容器，可以将对象间的依赖关系交由Spring进行控制，避免硬编码所造成的 过度程序耦合。用户也不必再为单例模式类、属性文件解析等这些很底层的需求编写代码，可以更 专注于上层的应用。

- **AOP编程的支持**

  通过Spring的AOP功能，方便进行面向切面的编程，许多不容易用传统OOP实现的功能可以通过 AOP轻松应付。

- **声明式事务的支持**

  @Transactional

  可以将我们从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活的进行事务的管理，提高开发效率和质量。

- **方便程序的测试**

  可以用非容器依赖的编程方式进行几乎所有的测试工作，测试不再是昂贵的操作，而是随手可做的事情。

- **集成各种优秀框架**

  Spring可以降低各种框架的使用难度，提供了对各种优秀框架（Struts、Hibernate、Hessian、 Quartz等）的直接支持。

- **降低JavaEE API的使用难度**

  Spring对JavaEE API（如JDBC、JavaMail、远程调用等）进行了薄薄的封装层，使这些API的使用难度大为降低。

- **源码是经典的 Java 学习范例**

  Spring的源代码设计精妙、结构清晰、匠心独用，处处体现着大师对Java设计模式灵活运用以及对Java技术的高深造诣。它的源代码无意是Java技术的最佳实践的范例。

Spring的核心思想

核心思想是IOC和AOP，OC和AOP不是Spring提出的，在Spring之前就已经存在，只不过更偏向于理论化，Spring在技术层次把这两个思想做了非常好的实现（Java）。

什么是IOC

IOC简介：

IOC Inversion of Control (控制反转/反转控制)，注意它是一个技术思想，不是一个技术实现

描述的事情：Java开发领域对象的创建，管理的问题

传统开发方式：比如类A依赖于类B，往往会在类A中new一个B的对象

IOC思想下开发方式：我们不用自己去new对象了，而是由IOC容器（Spring框架）去帮助我们实例化对象并且管理它，我们需要使用哪个对象，去问IOC容器要即可

我们丧失了一个权利（创建、管理对象的权利）,得到了一个福利（不用考虑对象的创建、管理等一系列事情）

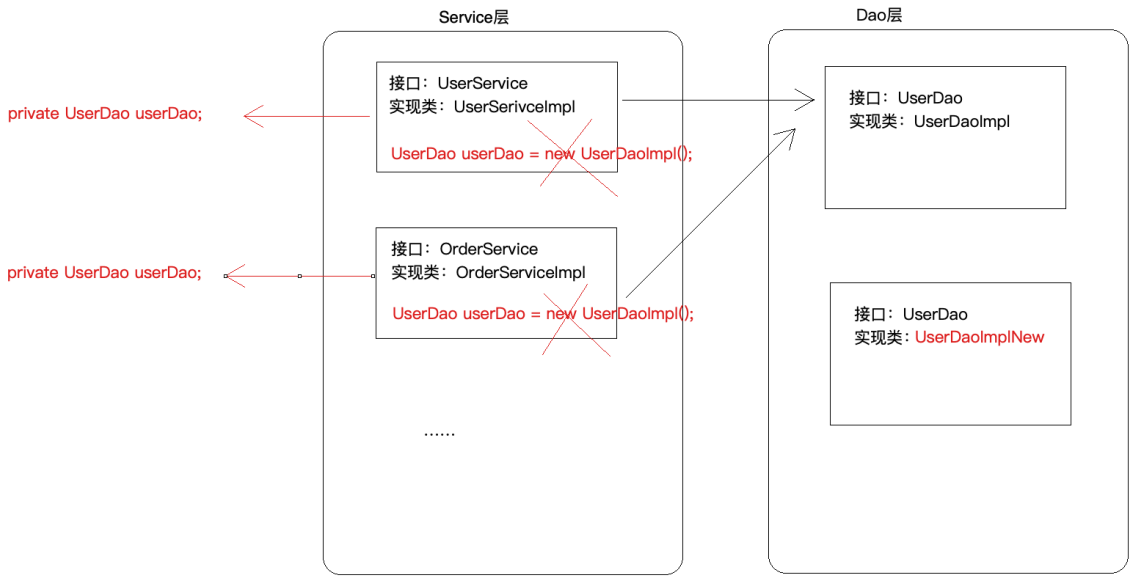
为什么叫做控制反转?

控制：指的是对象创建（实例化、管理）的权利

反转：控制权交给外部环境了（Spring框架、IoC容器）

IoC解决了什么问题?：

IoC解决对象之间的耦合问题

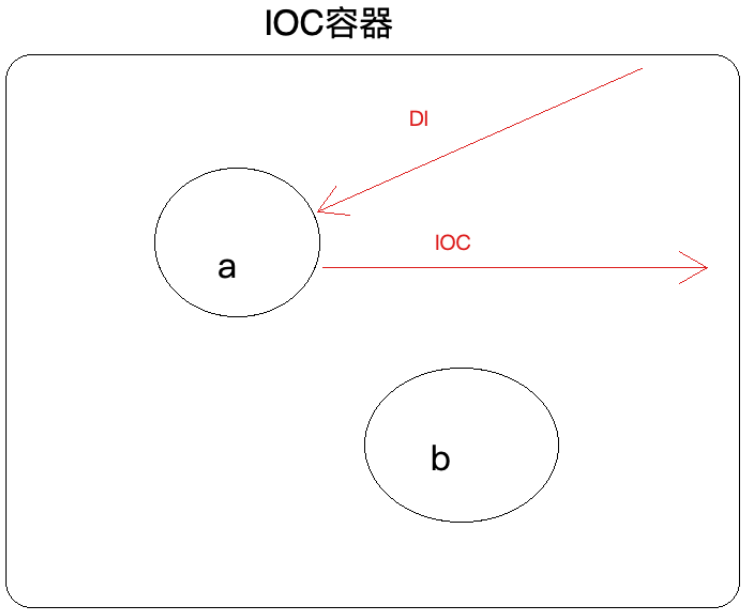


IoC和DI的区别

DI：Dependancy Injection（依赖注入）

怎么理解：

IOC和DI描述的是同一件事情，只不过角度不一样罢了



IOC和DI描述的是同一件事情（对象实例化及依赖关系维护这件事情），只不过角度不同罢了

IOC是站在对象的角度，对象实例化及其管理的权利交给了（反转）给了容器

DI是站在容器的角度，

容器会把对象依赖的其他对象注入（送进去），比如A对象实例化过程中因为声明了一个B类型的属性，那么就需要容器把B对象注入给A

2.

BeanFactory与ApplicationContext区别

BeanFactory是Spring框架中IoC容器的顶层接口,它只是用来定义一些基础功能,定义一些基础规范,而ApplicationContext是它的一个子接口，所以ApplicationContext是具备BeanFactory提供的全部功能的。
通常，我们称BeanFactory为SpringIOC的基础容器，ApplicationContext是容器的高级接口，比BeanFactory要拥有更多的功能，比如说国际化支持、资源访问（xml，java配置类）>等等

Bean的定义、加载、实例化，依赖注入和生命周期管理。ApplicationContext接口作为BeanFactory的子类，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

- 继承MessageSource，因此支持国际化。
- 资源文件访问，如URL和文件（ResourceLoader）。
- 载入多个（有继承关系）上下文（即同时加载多个配置文件），使得每一个上下文都专注于一个特定的层次，比如应用的web层。
- 提供在监听器中注册bean的事件。

Java环境下启动IoC容器

- ClassPathXmlApplicationContext：从类的根路径下加载配置文件（推荐使用）
- FileSystemXmlApplicationContext：从磁盘路径上加载配置文件
- AnnotationConfigApplicationContext：纯注解模式下启动Spring容器

实例化Bean的三种方式

方式一：使用无参构造函数（推荐，如果使用xml配置很麻烦可以使用2、3创建对象再放入容器中）

在默认情况下，它会通过反射调用无参构造函数来创建对象。如果类中没有无参构造函数，将创建失败。

```
<!--配置service对象-->
<bean id="userService" class="com.lagou.service.impl.TransferServiceImpl"></bean>
```

方式二：使用静态方法创建

在实际开发中，我们使用的对象有些时候并不是直接通过构造函数就可以创建出来的，它可能在创 建的过程 中会做很多额外的操作。此时会提供一个创建对象的方法，恰好这个方法是static修饰的方法，即是此种情况。

例如，我们在做Jdbc操作时，会用到java.sql.Connection接口的实现类，如果是mysql数据库，那 么用的就 是JDBC4Connection，但是我们不会去写 JDBC4Connection connection = new JDBC4Connection() ，因为我们要注册驱动，还要提供URL和凭证信息， 用 DriverManager.getConnection 方法来获取连接。

contextConfigLocation com.lagou.edu.SpringConfig

org.Springframework.web.context.ContextLoaderListener

那么在实际开发中，尤其早期的项目没有使用Spring框架来管理对象的创建，但是在设计时使用了工厂模式解耦，那么当接入Spring之后，工厂类创建对象就具有和上述例子相同特征，即可采用此种方式配置。

```
<!--使用静态方法创建对象的配置方式-->
<bean id="userService" class="com.lagou.factory.CreateBeanFactory"
 factory-method="getTransferService"></bean>

public class CreateBeanFactory {
 public static ConnectionUtils getTransferService() {
        return new ConnectionUtils();
    }
}
```

