Spring的优点

- 通过控制反转和依赖注入实现**松耦合**。
- 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。
- 通过切面和模板减少样板式代码。
- 声明事物的支持。可以从单调繁冗的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。
- 方便集成各种优秀框架。内部提供了对各种优秀框架的直接支持（如：Hessian、Quartz、MyBatis等）。
- 方便程序的测试。Spring支持Junit4，添加注解便可以测试Spring程序。

Spring 用到了哪些设计模式?

1、**简单工厂模式**：`BeanFactory`就是简单工厂模式的体现，根据传入一个唯一标识来获得 Bean 对象。

```
@Override
public Object getBean(String name) throws BeansException {
    assertBeanFactoryActive();
    return getBeanFactory().getBean(name);
}
```

2、**工厂方法模式**：`FactoryBean`就是典型的工厂方法模式。spring在使用`getBean()`调用获得该bean时，会自动调用该bean的`getObject()`方法。每个 Bean 都会对应一个`FactoryBean`，如`SqlSessionFactory`对应`SqlSessionFactoryBean`。

3、**单例模式**：一个类仅有一个实例，提供一个访问它的全局访问点。Spring 创建 Bean 实例默认是单例的。

4、**适配器模式**：SpringMVC中的适配器`HandlerAdatper`。由于应用会有多个Controller实现，如果需要直接调用Controller方法，那么需要先判断是由哪一个Controller处理请求，然后调用相应的方法。当增加新的 Controller，需要修改原来的逻辑，违反了开闭原则（对修改关闭，对扩展开放）。

为此，Spring提供了一个适配器接口，每一种 Controller 对应一种`HandlerAdapter`实现类，当请求过来，SpringMVC会调用`getHandler()`获取相应的Controller，然后获取该Controller对应的`HandlerAdapter`，最后调用`HandlerAdapter`的`handle()`方法处理请求，实际上调用的是Controller的`handleRequest()`。每次添加新的 Controller 时，只需要增加一个适配器类就可以，无需修改原有的逻辑。

常用的处理器适配器：`SimpleControllerHandlerAdapter`，`HttpRequestHandlerAdapter`，`AnnotationMethodHandlerAdapter`。

```
// Determine handler for the current request.
mappedHandler = getHandler(processedRequest);

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

// Actually invoke the handler.
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

public class HttpRequestHandlerAdapter implements HandlerAdapter {
```

```
    @Override
    public boolean supports(Object handler) {//handler是被适配的对象，这里使用的是对象的适配器模式
        return (handler instanceof HttpRequestHandler);
    }

    @Override
    @Nullable
    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)
        throws Exception {

        ((HttpRequestHandler) handler).handleRequest(request, response);
        return null;
    }
}
```

5、**代理模式**：spring 的 aop 使用了动态代理，有两种方式`JdkDynamicAopProxy`和`Cglib2AopProxy`。

6、**观察者模式**：spring 中 observer 模式常用的地方是 listener 的实现，如`ApplicationListener`。

7、**模板模式**： Spring 中`jdbcTemplate`、`hibernateTemplate`等，就使用到了模板模式。

什么是AOP?

面向切面编程，作为面向对象的一种补充，将公共逻辑（事务管理、日志、缓存等）封装成切面，跟业务代码进行分离，可以减少系统的重复代码和降低模块之间的耦合度。切面就是那些与业务无关，但所有业务模块都会调用的公共逻辑。

AOP有哪些实现方式?

AOP有两种实现方式：静态代理和动态代理。

静态代理

静态代理：代理类是我们定义好的，在编译阶段生成，在编译阶段将通知织入Java字节码中，也称编译时增强。AspectJ使用的是静态代理。

缺点：代理对象需要与目标对象实现一样的接口，并且实现接口的方法，会有冗余代码。同时，一旦接口增加方法，目标对象与代理对象都要维护。

例子：首先，我们创建一个Person接口。这个接口就是学生（被代理类），和班长（代理类）的公共接口，他们都有交作业的行为。这样，学生交作业就可以让班长来代理执行。

```
/**
 * Created by Mapei on 2018/11/7
 * 创建person接口
 */
public interface Person {
    //交作业
    void giveTask();
}
```