方式三：使用实例化方法创建

此种方式和上面静态方法创建其实类似，区别是用于获取对象的方法不再是static修饰的了，而是类中的一个普通方法。此种方式比静态方法创建的使用几率要高一些。

在早期开发的项目中，工厂类中的方法有可能是静态的，也有可能是非静态方法，当是非静态方法时，即可采用下面的配置方式：

```
<!--使用实例方法创建对象的配置方式-->
<bean id="createBeanFactory"
class="com.lagou.factory.instancemethod.CreateBeanFactory"></bean>
<bean id="transferService" factory-bean="createBeanFactory"
factorymethod="getInstance"></bean>

public class CreateBeanFactory {
  public ConnectionUtils getInstance() {
        return new ConnectionUtils();
    }
}
```

注入三种方式

三种注入方式的比较

- 接口注入。从注入方式的使用上来说，接口注入是现在不甚提倡的一种方式，基本处于“退役状态”。**因为它强制被注入对象实现不必要的接口，带有侵入性。**而构造方法注入和setter 方法注入则不需要如此。
- 构造方法注入。这种注入方式的优点就是，对象在构造完成之后，即已进入就绪状态，可以马上使用。缺点就是，当依赖对象比较多的时候，构造方法的参数列表会比较长。而通过反射构造对象的时候，**对相同类型的参数的处理会比较困难**，维护和使用上也比较麻烦。而且在Java中，构造方法无法被继承，无法设置默认值。对于非必须的依赖处理，可能需要引入多个构造方法，而参数数量的变动可能造成维护上的不便。
- setter方法注入。因为方法可以命名，所以setter方法注入在描述性上要比构造方法注入好一些。 另外，setter方法可以被继承，允许设置默认值，而且有良好的IDE支持，缺点当然就是对象无法在构造完成后马上进入就绪状态，**不能将对象设为final。**

  set方法注入和constructor注入

  set方法注入

```
<property name="name" value="zhangsan"/>
<property name="sex" value="1"/>
<property name="money" value="100.3"/>

private String name;
private int sex;
private float money;
public void setName(String name) {this.name = name;}
public void setSex(int sex) {this.sex = sex;}
public void setMoney(float money) {this.money = money;}
```

constructor注入

```
public JdbcAccountDaoImpl(String name, int sex, float money) {
        this.connectionUtils = connectionUtils;
        this.name = name;
        this.sex = sex;
        this.money = money;
}
<constructor-arg name="name" value="zhangsan"/>
<constructor-arg name="sex" value="1"/>
<constructor-arg name="money" value="100.6"/>
```

Bean的作用范围

singleton （单例模式）：

对象出生：当创建容器时，对象就被创建了。

对象活着：只要容器在，对象一直活着。

对象死亡：当销毁容器时，对象就被销毁了。

一句话总结：单例模式的bean对象生命周期与容器相同。

prototype（多例模式）：

对象出生：当使用对象时，创建新的对象实例。

对象活着：只要对象在使用中，就一直活着。

对象死亡：当对象长时间不用时，被java的垃圾回收器回收了。

一句话总结：多例模式的bean对象，Spring框架只负责创建，不负责销毁。

request：

每次HTTP请求都会创建一个新的Bean，作用域仅适用于WebApplicationContext。

session：

同一个HTTP Session共享一个Bean，不同Session使用不同Bean，仅适用于WebApplicationContext环境。

globalsession：

在一个全局的HTTP Session中，仅适用于WebApplicationContext环境。

spring的bean回收时机：

Spring 中的Bean默认是singleton
singleton（全局的）是随着spring的存亡而存亡
GC回收原则，当bean的引用没有指向任何地方的时候，它就会被回收
spring中的singleton 存在于ioc 中，本身就是单例，是基于spring的上下文的，当spring本身不消失，自然ioc容器也不会消失，自然ioc容器中的引用也会一直被持有，那么自然spring中的bean也就不会被回收会一直存在

- prototype 又叫多例模式，用的时候就new一下，用完就没有了。
- session 存在这一次会话 session 中，session没有过期它就一直存在，session过期后它就没了。
- request 存在这一次请求中，请求结束了它就结束。

Spring框架中的单例bean是线程安全的吗?

不，Spring框架中的单例bean不是线程安全的。

Bean标签属性

在基于xml的IoC配置中，bean标签是最基础的标签。它表示了IoC容器中的一个对象。换句话说，如果一个对象想让Spring管理，在XML的配置中都需要使用此标签配置，Bean标签的属性如下：

id属性： 用于给bean提供一个唯一标识。在一个标签内部，标识必须唯一。

class属性：用于指定创建Bean对象的全限定类名。

name属性：用于给bean提供一个或多个名称。多个名称用空格分隔。

factory-bean属性：用于指定创建当前bean对象的工厂bean的唯一标识。当指定了此属性之后， class属性失效。

factory-method属性：用于指定创建当前bean对象的工厂方法，如配合factory-bean属性使用， 则class属性失效。如配合class属性使用，则方法必须是static的。

scope属性：用于指定bean对象的作用范围。通常情况下就是singleton。当要用到多例模式时， 可以配置为prototype。

init-method属性：用于指定bean对象的初始化方法，此方法会在bean对象装配后调用。必须是一个无参方法。

destory-method属性：用于指定bean对象的销毁方法，此方法会在bean对象销毁前执行。它只能为scope是singleton时起作用。

lazy-Init 延迟加载

```
<bean id="testBean" class="cn.lagou.LazyBean" />
该bean默认的设置为:
<bean id="testBean" calss="cn.lagou.LazyBean" lazy-init="false" />
```

设置了延迟加载，只有使用时才会实例化bean。

应用场景

（1）开启延迟加载一定程度提高容器启动和运转性能

（2）对于不常使用的 Bean 设置延迟加载，这样偶尔使用的时候再加载，不必要从一开始该 Bean 就占用资源

Spring后置处理器

Spring的BeanPostProcessor应用：

例：自定义注解@RountingInjected，在初始化后将接口的具体实现类注入进去。

```
@Component
public class HelloServiceTest {

    @RountingInjected(value = "helloServiceImpl2")//自定义的注解，在postProcessAfterInitialization方法就把实现类注入进去了。
    private HelloService helloService;

    public void testSayHello() {
        helloService.sayHello();
    }

    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext("colin.Spring.basic.advanced.bbp");
        HelloServiceTest helloServiceTest = applicationContext.getBean(HelloServiceTest.class);
        helloServiceTest.testSayHello();
    }
```

```
 @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        Class<?> targetCls = bean.getClass();
        Field[] targetFld = targetCls.getDeclaredFields();
        for (Field field : targetFld) {
            //找到制定目标的注解类
            if (field.isAnnotationPresent(RountingInjected.class)) {
                if (!field.getType().isInterface()) {
                    throw new BeanCreationException("RoutingInjected field must be declared as an interface:" + field.getName()
                            + " @Class " + targetCls.getName());
                }
                try {
                    this.handleRoutingInjected(field, bean, field.getType());
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
```

```
            }
        }
        return bean;
    }
```

Spring后置处理器介绍:

Spring提供了两种后处理bean的扩展接口，分别为 BeanPostProcessor 和 BeanFactoryPostProcessor，两者在使用上是有所区别的,我们定义一个类实现了BeanPostProcessor，默认是会对整个Spring容器中所有的bean进行处理。既然是默认全部处理，那么我们怎么确认我们需要处理的某个具体的bean呢？可以看到方法中有两个参数。类型分别为Object和String，第一个参数是每个bean的实例，第二个参数是每个bean的name或者id属性的值。所以我们可以第二个参数，来确认我们将要处理的具体的bean。

BeanPostProcessor 和 BeanFactoryPostProcessor的区别:

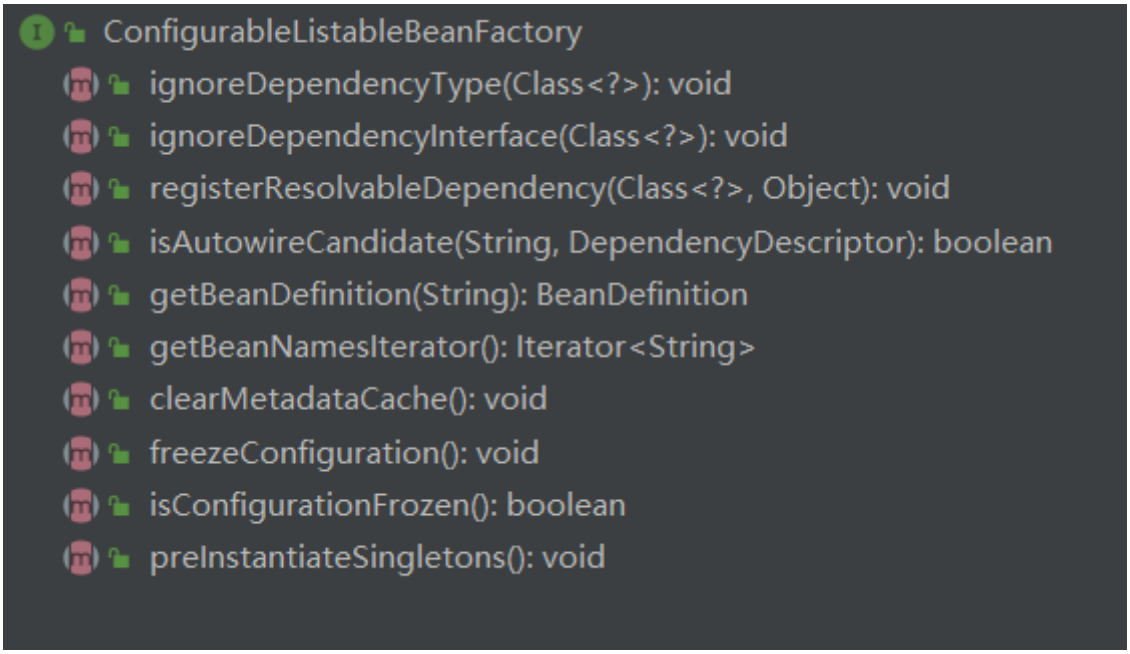
BeanPostProcessorr是针对Bean级别的处理，可以针对某个具体的Bean初始化前后扩展方法。

```
public interface BeanPostProcessor {
    @Nullable
    default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }

    @Nullable
    default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }
}
```