Student类实现Person接口，Student可以具体实施交作业这个行为。

```java
/**
 * Created by Mapei on 2018/11/7
 */
public class Student implements Person {
    private String name;
    public Student(String name) {
        this.name = name;
    }

    public void giveTask() {
        System.out.println(name + "交语文作业");
    }
}
```

StudentsProxy类，这个类也实现了Person接口，但是还另外持有一个学生类对象，那么他可以代理学生类对象执行交作业的行为。

```java
/**
 * Created by Mapei on 2018/11/7
 * 学生代理类，也实现了Person接口，保存一个学生实体，这样就可以代理学生产生行为
 */
public class StudentsProxy implements Person{
    //被代理的学生
    Student stu;

    public StudentsProxy(Person stu) {
        // 只代理学生对象
        if(stu.getClass() == Student.class) {
            this.stu = (Student)stu;
        }
    }

    //代理交作业，调用被代理学生的交作业的行为
    public void giveTask() {
        stu.giveTask();
    }
}
```

下面测试一下，看代理模式如何使用：

```java
/**
 * Created by Mapei on 2018/11/7
 */
public class StaticProxyTest {
    public static void main(String[] args) {
        //被代理的学生林浅，他的作业上交有代理对象monitor完成
        Person linqian = new Student("林浅");

        //生成代理对象，并将林浅传给代理对象
        Person monitor = new StudentsProxy(linqian);

        //班长代理交作业
        monitor.giveTask();
    }
}
```

动态代理

动态代理：代理类在程序运行时创建，AOP框架不会去修改字节码，而是在内存中临时生成一个代理对象，在运行期间对业务方法进行增强，不会生成新类。

Spring AOP的实现原理

`Spring`的`AOP`实现原理其实很简单，就是通过**动态代理**实现的。如果我们为`Spring`的某个`bean`配置了切面，那么`Spring`在创建这个`bean`的时候，实际上创建的是这个`bean`的一个代理对象，我们后续对`bean`中方法的调用，实际上调用的是代理类重写的代理方法。而`Spring`的`AOP`使用了两种动态代理，分别是**JDK的动态代理**，以及**CGLib的动态代理**。

JDK动态代理和CGLIB动态代理的区别?

Spring AOP中的动态代理主要有两种方式：JDK动态代理和CGLIB动态代理。在AOP的源码中会判断：如果目标类是接口类（目标对象实现了接口），则直接使用JDKproxy；其他情况使用CGlib动态代理。

JDK动态代理

如果目标类实现了接口，Spring AOP会选择使用JDK动态代理目标类。代理类根据目标类实现的接口动态生成，不需要自己编写，生成的动态代理类和目标类都实现相同的接口。JDK动态代理的核心是`InvocationHandler`接口和`Proxy`类。

缺点：**目标类必须有实现的接口。**如果某个类没有实现接口，那么这个类就不能用JDK动态代理。

CGLIB动态代理

通过继承实现。如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library）可以在运行时动态生成类的字节码，动态创建目标类的子类对象，在子类对象中增强目标类。

CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为`final`，那么它是无法使用CGLIB做动态代理的。

优点：目标类不需要实现特定的接口，更加灵活。

什么时候采用哪种动态代理？（代码底层判断）

1. 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
2. 如果目标对象实现了接口，可以强制使用CGLIB实现AOP
3. 如果目标对象没有实现了接口，必须采用CGLIB库

两者的区别：

1. jdk动态代理使用jdk中的类Proxy来创建代理对象，它使用反射技术来实现，不需要导入其他依赖。cglib需要引入相关依赖：`asm.jar`，它使用字节码增强技术来实现。
2. 当目标类实现了接口的时候Spring AOP默认使用jdk动态代理方式来增强方法，没有实现接口的时候使用cglib动态代理方式增强方法。

Spring AOP相关术语

（1）**切面**（Aspect）：切面是通知和切点的结合。通知和切点共同定义了切面的全部内容。

（2）**连接点**（Join point）：指方法，在Spring AOP中，一个连接点总是代表一个方法的执行。连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

（3）**通知**（Advice）：在AOP术语中，切面的工作被称为通知。

（4）**切入点**（Pointcut）：切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。

（5）**引入**（Introduction）：引入允许我们向现有类添加新方法或属性。

（6）**目标对象**（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。它通常是一个代理对象。

（7）**织入**（Weaving）：织入是把切面应用到目标对象并创建新的代理对象的过程。在目标对象的生命周期里有以下时间点可以进行织入：

- 编译期：切面在目标类编译时被织入。AspectJ的织入编译器是以这种方式织入切面的。
- 类加载期：切面在目标类加载到VM时被织入。需要特殊的类加载器，它可以在目标类被引入应用之前增强该目标类的字节码。AspectJ5的加载时织入就支持以这种方式织入切面。
- 运行期：切面在应用运行的某个时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态地创建一个代理对象。SpringAOP就是以这种方式织入切面。

Spring通知有哪些类型?

在AOP术语中，切面的工作被称为通知。通知实际上是程序运行时要通过Spring AOP框架来触发的代码段。

Spring切面可以应用5种类型的通知：

1. **前置通知**（Before）：在目标方法被调用之前调用通知功能；
2. **后置通知**（After）：在目标方法完成之后调用通知，此时不会关心方法的输出是什么；
3. **返回通知**（After-returning ）：在目标方法成功执行之后调用通知；
4. **异常通知**（After-throwing）：在目标方法抛出异常后调用通知；
5. **环绕通知**（Around）：通知包裹了被通知的方法，在被通知的方法调用之前和调用之后执行自定义的逻辑。

什么是IOC?

IOC：**控制反转**，由Spring容器管理bean的整个生命周期。通过反射实现对其他对象的控制，包括初始化、创建、销毁等，解放手动创建对象的过程，同时降低类之间的耦合度。

IOC的好处?

ioc的思想最核心的地方在于，资源不由使用资源者管理，而由不使用资源的第三方管理，这可以带来很多好处。第一，资源集中管理，实现资源的可配置和易管理。第二，降低了使用资源双方的依赖程度，也就是我们说的耦合度。

也就是说，甲方要达成某种目的不需要直接依赖乙方，它只需要达到的目的告诉第三方机构就可以了，比如甲方需要一双袜子，而乙方它卖一双袜子，它要把袜子卖出去，并不需要自己去直接找到一个卖家来完成袜子的卖出。它也只需要找第三方，告诉别人我要卖一双袜子。这下好了，甲乙双方进行交易活动，都不需要自己直接去找卖家，相当于程序内部开放接口，卖家由第三方作为参数传入。甲乙互相不依赖，而且只有在进行交易活动的时候，甲才和乙产生联系。反之亦然。这样做什么好处么呢，甲乙可以在对方不真实存在的情况下独立存在，而且保证不交易时候无联系，想交易的时候可以很容易的产生联系。甲乙交易活动不需要双方见面，避免了双方的互不信任造成交易失败的问题。因为交易由第三方来负责联系，而且甲乙都认为第三方可靠。那么交易就能很可靠很灵活的产生和进行了。

这就是ioc的核心思想。生活中这种例子比比皆是，支付宝在整个淘宝体系里就是庞大的ioc容器，交易双方之外的第三方，提供可靠性可依赖可灵活变更交易方的资源管理中心。另外人事代理也是，雇佣机构和个人之外的第三方。

参考链接：https://www.zhihu.com/question/23277575/answer/24259844

什么是依赖注入?

在Spring创建对象的过程中，把对象依赖的属性注入到对象中。依赖注入主要有两种方式：构造器注入、Setter注入和注解注入。

@Autowired（自动注入）修饰符有三个属性：Constructor，byType，byName。默认按照byType注入。

constructor：通过构造方法进行自动注入，spring会匹配与构造方法参数类型一致的bean进行注入，如果有一个多参数的构造方法，一个只有一个参数的构造方法，在容器中查找到多个匹配多参数构造方法的bean，那么spring会优先将bean注入到多参数的构造方法中。

byName：被注入bean的id名必须与set方法后半截匹配，并且id名称的第一个单词首字母必须小写，这一点与手动set注入有点不同。
byType：查找所有的set方法，将符合符合参数类型的bean注入。

主要有四种注解可以注册bean，每种注解可以任意使用，只是语义上有所差异：

@Component：可以用于注册所有bean
@Repository：主要用于注册dao层的bean
@Controller：主要用于注册控制层的bean
@Service：主要用于注册服务层的bean

IOC容器初始化过程?