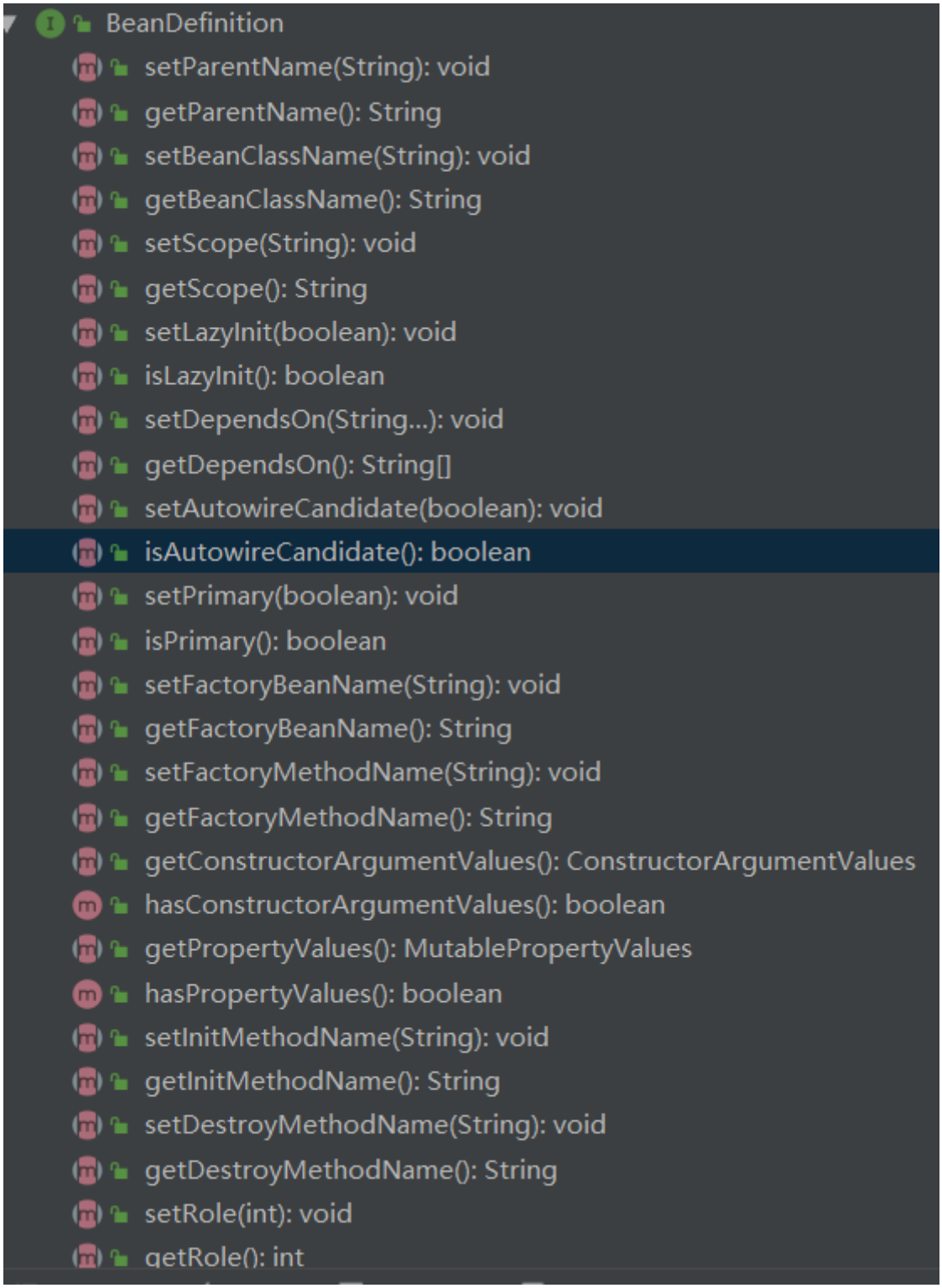
BeanFactoryPostProcessor BeanFactory级别的处理，是针对整个Bean的工厂进行处理，可以找到某个Bean，然后使用BeanDefinition对象的方法对bean的属性进行修改，**调用 BeanFactoryPostProcessor 方法时，这时候bean还没有实例化，此时 bean 刚被解析成 BeanDefinition对象。**

```
@FunctionalInterface
public interface BeanFactoryPostProcessor {
    void postProcessBeanFactory(ConfigurableListableBeanFactory var1) throws BeansException;
}
```

此接口只提供了一个方法，方法参数为ConfigurableListableBeanFactory，该参数类型定义了一些方法



其中有个方法名为getBeanDefinition的方法，我们可以根据此方法，找到我们定义bean 的 BeanDefinition 对象。然后我们可以对定义的属性进行修改，以下是BeanDefinition中的方法。



方法名字类似我们bean标签的属性，setBeanClassName对应bean标签中的class属性，所以当我们拿到BeanDefinition对象时，我们可以手动修改bean标签中所定义的属性值。 BeanDefinition对象：我们在 XML 中定义的 bean标签，Spring 解析 bean 标签成为一个 JavaBean， 这个JavaBean 就是 BeanDefinition

BeanFactoryPostProcessor典型应用:PropertyPlaceholderConfigurer：

```
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
    try {
        Properties mergedProps = this.mergeProperties();
        this.convertProperties(mergedProps);
        this.processProperties(beanFactory, mergedProps);
    } catch (IOException var3) {
        throw new BeanInitializationException("Could not load properties", var3);
    }
}
```

mergeProperties（）合并属性信息（后面的覆盖前面的），processProperties（）拦截BeanDefinition对象，将相应合并属性赋值给xml文件中。

例：有两个properties文件命名不同数据源信息合并，那么这个方法会以最后拿到的数据源信息为准，再赋值给bean。

总结：

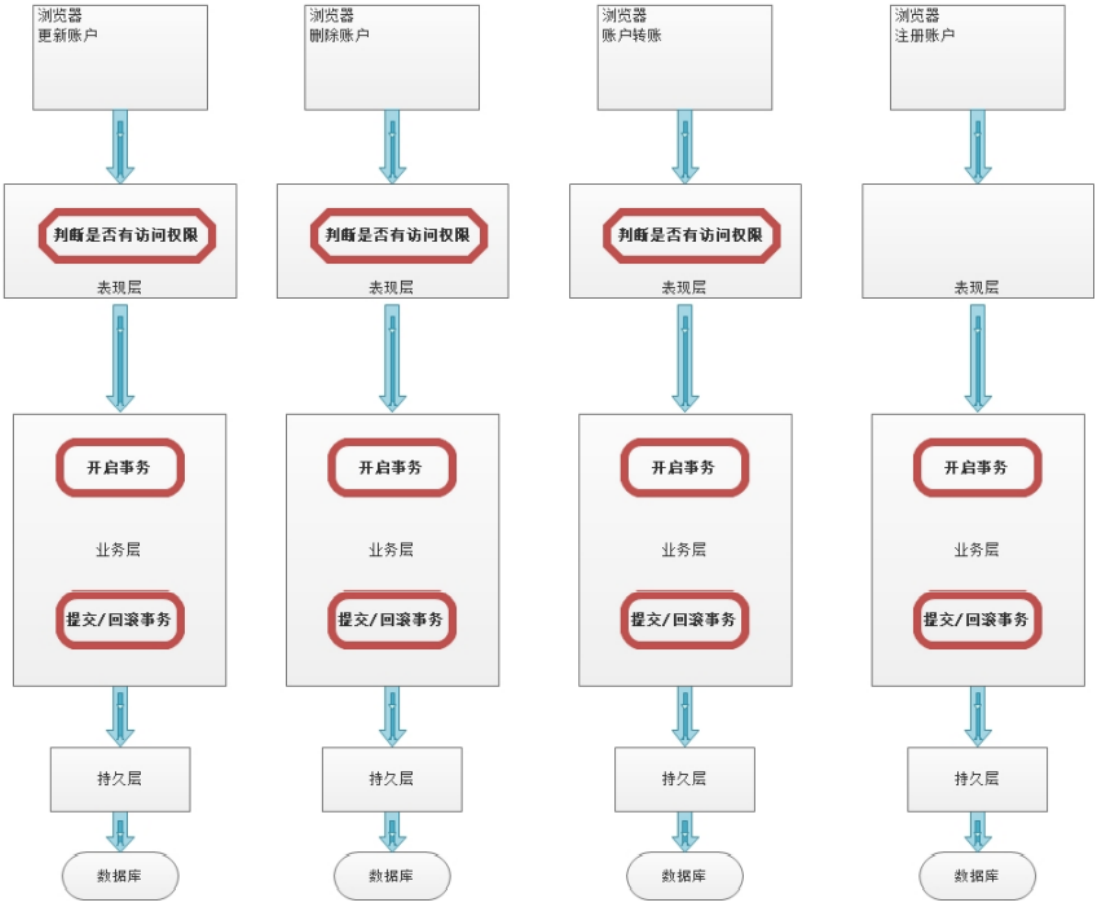
BeanPostProcessor 和 BeanFactoryPostProcessor的区别：

1. BeanPostProcessor 是bean对象级别的操作，只允许操作一个bean，BeanFactoryPostProcessor是bean工厂级别的操作，可以操作多个bean。
2. BeanPostProcessor 是在bean实例化后，初始化前后调用的，BeanFactoryPostProcessor是在bean实例化前，刚被解析成 BeanDefinition对象时调用的。

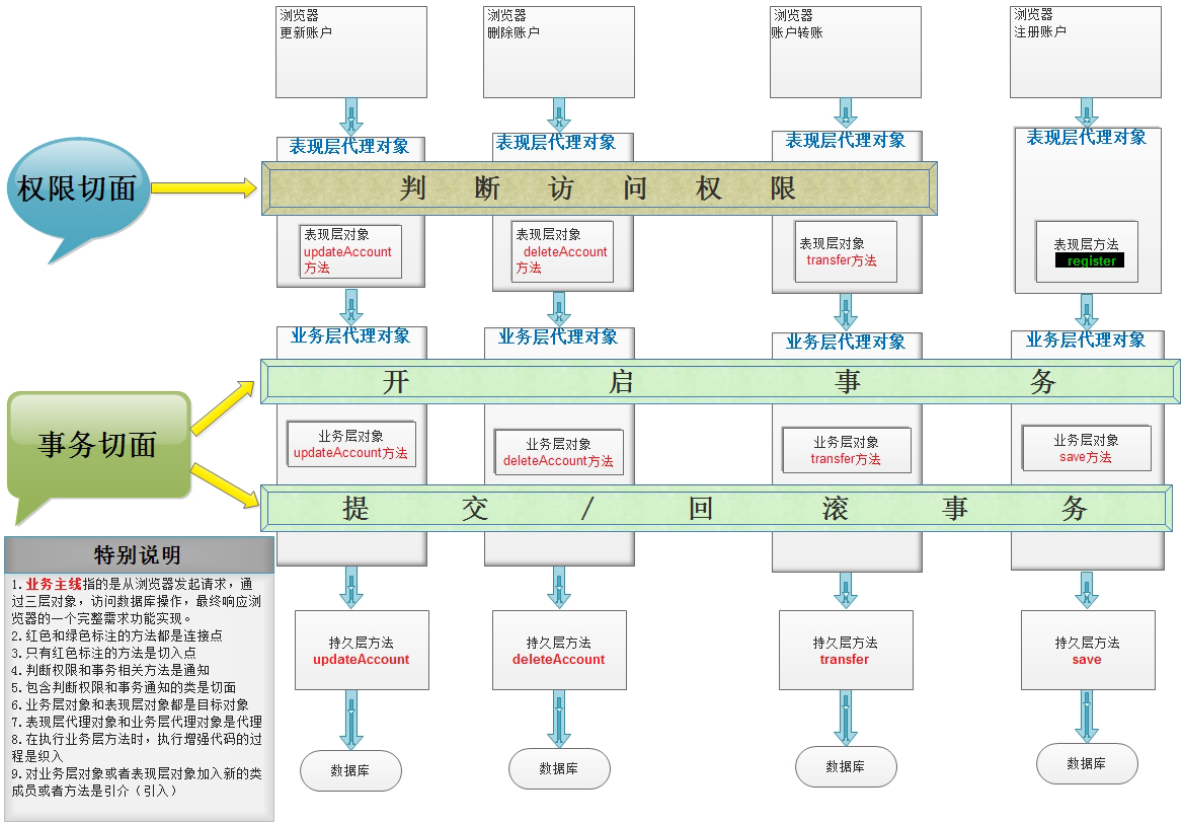
Aop介绍

AOP本质：

在不改变原有业务逻辑的情况下增强横切逻辑，通常使用于权限校验代码、日志代码、事务控制代码、性能监控代码。



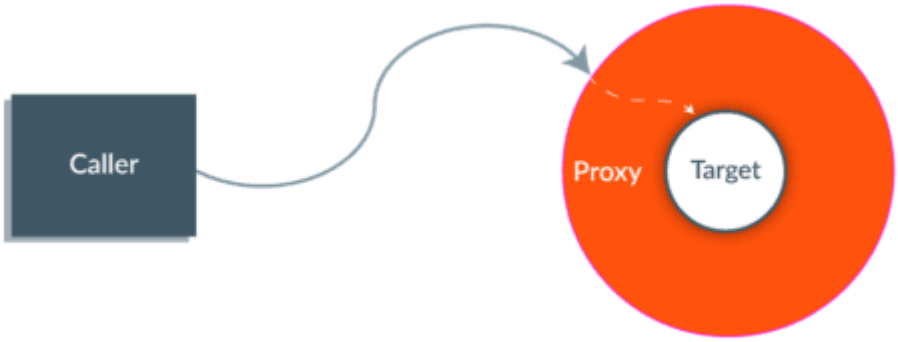
上图描述的就是未采用AOP思想设计的程序，当我们红色框中圈定的方法时，会带来大量的重复劳动。程序中充斥着大量的重复代码，使我们程序的独立性很差。而下图中是采用了AOP思想设计的程序，它把红框部分的代码抽取出来的同时，运用动态代理技术，在运行期对需要使用的业务逻辑方法进行增强。



什么是AOP

AOP术语：

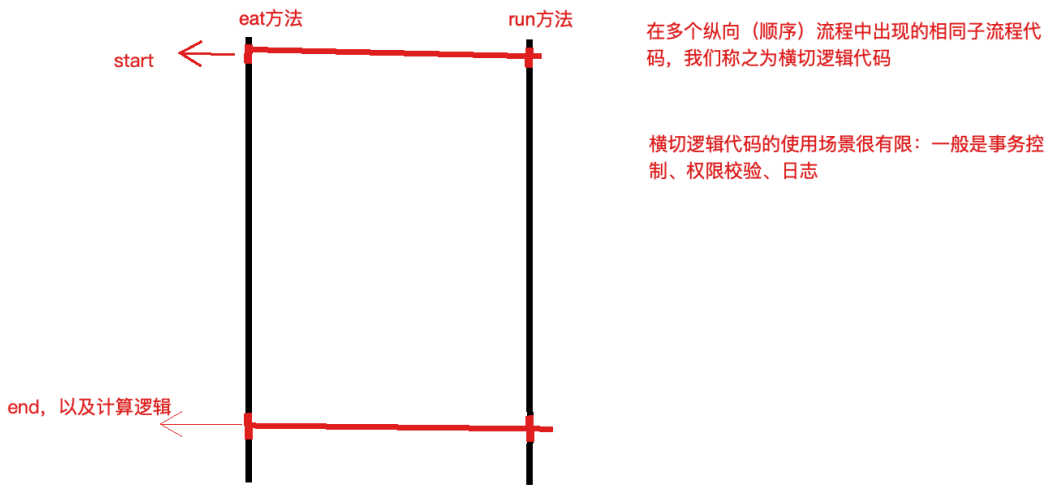
名词	解释
Joinpoint(连接点)	程序运行中的一些时间点, 例如一个方法的执行, 或者是一个异常的处理
Pointcut(切入点)	
Advice(通知/增强)	特定 JoinPoint 处的 Aspect 所采取的动作称为 Advice
Target(目标对象)	它指的是代理的目标对象。即被代理对象
Proxy(代理)	代理是通知目标对象后创建的对象。从客户端的角度看，代理对象和目标对象是一样的。
Weaving(织入)	为了创建一个 advice 对象而链接一个 aspect 和其它应用类型或对象，称为编织（Weaving），如下图
Aspect(切面)	使用 @Aspect 注解的类就是切面



AOP示例：

```java
 * @author 应癫
 */
public class Animal {

    private String height; // 高度
    private float weight;  // 体重

    public void eat(){

        // 性能监控代码
        long start = System.currentTimeMillis();

        // 业务逻辑代码
        System.out.println("I can eat...");

        // 性能监控代码
        long end = System.currentTimeMillis();
        System.out.println("执行时长: " + (end-start)/1000f + "s");
    }

    public void run(){
        // 性能监控代码
        long start = System.currentTimeMillis();

        // 业务逻辑代码
        System.out.println("I can run...");

        // 性能监控代码
        long end = System.currentTimeMillis();
        System.out.println("执行时长: " + (end-start)/1000f + "s");
    }

}
```

横切逻辑代码：



AOP在解决什么问题

在不改变原有业务逻辑情况下，增强横切逻辑代码，根本上解耦合，避免横切逻辑代码重复

Spring中AOP的代理选择：

Spring实现AOP思想使用的是动态代理技术

代理对象没有实现接口选择CGLIB代理，实现了接口选择JDK代理，也可用配置方式强制使用CGLIB代理。

AOP的XML实现：

配置文件：

```
<!--把通知bean交给Spring来管理-->
<bean id="logUtil" class="com.lagou.utils.LogUtil"></bean>
<!--开始aop的配置-->
<aop:config>
    <!--配置切面-->
     <aop:aspect id="logAdvice" ref="logUtil">
     <!--配置前置通知-->
     <aop:before method="printLog" pointcut="execution(public *
    com.lagou.service.impl.TransferServiceImpl.updateAccountByCardNo(com.lagou
    .pojo.Account))"></aop:before>
     <!--配置后置通知-->
     <aop:before method="printLog" pointcut-ref="pointcut1"></aop:before>
     <!--配置正常执行时通知-->
     <aop:after-returning method="afterReturningPrintLog" pointcutref="pt1">
</aop:after-returning>
     <!--配置异常通知-->
     <aop:after-throwing method="afterThrowingPrintLog" pointcut-ref="pt1">
</aop:after-throwing>
     <!--配置最终通知-->
     <aop:after method="afterPrintLog" pointcut-ref="pt1"></aop:after>
     <!--配置环绕通知-->
     <aop:around method="aroundPrintLog" pointcut-ref="pt1"></aop:around>
    </aop:aspect>
</aop:config>
```