1. 从XML中读取配置文件。
2. 将bean标签解析成 BeanDefinition，如解析 property 元素， 并注入到 BeanDefinition 实例中。
3. 将 BeanDefinition 注册到容器 BeanDefinitionMap 中。
4. BeanFactory 根据 BeanDefinition 的定义信息创建实例化和初始化 bean。

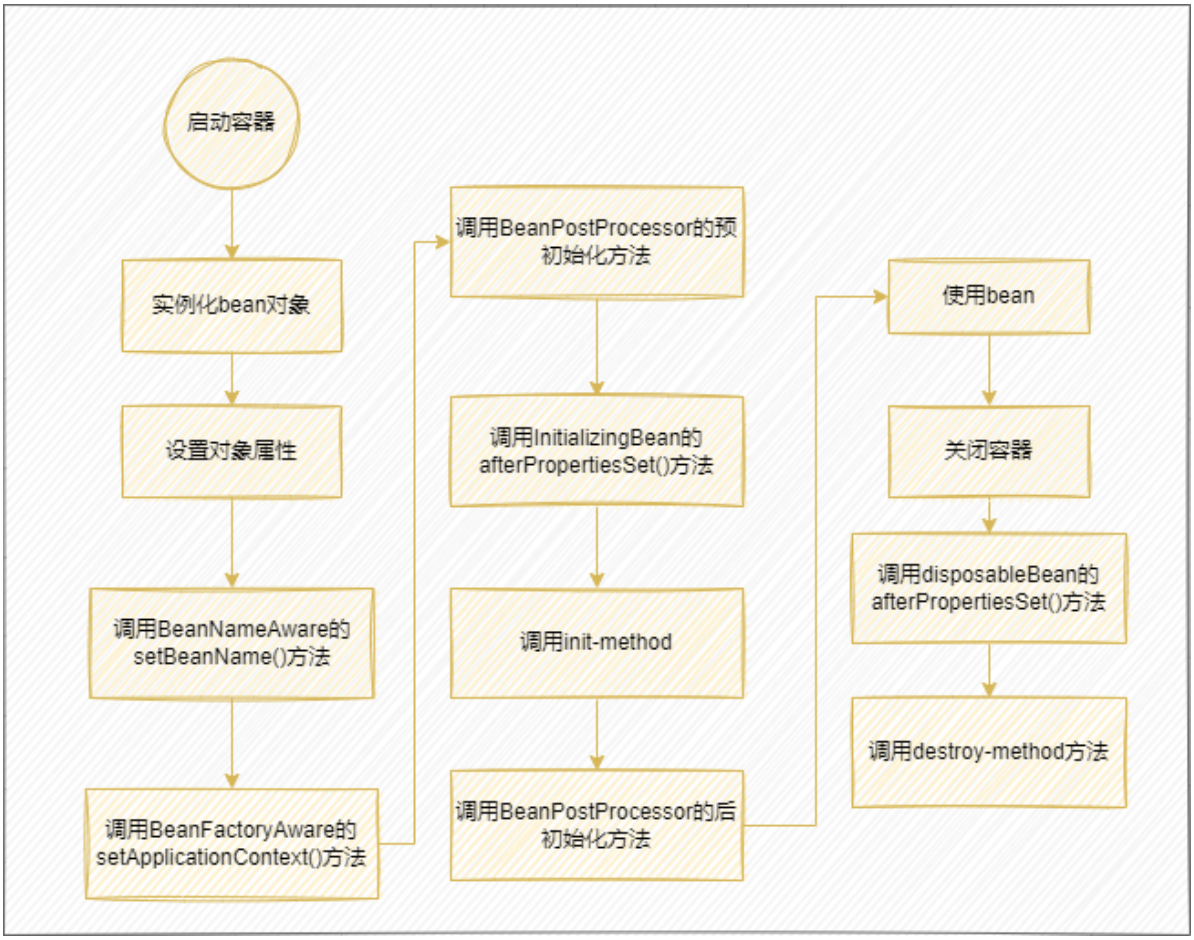
单例bean的初始化以及依赖注入一般都在容器初始化阶段进行，只有懒加载（lazy-init为true）的单例bean是在应用第一次调用getBean()时进行初始化和依赖注入。

```
// AbstractApplicationContext
// Instantiate all remaining (non-lazy-init) singletons.
finishBeanFactoryInitialization(beanFactory);
```