通知的内部逻辑：

```
@Component
@Aspect
public class LogUtils {
    /**
     * 业务逻辑开始之前执行
     */
    public void beforeMethod(JoinPoint joinPoint) {
        Object[] args = joinPoint.getArgs();
        for (int i = 0; i < args.length; i++) {
            Object arg = args[i];
            System.out.println(arg);
        }
        System.out.println("业务逻辑开始执行之前执行.......");
    }

    /**
     * 业务逻辑结束时执行（无论异常与否）
     */
    public void afterMethod() {
        System.out.println("业务逻辑结束时执行，无论异常与否都执行.......");
    }

    /**
     * 异常时时执行
     */
    public void exceptionMethod() {
        System.out.println("异常时执行.......");
    }

    /**
     * 业务逻辑正常时执行
```

```
     */
    public void successMethod(Object retVal) {
        System.out.println("业务逻辑正常时执行.......");
    }

    /**
     * 环绕通知
     *
     */
    public Object arroundMethod(ProceedingJoinPoint proceedingJoinPoint) throws
Throwable       {
        System.out.println("环绕通知中的beforemethod....");

        Object result = null;
        try{
            // 控制原有业务逻辑是否执行
            // result =
proceedingJoinPoint.proceed(proceedingJoinPoint.getArgs());
        }catch(Exception e) {
            System.out.println("环绕通知中的exceptionmethod....");
        }finally {
            System.out.println("环绕通知中的after method....");
        }
        return result;
    }
}

//环绕通知：
    //通过proceedingJoinPoint.proceed(proceedingJoinPoint.getArgs())是否存在决定被代
理对象是否执行，存在的话在代码前后加逻辑或加try catch（）也可以实现其他通知一样的效果。
```

AOP改变代理方式的配置：

有两种：

- 使用aop:config标签配置

```
<aop:config proxy-target-class="true">
```

- 使用aop:aspectj-autoproxy标签配置

```
<!--此标签是基于XML和注解组合配置AOP时的必备标签，表示Spring开启注解配置AOP的支持-->
<aop:aspectj-autoproxy proxy-target-class="true" />
```

AOP注解实现：

```java
@Aspect
public class CheckUser {

    @Pointcut("execution(* com.tgb.spring.aop.*.find*(..))")
    public void checkUser(){
        System.out.println("**************The System is Searching Information For You***************");
    }

    @Before("checkUser()")
    public void beforeCheck() { System.out.println(">>>>>>>> 准备搜查用户.........."); }

    @After("checkUser()")
    public void afterCheck() { System.out.println(">>>>>>>>  搜查用户完毕.........."); }

    //声明环绕通知
    @Around("checkUser()")
    public Object doAround(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("进入方法---环绕通知");
        Object o = pjp.proceed();
        System.out.println("退出方法---环绕通知");
        return o;
    }
}
```

该方法通过表达式制定切入方法

添加切入方法的前置通知

添加切入方法的后置通知

添加切入方法的环绕通知

Spring事务介绍

编程式事务：在业务代码中添加事务控制代码，这样的事务控制机制就叫做编程式事务

声明式事务：通过xml或者注解配置的方式达到事务控制的目的，叫做声明式事务

事务的四大特性：

原子性（Atomicity）

原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。 从操作的角度来描述，事务中的各个操作要么都成功要么都失败

一致性（Consistency）

事务必须使数据库从一个一致性状态变换到另外一个一致性状态。

例如转账前A有1000，B有1000。转账后A+B也得是2000。 一致性是从数据的角度来说的，（1000，1000） （900，1100），不应该出现（900，1000）

隔离性（Isolation）

事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务， 每个事务不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。

比如：事务1给员工涨工资2000，但是事务1尚未被提交，员工发起事务2查询工资，发现工资涨了2000 块钱，读到了事务1尚未提交的数据（脏读）

持久性（Durability）

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障 也不应该对其有任何影响。

事务的隔离级别和传播行为级别介绍：

先介绍脏读、不可重复读、幻读（虚读）：

脏读：

一个线程中的事务读到了另外一个线程中未提交的数据。

不可重复读：

一个线程中的事务读到了另外一个线程中已经提交的update的数据（前后**内容**不一样）

场景： 员工A发起事务1，查询工资，工资为1w，此时事务1尚未关闭财务人员发起了事务2，给员工A张了2000块钱，并且提交了事务员工A通过事务1再次发起查询请求，发现工资为1.2w，原来读出来1w读不到了，叫做不可重复读

幻读（虚读）：

一个线程中的事务读到了另外一个线程中已经提交的insert或者delete的数据（前后**条数**不一样）

场景： 事务1查询所有工资为1w的员工的总数，查询出来了10个人，此时事务尚未关闭，事务2财务人员发起新来员工，工资1w，向表中插入了2条数据，并且提交了事务，事务1再次查询工资为1w的员工个数，发现有12个人，见了鬼了。

数据库共定义了四种隔离级别：

事务隔离级别	回滚覆盖	脏读	不可重复读	提交覆盖	幻读
读未提交	x	可能发生	可能发生	可能发生	可能发生
读已提交	x	x	可能发生	可能发生	可能发生
可重复读	x	x	x	x	可能发生
串行化	x	x	x	x	x

1. Serializable（串行化）：可避免**脏读、不可重复读、幻读** 情况的发生。（串行化） 最高
2. Repeatable READ（可重复读）：可避免**脏读、不可重复读**情况的发生。(幻读有可能发生) 第二该机制下会对要update的行进行加锁
3. Read Committed（读已提交）：可避免**脏读**情况发生。不可重复读和幻读一定会发生。
4. Read uncommitted（读未提交）：最低级别，以上情况均无法保证。(读未提交) 最低
5. 回滚覆盖：事务1的修改提交后，事务2回滚即使最低级别的读未提交也不会将事务1提交的修改操作回滚
6. 提交覆盖：事务1的修改提交后，有可能把事务2的事务给覆盖。

   注意：级别依次升高，效率依次降低

   MySQL的默认隔离级别是：可重复读

   Oracle、SQLServer默认隔离级别：读已提交

   Oracle的事务隔离级别有两种

   - READ COMMITTED：读已提交

- SERIALIZABLE：串行读取

MySQL的事务隔离级别有四种

- 读未提交 (Read uncommitted)
- 读已提交 (Read committed)
- 可重复读 (Repeated read)
- 可串行化 (Serializable)

MySQL默认隔离级别为：可重复读 (Repeated read)

> 一般使用时，建议采用默认隔离级别，然后存在的一些并发问题，可以通过悲观锁、乐观锁等实现处 理。

查询当前使用的隔离级别： select @@tx_isolation;

sql语句设置MySQL事务的隔离级别（设置的是当前 mysql连接会话的，并不是永久改变的）：

```
//设置read uncommitted级别：
set session transaction isolation level read uncommitted;

//设置read committed级别：
set session transaction isolation level read committed;

//设置repeatable read级别：
set session transaction isolation level repeatable read;

//设置serializable级别：
set session transaction isolation level serializable;
```

事务隔离级别实现的原理：

Serializable（串行化）：相当于单线程执行。

Repeatable READ（可重复读）、Read Committed（读已提交）：这两种隔离级别通过MVCC、行锁、间隙锁实现。

Read uncommitted（读未提交）：不用加锁限制。

事务的传播行为介绍：

事务往往在service层进行控制，如果出现service层方法A调用了另外一个service层方法B，A和B方法本身都已经被添加了事务控制，那么A调用B的时候，就需要进行事务的一些协商，这就叫做事务的传播行为。

A调用B，我们站在B的角度来观察来定义事务的传播行为种类

种类	描述
REQUIRED	如果当前没有事务，就新建一个事务，如果已经存在一个事务中， 加入到这个事务中。这是最常见的选择。
SUPPORTS	支持当前事务，如果当前没有事务，就以非事务方式执行。
MANDATORY	使用当前的事务，如果当前没有事务，就抛出异常。
REQUIRES_NEW	新建事务，如果当前存在事务，把当前事务挂起。
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。
NEVER	以非事务方式执行，如果当前存在事务，则抛出异常。
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，则执行与PROPAGATION_REQUIRED类似的操作。

Spring中事务的API：

PlatformTransactionManager：

该接口是DataSourceTransactionManager （mysql事务管理器），HibernateTransactionManager （Hibernate事务管理器）都实现，目的是统一事务管理。

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(@Nullable TransactionDefinition var1) throws TransactionException;

    void commit(TransactionStatus var1) throws TransactionException;

    void rollback(TransactionStatus var1) throws TransactionException;
}
```

DataSourceTransactionManager 归根结底是横切逻辑代码，声明式事务要做的就是使用Aop（动态代 理）来将事务控制逻辑织入到业务代码

事务的配置：

纯xml模式：

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
     <tx:attributes>
         name：针对某个方法添加事务
         read-only：事务是否只读
         propagation：事务传播行为
         isolation：事务隔离级别（DEFAULT为默认级别REPEATABLE READ ）
         timeout：设置超时时间（-1代表不限制超时时间）
         rollback-for：发生该异常回滚（默认异常都回滚）
         no-rollback-for：发生该异常不回滚
        <tx:method name="query*" read-only="true" propagation="SUPPORTS" isolation="DEFAULT" timeout="-1" rollback-for="" no-rollback-for=""/>
    </tx:attributes>
 </tx:advice>

<aop:config>
    指定类加入事务（advice-ref应用上方事务）
    <aop:advisor advice-ref="txAdvice" pointcut="execution(*
    com.lagou.edu.service.impl.TransferServiceImpl.*(..))"/>
 </aop:config>
```

基于XML+注解：

- xml配置

```
<!--配置事务管理器-->
<bean id="transactionManager"
class="org.Springframework.jdbc.datasource.DataSourceTransactionManager">

 <property name="dataSource" ref="dataSource"></property>
</bean>
<!--开启Spring对注解事务的支持-->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

- 在接口、类或者方法上添加@Transactional注解

```
@Transactional(readOnly = true,propagation = Propagation.SUPPORTS)
```

基于纯注解：

在 Spring 的配置类上添加 @EnableTransactionManagement 注解即可

```
@EnableTransactionManagement//开启Spring注解事务的支持
public class SpringConfiguration {
}
//在接口、类或者方法上添加@Transactional注解
@Transactional(readOnly = true,propagation = Propagation.SUPPORTS)
```

@Transactional注解失效的场景：

1. @Transactional 应用在非 public 修饰的方法上

如果Transactional注解应用在非public 修饰的方法上，Transactional将会失效。

之所以会失效是因为在Spring AOP 代理时，如上图所示 TransactionInterceptor （事务拦截器）在目标方法执行前后进行拦截，DynamicAdvisedInterceptor（CglibAopProxy 的内部类）的 intercept 方法或 JdkDynamicAopProxy 的 invoke 方法会间接调用 AbstractFallbackTransactionAttributeSource 的 computeTransactionAttribute 方法，获取Transactional 注解的事务配置信息。

```
protected TransactionAttribute computeTransactionAttribute(Method method,
    Class<?> targetClass) {
        // Don't allow no-public methods as required.
        if (allowPublicMethodsOnly() &&
!Modifier.isPublic(method.getModifiers())) {
        return null;
}
```

此方法会检查目标方法的修饰符是否为 public，不是 public则不会获取@Transactional 的属性配置信息。

注意：protected、private 修饰的方法上使用 @Transactional 注解，虽然事务无效，但不会有任何报错，这是我们很容犯错的一点。

2. @Transactional 注解属性 propagation （传播行为）设置错误

这种失效是由于配置错误，若是错误的配置以下三种 propagation，事务将不会发生回滚。

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。

3. @Transactional 注解属性 rollbackFor 设置错误

rollbackFor 可以指定**能够触发事务回滚**的异常类型。Spring**默认**抛出了未检查unchecked异常（**继承自 RuntimeException** 的异常）或者 Error才回滚事务；其他异常不会触发回滚事务。**如果在事务中抛出其他类型的异常，但却期望 Spring 能够回滚事务，就需要指定 rollbackFor 属性，如果未指定 rollbackFor 属性则事务不会回滚。**

```
// 希望自定义的异常可以进行回滚
@Transactional(propagation= Propagation.REQUIRED,rollbackFor= MyException.class)
```

若在目标方法中抛出的异常是 rollbackFor **指定的异常的子类**，事务同样会回滚。Spring 源码如下：

```
private int getDepth(Class<?> exceptionClass, int depth) {
  if (exceptionClass.getName().contains(this.exceptionName)) {
    // Found it!    return depth;
}
// If we've gone as far as we can go and haven't found it...
if (exceptionClass == Throwable.class) {
    return -1;
}
return getDepth(exceptionClass.getSuperclass(), depth + 1);
}
```

4. 同一个类中方法调用，导致 @Transactional 失效

开发中避免不了会对同一个类里面的方法调用，比如有一个类Test，它的一个方法A，A再调用本类的方法B（不论方法B是用public还是private修饰），但方法A没有声明注解事务，而B方法有。则**外部调用方法A之后，方法B的事务是不会起作用的。这也是经常犯错误的一个地方。**

那为啥会出现这种情况？其实这还是由于使用 `Spring AOP`代理造成的，因为 **只有当事务方法被当前类以外的代码调用时，才会由`Spring`生成的代理对象来管理。**

```
//@Transactional
@GetMapping("/test")
private Integer A() throws Exception {
    CityInfoDict cityInfoDict = new CityInfoDict();
    cityInfoDict.setCityName("2");
    /**
     * B 插入字段为 3的数据
     */
    this.insertB();
    /**
     * A 插入字段为 2的数据
     */
    int insert = cityInfoDictMapper.insert(cityInfoDict);
    return insert;
}

@Transactional()
public Integer insertB() throws Exception {
    CityInfoDict cityInfoDict = new CityInfoDict();
    cityInfoDict.setCityName("3");
    cityInfoDict.setParentCityId(3);
    return cityInfoDictMapper.insert(cityInfoDict);
}
```

5. 异常被你的 catch“吃了”导致 @Transactional 失效

这种情况是最常见的一种 `@Transactional` 注解失效场景，

```
@Transactional
private Integer A() throws Exception {
    int insert = 0;
    try {
        CityInfoDict cityInfoDict = new CityInfoDict();
        cityInfoDict.setCityName("2");
        cityInfoDict.setParentCityId(2);
        /**
         * A 插入字段为 2的数据
         */
        insert = cityInfoDictMapper.insert(cityInfoDict);
```

```
        /**
         * B 插入字段为 3的数据
         */
        b.insertB();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

如果B方法内部抛了异常，而A方法此时try catch了B方法的异常，那这个事务还能正常回滚吗?

答案：不能!

会抛出异常：

```
org.Springframework.transaction.UnexpectedRollbackException: Transaction rolled back because it has been marked as rollback-only
```

因为当ServiceB中抛出了一个异常以后，ServiceB标识当前事务需要rollback。但是ServiceA中由于你手动的捕获这个异常并进行处理，ServiceA认为当前事务应该正常commit。此时就出现了前后不一致，也就是因为这样，抛出了前面的UnexpectedRollbackException异常。

Spring的事务是在调用业务方法之前开始的，业务方法执行完毕之后才执行commit or rollback，事务是否执行取决于是否抛出runtime异常。如果抛出runtime exception 并在你的业务方法中没有catch到的话，事务会回滚。

在业务方法中一般不需要catch异常，如果**非要catch一定要抛出throw new RuntimeException()**，或者注解中指定抛异常类型**@Transactional(rollbackFor=Exception.class)**，否则会导致事务失效，数据commit造成数据不一致，所以有些时候 try catch反倒会画蛇添足。

6. **数据库引擎不支持事务**

这种情况出现的概率并不高，事务能否生效数据库引擎是否支持事务是关键。常用的MySQL数据库默认使用支持事务的innodb引擎。一旦数据库引擎切换成不支持事务的myisam，那事务就从根本上失效了。

事务的源码：

属性解析器：SpringTransactionAnnotationParser，解析@Transactional配置的事务属性

事务拦截器：TransactionInterceptor实现了MetshodInterceptor接口，该拦截器在产生代理对象之前和aop增强逻辑合并，最终一起影响到代理对象。

合并方式：

```
//将它两合到Advisor数组中
Advisor[] advisors = buildAdvisors(beanNames,specificInterceptors);
```

```
    <aop:config>
        <!--advice-ref指向增强=横切逻辑+方位-->
        <aop:advisor advice-ref="txAdvice" pointcut="execution(*
com.lagou.edu.service.impl.TransferServiceImpl.*(..))"/>
    </aop:config>
```

如果异常回滚代码：

```
txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus())
```

如何给Spring 容器提供配置元数据?

这里有三种重要的方法给Spring 容器提供配置元数据。

XML配置文件。

基于注解的配置。

基于java的配置。

@Required 注解

这个注解表明bean的属性必须在配置的时候设置，通过一个bean定义的显式的属性值或通过自动装配，若@Required注解的bean属性未被设置，容器将抛出BeanInitializationException。

@Qualifier 注解

当有多个相同类型的bean却只有一个需要自动装配时，将@Qualifier 注解和@Autowire 注解结合使用以消除这种混淆，指定需要装配的确切的bean。

哪些是重要的bean生命周期方法？ 你能重载它们吗?

有两个重要的bean 生命周期方法，第一个是setup ， 它是在容器加载bean的时候被调用。第二个方法是teardown 它是在容器卸载类的时候被调用。

The bean 标签有两个重要的属性（init-method和destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct和@PreDestroy）。

什么是Spring的内部bean？

当一个bean仅被用作另一个bean的属性时，它能被声明为一个内部bean，为了定义inner bean，在Spring 的 基于XML 的 配置元数据中，可以在 或 元素内使用 元素，内部bean通常是匿名的，它们的Scope一般是prototype。

在 Spring 中如何注入一个java集合?

Spring提供以下几种集合的配置元素：

- 类型用于注入一列值，允许有相同的值。
- 类型用于注入一组值，不允许有相同的值。
- <map.>类型用于注入一组键值对，键和值都可以为任意类型。
- 类型用于注入一组键值对，键和值都只能为String类型。

你可以在Spring中注入一个null 和一个空字符串吗?

可以。

XMLBeanFactory

最常用的就是org.Springframework.beans.factory.xml.XmlBeanFactory ，它根据XML文件中的定义加载beans。该容器从XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

什么是 Spring 装配

当 bean 在 Spring 容器中组合在一起时，它被称为装配或 bean 装配。 Spring 容器需要知道需要什么 bean 以及容器应该如何使用依赖注入来将 bean 绑定在一起，同时装配 bean。

自动装配有哪些方式?