多例bean 在容器启动时不实例化，即使设置 lazy-init 为 false 也没用，只有调用了getBean()才进行实例化。

`loadBeanDefinitions`采用了模板模式，具体加载 `BeanDefinition` 的逻辑由各个子类完成。

Bean的生命周期



1.调用bean的构造方法创建Bean

2.通过反射调用setter方法进行属性的依赖注入

3.如果Bean实现了`BeanNameAware`接口，Spring将调用`setBeanName()`，设置`Bean`的name（xml文件中bean标签的id）

4.如果Bean实现了`BeanFactoryAware`接口，Spring将调用`setBeanFactory()`把bean factory设置给Bean

5.如果Bean实现了`ApplicationContextAware`接口，Spring容器将调用`setApplicationContext()`给Bean设置ApplictionContext

6.如果存在`BeanPostProcessor`，Spring将调用它们的`postProcessBeforeInitialization`（预初始化）方法，在Bean初始化前对其进行处理

7.如果Bean实现了`InitializingBean`接口，Spring将调用它的`afterPropertiesSet`方法，然后调用xml定义的 init-method 方法，两个方法作用类似，都是在初始化 bean 的时候执行

8.如果存在`BeanPostProcessor`，Spring将调用它们的`postProcessAfterInitialization`（后初始化）方法，在Bean初始化后对其进行处理

9.Bean初始化完成，供应用使用，直到应用被销毁

10.如果Bean实现了`DisposableBean`接口，Spring将调用它的`destory`方法，然后调用在xml中定义的`destory-method`方法，这两个方法作用类似，都是在Bean实例销毁前执行

```
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;
    }
}
public interface InitializingBean {
    void afterPropertiesSet() throws Exception;
}
```

BeanFactory和FactoryBean的区别?

BeanFactory：管理Bean的容器，Spring中生成的Bean都是由这个接口的实现来管理的。

FactoryBean：通常是用来创建比较复杂的bean，一般的bean 直接用xml配置即可，但如果一个bean的创建过程中涉及到很多其他的bean 和复杂的逻辑，直接用xml配置比较麻烦，这时可以考虑用FactoryBean，可以隐藏实例化复杂Bean的细节。

当配置文件中bean标签的class属性配置的实现类是FactoryBean时，通过 getBean()方法返回的不是FactoryBean本身，而是调用FactoryBean#getObject()方法所返回的对象，相当于FactoryBean#getObject()代理了getBean()方法。如果想得到FactoryBean必须使用 '&' + beanName 的方式获取。

Mybatis 提供了 `SqlSessionFactoryBean`，可以简化 `SqlSessionFactory` 的配置：

```
public class SqlSessionFactoryBean implements FactoryBean<SqlSessionFactory>,
InitializingBean, ApplicationListener<ApplicationEvent> {
  @Override
  public void afterPropertiesSet() throws Exception {
    notNull(dataSource, "Property 'dataSource' is required");
    notNull(sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is
required");
    state((configuration == null && configLocation == null) || !(configuration
!= null && configLocation != null),
              "Property 'configuration' and 'configLocation' can not specified
with together");
    this.sqlSessionFactory = buildSqlSessionFactory();
  }

  protected SqlSessionFactory buildSqlSessionFactory() throws IOException {
    //复杂逻辑
  }

  @Override
  public SqlSessionFactory getObject() throws Exception {
    if (this.sqlSessionFactory == null) {
      afterPropertiesSet();
    }
    return this.sqlSessionFactory;
  }
```

```
}
```

在 xml 配置 SqlSessionFactoryBean：

```
<bean id="tradeSqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="trade" />
    <property name="mapperLocations" value="classpath*:mapper/trade/*Mapper.xml"
/>
    <property name="configLocation" value="classpath:mybatis-config.xml" />
    <property name="typeAliasesPackage"
value="com.bytebeats.mybatis3.domain.trade" />
</bean>
```

Spring 将会在应用启动时创建 `SqlSessionFactory`，并使用 `sqlSessionFactory` 这个名字存储起来。

Bean注入容器有哪些方式?