Spring 容器能够自动装配 bean。也就是说，可以通过检查 BeanFactory 的内容让 Spring 自动解析 bean 的协作者。

自动装配的不同模式:

- **no** - 这是默认设置，表示没有自动装配。应使用显式 bean 引用进行装配。
- **byName** - 它根据 bean 的名称注入对象依赖项。它匹配并装配其属性与 XML 文件中由相同名称定义的 bean。
- **byType** - 它根据类型注入对象依赖项。如果属性的类型与 XML 文件中的一个 bean 名称匹配，则匹配并装配属性。
- **构造函数** - 它通过调用类的构造函数来注入依赖项。它有大量的参数。
- **autodetect** - 首先容器尝试通过构造函数使用 autowire 装配，如果不能，则尝试通过 byType 自动装配。

自动装配有什么局限?

- 覆盖的可能性 - 您始终可以使用和设置指定依赖项，这将覆盖自动装配。
- 基本元数据类型 - 简单属性（如原数据类型，字符串和类）无法自动装配。
- 令人困惑的性质 - 总是喜欢使用明确的装配，因为自动装配不太精确。

自定义BeanFactory的作用?

加载解析xml，读取xml中的bean信息

通过反射技术实例化bean对象，放入map中待用

提供接口方法根据id从map中获取bean

3. 下列那个选项不是Spring事务管理的核心接口：（）

- A.PlatformTransactionManager
- B.TransactionDriver
- C.TransactionDefinition
- D.TransactionStatus

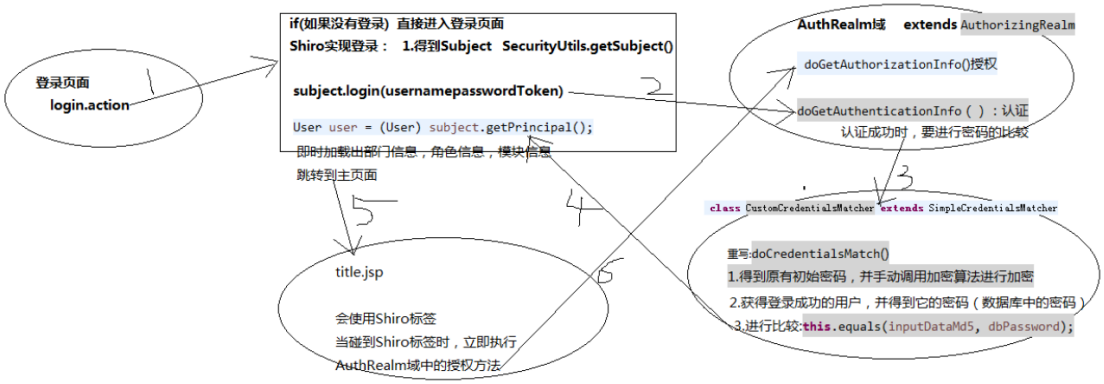
回答错误

正确答案:

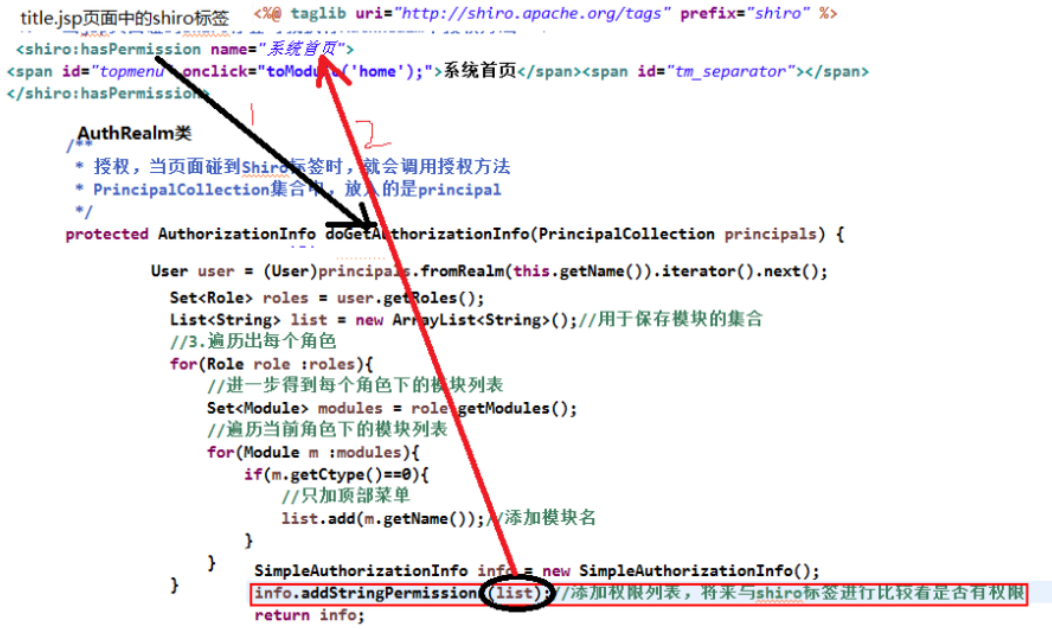
B.TransactionDriver

shiro介绍：

- 认证流程：



- 权限控制（授权流程）：



前端jsp页面：

```
 <%@ taglib uri="http://shiro.apache.org/tags" prefix="shiro" %>

 //要想使用shiro的标签就一点要导数据标签
<shiro:hasPermission name="部门管理">
//shiro:hasPermission：拥有权限的资源  name：权限访问的名字，如上:当该用户里有部门管理权限时，才会让其看到部门管理这个按钮
    <li><a href="${ctx}/sysadmin/deptAction_list" onclick="linkHighlighted(this)" target="main" id="aa_1">部门管理</a></li>
</shiro:hasPermission>
```

以上方法可以使用户看不到自己没有权限的数据，但是如果用户自己写链接还是能访问到数据的（**过滤器链的权限配置方式**）：

```
<value>

                /index.jsp* = anon

                /home* = anon

                /sysadmin/login/login.jsp* = anon

                /sysadmin/login/loginAction_logout* = anon

                /login* = anon

                /logout* = anon

                /components/** = anon
                /css/** = anon
                /img/** = anon
                /js/** = anon
                /plugins/** = anon
                /images/** = anon
                /js/** = anon
                /make/** = anon
                /skin/** = anon
                /stat/** = anon
```

```
                /ufiles/** = anon
                /validator/** = anon
                /resource/** = anon
                 //在这里进行配置，下面的以上为 /sysadmin/deptAction_*路径下的所有方法都必须要有部门管理权限才可以进入访问，（ps：在写方法时，建议写一个在这里就配置一个，避免混淆了）
                /sysadmin/deptAction_* = perms["部门管理"]
                /** = authc
                /*.* = authc
            </value>
```

注解的方式：

```
//这里代表的时要走这个方法模块中就得有角色管理这个模块，没有就拒绝访问
@RequiresPermissions(value="角色管理")
public Page<Role> findPage(Specification<Role> spec, Pageable pageable) {
    return roleDao.findAll(spec, pageable);
}
```

面试题

@Component和@Bean的区别是什么?

- 作用对象不同：@Component注解作用于类，而@Bean注解作用于方法。
- @Component注解通常是通过类路径扫描来自动侦测以及自动装配到Spring容器中（我们可以使用@ComponentScan注解定义要扫描的路径）。

  @Bean注解通常是在标有该注解的方法中定义产生这个bean，告诉Spring这是某个类的实例，当我需要用它的时候还给我。
- @Bean注解比@Component注解的自定义性更强，而且很多地方只能通过@Bean注解来注册bean。比如当引用第三方库的类需要装配到Spring容器的时候，就只能通过@Bean注解来实现。

@Autowired和@Resource的区别是什么?

如下图，@Autowired先按类型找，找不到则按名称找

@Resource相反，先按名称找，找不到按类型找

@Qualifier：结合@Autowired一起使用，自动注入策略由byType变成byName

FileSystemResource和ClassPathResource之间的区别是什么?

在FileSystemResource中你需要给出Spring-config.xml(Spring配置)文件相对于您的项目的相对路径或文件的绝对位置。
在ClassPathResource中Sping查找文件使用ClassPath，因此Spring-config.xml应该包含在类路径下。
一句话,**ClassPathResource在类路径下搜索和FileSystemResource在文件系统下搜索。**

构造方法注入和setter注入之间的区别

1、在Setter注入,可以将依赖项部分注入,构造方法注入不能部分注入，因为调用构造方法如果传入所有的参数就会报错。

2、如果我们为同一属性提供Setter和构造方法注入，Setter注入将覆盖构造方法注入。但是构造方法注入不能覆盖setter注入值。显然，构造方法注入被称为创建实例的第一选项。

3、使用setter注入你不能保证所有的依赖都被注入,这意味着你可以有一个对象依赖没有被注入。在另一方面构造方法注入直到你所有的依赖都注入后才开始创建实例。

4、在构造函数注入,如果A和B对象相互依赖：A依赖于B,B也依赖于A,此时在创建对象的A或者B时，Spring抛出ObjectCurrentlyInCreationException。所以Spring可以通过setter注入,从而解决循环依赖的问题。

ApplicationContext通常的实现是什么?

FileSystemXmlApplicationContext ：此容器从一个XML文件中加载beans的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。

ClassPathXmlApplicationContext：此容器也从一个XML文件中加载beans的定义，这里，你需要正确设置classpath因为这个容器将在classpath里找bean配置。

WebXmlApplicationContext：此容器加载一个XML文件，此文件定义了一个WEB应用的所有bean。

Spring自动装配有哪些方式?

自动装配的不同模式：

no - 这是默认设置，表示没有自动装配。应使用显式 bean 引用进行装配。

byName - 它根据 bean 的名称注入对象依赖项。它匹配并装配其属性与 XML 文件中由相同名称定义的 bean。

byType - 它根据类型注入对象依赖项。如果属性的类型与 XML 文件中的一个 bean 名称匹配，则匹配并装配属性。

构造函数 - 它通过调用类的构造函数来注入依赖项。它有大量的参数。

autodetect - 首先容器尝试通过构造函数使用 autowire 装配，如果不能，则尝试通过 byType 自动装配。

描述Spring和SpringMVC父子容器关系, 父子容器重复扫描会出现什么样的问题, 子容器是否可以使用父容器中的bean, 如果可以如何配置?

1.Spring是父容器，SpringMVC是其子容器，并且在Spring父容器中注册的Bean对于SpringMVC容器中是可见的，而在SpringMVC容器中注册的Bean对于Spring父容器中是不可见的，也就是子容器可以看见父容器中的注册的Bean，反之就不行。

父容器是使用了ContextLoaderListener加载并实例化的ioc容器为父容器
子容器是使用了DispatcherServerlet加载并实例化的ioc容器为子容器
父容器不能调用子容器中任何的组件,子容器可以调用除了controller以外的组件.
2.事务失效
3.可以(默认不能只用)

```
<bean class="org.Springframework.web.servlet.mvc.method.annotation.
            RequestMappingHandlerMapping">
    <property name="detectHandlerMethodsInAncestorContexts">
        <value>true</value>
    </property>
</bean>
```

Spring 中 BeanFactory#getBean 方法是否线程安全的吗?

Spring 中 BeanFactory.getBean 方法是线程安全的，执行过程中加了 synchronized 互斥锁

Spring 中 ObjectFactory 、 BeanFactory 、FactoryBean的区别是什么?

BeanFactory 是个bean 工厂，FactoryBean是个bean。

ObjectFactory仅仅关注一个或者**一种类型Bean**的查找,而且自身不具有依赖查找的能力，BeanFactory则提供**单一类型,集合类型和层次性**的依赖查找能力。(意思是beanfactory查询bean的方式更多)。

- ## BeanFactory介绍:

  BeanFactory负责生产和管理bean的一个工厂。它定义了IOC容器的最基本形式，并提供了IOC容器应遵守的的最基本的接口，它的职责包括：实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。在Spring代码中，BeanFactory只是个接口，并不是IOC容器的具体实现，但是Spring容器给出了很多种实现，如 DefaultListableBeanFactory、XmlBeanFactory、ApplicationContext等，都是附加了某种功能的实现。

```
public interface BeanFactory {

    String FACTORY_BEAN_PREFIX = "&";

    //经过Bean名称获取Bean
    Object getBean(String name) throws BeansException;

    //根据名称和类型获取Bean
    <T> T getBean(String name, Class<T> requiredType) throws BeansException;

    //经过name和对象参数获取Bean
    Object getBean(String name, Object... args) throws BeansException;

    //经过类型获取Bean
    <T> T getBean(Class<T> requiredType) throws BeansException;

    //经过类型和参数获取Bean
    <T> T getBean(Class<T> requiredType, Object... args) throws BeansException;
    <T> ObjectProvider<T> getBeanProvider(Class<T> requiredType);
    <T> ObjectProvider<T> getBeanProvider(ResolvableType requiredType);
    boolean containsBean(String name);
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
    boolean isTypeMatch(String name, ResolvableType typeToMatch) throws
NoSuchBeanDefinitionException;
    boolean isTypeMatch(String name, Class<?> typeToMatch) throws
NoSuchBeanDefinitionException;
    @Nullable
    Class<?> getType(String name) throws NoSuchBeanDefinitionException;
    @Nullable
    Class<?> getType(String name, boolean allowFactoryBeanInit) throws
NoSuchBeanDefinitionException;
    String[] getAliases(String name);

}
```

- ## FactoryBean介绍

  Spring中Bean有两种，一种是普通Bean，一种是工厂Bean（FactoryBean），FactoryBean可以生成某一个类型的Bean实例（返回给我们），也就是说我们可以借助于它自定义Bean的创建过程。

Bean创建的三种方式中的静态方法和实例化方法和FactoryBean作用类似，FactoryBean使用较多，尤其在Spring框架一些组件中会使用，还有其他框架和Spring框架整合时使用

FactoryBean源码：

```
// 可以让我们自定义Bean的创建过程（完成复杂Bean的定义）
public interface FactoryBean<T> {
    @Nullable
    // 返回FactoryBean创建的Bean实例，如果isSingleton返回true，则该实例会放到Spring容器的单例对象缓存池中Map
    T getObject() throws Exception;
    @Nullable
     // 返回FactoryBean创建的Bean类型
     Class<?> getObjectType();
    // 返回作用域是否单例
    default boolean isSingleton() {
    return true;
  }
}
```

FactoryBean使用示例：

```
public class Company {
 private String name;
 private String address;
 private int scale;
 get、set方法省略
}
```

```
public class CompanyFactoryBean implements FactoryBean<Company> {
    private String companyInfo; // 公司名称,地址,规模
    public void setCompanyInfo(String companyInfo) {
        this.companyInfo = companyInfo;
    }
    @Override
    public Company getObject() throws Exception {
        // 模拟创建复杂对象Company
        Company company = new Company();
        String[] strings = companyInfo.split(",");
        company.setName(strings[0]);
        company.setAddress(strings[1]);
        company.setScale(Integer.parseInt(strings[2]));
        return company;
    }
    @Override
    public Class<?> getObjectType() {
        return Company.class;
    }
    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

xml配置

```xml
<bean id="companyBean" class="com.lagou.edu.factory.CompanyFactoryBean">
    <property name="companyInfo" value="拉勾,中关村,500"/>
</bean>
```

```java
//测试，获取FactoryBean产生的对象
Object companyBean = applicationContext.getBean("companyBean");
System.out.println("bean:" + companyBean);
// 结果如下
bean:Company{name='拉勾', address='中关村', scale=500}

//测试，获取FactoryBean对象，需要在id之前添加“&”
Object companyBean = applicationContext.getBean("&companyBean");
System.out.println("bean:" + companyBean);
// 结果如下
bean:com.lagou.edu.factory.CompanyFactoryBean@53f6fd09
```

总结：可以让我们自定义Bean的创建过程，完成复杂Bean的定义。如上例子，我们可以将bean为companyBean的value用“，”分开打印。

- **ObjectFactory 介绍：**

```java
public interface ObjectFactory {
    //为指定对象和环境建立一个对象实例
    public Object getObjectInstance(Object obj, Name name, Context nameCtx,
                                    Hashtable<?,?> environment)
        throws Exception;
}

```

Spring销毁的过程?

1. 找到所有的DisposableBean
2. 遍历找出所有依赖了当前DisposableBean的所有bean，将这些bean从单例池中移除
3. 调用了DisposableBean的destroy方法
4. 找到当前bean的所有inner Bean，将这些Inner Bean全部移除掉

参考博文：https://blog.csdn.net/long9870/article/details/100544690

可以自主销毁单例bean吗?

长时间不用时垃圾管理器会自动销毁。

Spring中使用了哪些设计模式

工厂模式：比如BeanFactory

单例模式：比如将bean设置为singleton

适配器模式：AOP里面有用到

> 适配器模式介绍：把一个类的接口变换成客户端所期待的另一种接口，从而使原本接口不匹配而无法一起工作的两个类能够在一起工作

代理模式：AOP有用到

观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

策略模式：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。 spring 中在实例化对象的时候用到 Strategy 模式在 SimpleInstantiationStrategy（**功能是类的实例化**）中有如下代码说明了策略模式的使用情况：

模板方法模式：JdbcTemplate使用到。

JdbcTemplate使用：

```
        // 创建表的SQL语句
        String sql = "CREATE TABLE product("
                + "pid INT PRIMARY KEY AUTO_INCREMENT,"
                + "pname VARCHAR(20),"
                + "price DOUBLE"
                + ");";

        JdbcTemplate jdbcTemplate = new
JdbcTemplate(DataSourceUtils.getDataSource());
        jdbcTemplate.execute(sql);
```

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(DataSourceUtils.getDataSource());

        String sql = "INSERT INTO product VALUES (NULL, ?, ?);";

        jdbcTemplate.update(sql, "iPhone3GS", 3333);
        jdbcTemplate.update(sql, "iPhone4", 5000);
```

aop通知需要注意什么问题

1.不要把业务逻辑代码放入aop中，应该放入非功能性的需求，比如 权限控制 缓存控制 事物控制审计日志 性能监控 分布式追踪 异常处理

2.无法拦截static final private方法

3.无法拦截内部方法调用

Spring的坑：

1. @Autowired出现空指针：
   - 注入和被注入类没有标记为Spring Bean。
   - 被注入类使用new去获取类（违背了Bean的整个过程被Spring容器管理）。
   - 需要注入的类在被注入类的外层，导致扫描不到。
2. spring中的bean是默认单例，不要在service/controller中使用共享变量：

   因为bean默认单例的，在多线程下使用共享变量是会出现线程安全问题，如果非要使用，可以将共享变量放进Tread Local中，以保证线程安全。
3. https://www.iteye.com/blog/takeme-2383443

   项目中使用session实例