1、@Configuration + @Bean

@Configuration用来声明一个配置类，然后使用 @Bean 注解，用于声明一个bean，将其加入到Spring容器中。

```
@Configuration
public class MyConfiguration {
    @Bean
    public Person person() {
        Person person = new Person();
        person.setName("大彬");
        return person;
    }
}
```

2、通过包扫描特定注解的方式

@ComponentScan放置在我们的配置类上，然后可以指定一个路径，进行扫描带有特定注解的bean，然后加至容器中。

特定注解包括@Controller、@Service、@Repository、@Component

```
@Component
public class Person {
    //...
}

@ComponentScan(basePackages = "com.dabin.test.*")
public class Demo1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(Demo1.class);
        Person bean = applicationContext.getBean(Person.class);
        System.out.println(bean);
    }
}
```

3、@Import注解导入

@Import注解平时开发用的不多，但是也是非常重要的，在进行Spring扩展时经常会用到，它经常搭配自定义注解进行使用，然后往容器中导入一个配置文件。

```
@ComponentScan
/*把用到的资源导入到当前容器中*/
@Import({Person.class})
public class App {
    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context =
SpringApplication.run(App.class, args);
        System.out.println(context.getBean(Person.class));
        context.close();
    }
}
```

4、实现BeanDefinitionRegistryPostProcessor进行后置处理。

在Spring容器启动的时候会执行 BeanDefinitionRegistryPostProcessor 的 postProcessBeanDefinitionRegistry 方法，就是等beanDefinition加载完毕之后，对beanDefinition进行后置处理，可以在此进行调整IOC容器中的beanDefinition，从而干扰到后面进行初始化bean。

在下面的代码中，我们手动向beanDefinitionRegistry中注册了person的BeanDefinition。最终成功将person加入到applicationContext中。

```
public class Demo1 {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext();
        MyBeanDefinitionRegistryPostProcessor
beanDefinitionRegistryPostProcessor = new
MyBeanDefinitionRegistryPostProcessor();

 applicationContext.addBeanFactoryPostProcessor(beanDefinitionRegistryPostProces
sor);
        applicationContext.refresh();
        Person bean = applicationContext.getBean(Person.class);
        System.out.println(bean);
    }
}

class MyBeanDefinitionRegistryPostProcessor implements
BeanDefinitionRegistryPostProcessor {
    @Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry
registry) throws BeansException {
        AbstractBeanDefinition beanDefinition =
BeanDefinitionBuilder.rootBeanDefinition(Person.class).getBeanDefinition();
        registry.registerBeanDefinition("person", beanDefinition);
    }

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException {
    }
}
```

5、使用FactoryBean接口

如下图代码，使用@Configuration + @Bean的方式将 PersonFactoryBean 加入到容器中，这里没有向容器中直接注入 Person，而是注入 PersonFactoryBean，然后从容器中拿Person这个类型的bean。

```
@Configuration
public class Demo1 {
    @Bean
    public PersonFactoryBean personFactoryBean() {
        return new PersonFactoryBean();
    }

    public static void main(String[] args) {
        AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(Demo1.class);
        Person bean = applicationContext.getBean(Person.class);
        System.out.println(bean);
    }
}

class PersonFactoryBean implements FactoryBean<Person> {
    @Override
    public Person getObject() throws Exception {
        return new Person();
    }

    @Override
    public Class<?> getObjectType() {
        return Person.class;
    }
}
```

Bean的作用域

1、**singleton**：单例，Spring中的bean默认都是单例的。

2、**prototype**：每次请求都会创建一个新的bean实例。

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

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

5、**global-session**：全局session作用域。

Spring自动装配的方式有哪些?

Spring的自动装配有三种模式：**byType**(根据类型)，**byName**(根据名称)、**constructor**(根据构造函数)。

byType

找到与依赖类型相同的bean注入到另外的bean中，这个过程需要借助setter注入来完成，因此必须存在set方法，否则注入失败。

当xml文件中存在多个相同类型名称不同的实例Bean时，Spring容器依赖注入仍然会失败，因为存在多种适合的选项，Spring容器无法知道该注入那种，此时我们需要为Spring容器提供帮助，指定注入那个Bean实例。可以通过`<bean>`标签的autowire-candidate设置为false来过滤那些不需要注入的实例Bean

```
<bean id="userDao"  class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<!-- autowire-candidate="false" 过滤该类型 -->
<bean id="userDao2" autowire-candidate="false"
class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<!-- byType 根据类型自动装配userDao-->
<bean id="userService" autowire="byType"
class="com.zejian.spring.springIoc.service.impl.UserServiceImpl" />
```

byName

将属性名与bean名称进行匹配，如果找到则注入依赖bean。

```
<bean id="userDao"  class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />
<bean id="userDao2" class="com.zejian.spring.springIoc.dao.impl.UserDaoImpl" />

<!-- byName 根据名称自动装配，找到UserServiceImpl名为 userDao属性并注入-->
<bean id="userService" autowire="byName"
class="com.zejian.spring.springIoc.service.impl.UserServiceImpl" />
```

constructor

存在单个实例则优先按类型进行参数匹配（无论名称是否匹配），当存在多个类型相同实例时，按名称优先匹配，如果没有找到对应名称，则注入失败。

@Autowired和@Resource的区别?

Autowire是spring的注解。默认情况下@Autowired是按类型匹配的(byType)。如果需要按名称(byName)匹配的话，可以使用@Qualifier注解与@Autowired结合。@Autowired 可以传递一个`required=false`的属性，false指明当userDao实例存在就注入不存就忽略，如果为true，就必须注入，若userDao实例不存在，就抛出异常。

```
public class UserServiceImpl implements UserService {
    //标注成员变量
    @Autowired
    @Qualifier("userDao1")
    private UserDao userDao;
 }
```

Resource是j2ee的注解，默认按 byName模式自动注入。@Resource有两个中重要的属性：name和type。name属性指定bean的名字，type属性则指定bean的类型。因此使用name属性，则按byName模式的自动注入策略，如果使用type属性，则按 byType模式自动注入策略。倘若既不指定name也不指定type属性，Spring容器将通过反射技术默认按byName模式注入。

```
@Resource(name="userDao")
private UserDao  userDao;//用于成员变量

//也可以用于set方法标注
@Resource(name="userDao")
public void setUserDao(UserDao userDao) {
   this.userDao= userDao;
}
```

上述两种自动装配的依赖注入并不适合简单值类型，如int、boolean、long、String以及Enum等，对于这些类型，Spring容器也提供了@Value注入的方式。

@Value和@Autowired、@Resource类似，也是用来对属性进行注入的，只不过@Value是用来从Properties文件中来获取值的，并且@Value可以解析SpEL(Spring表达式)。

比如，jdbc.properties文件如下：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://127.0.0.1:3306/test?characterEncoding=UTF-
8&allowMultiQueries=true
jdbc.username=root
jdbc.password=root
```

利用注解@Value获取jdbc.url和jdbc.username的值，实现如下：

```
public class UserServiceImpl implements UserService {
    //占位符方式
    @Value("${jdbc.url}")
    private String url;
    //SpEL表达方式，其中代表xml配置文件中的id值configProperties
    @Value("#{configProperties['jdbc.username']}")
    private String userName;

}
```

@Qualifier 注解有什么作用

当需要创建多个相同类型的 bean 并希望仅使用属性装配其中一个 bean 时，可以使用`@Qualifier`注解和`@Autowired`通过指定应该装配哪个 bean 来消除歧义。

@Bean和@Component有什么区别?

都是使用注解定义 Bean。@Bean 是使用 Java 代码装配 Bean，@Component 是自动装配 Bean。

@Component 注解用在类上，表明一个类会作为组件类，并告知Spring要为这个类创建bean，每个类对应一个 Bean。

@Bean 注解用在方法上，表示这个方法会返回一个 Bean。@Bean 需要在配置类中使用，即类上需要加上@Configuration注解。

```
@Component
public class Student {
    private String name = "lkm";

    public String getName() {
```

```
        return name;
    }
}

@Configuration
public class WebSocketConfig {
    @Bean
    public Student student(){
        return new Student();
    }
}
```

@Bean 注解更加灵活。当需要将第三方类装配到 Spring 容器中，因为没办法源代码上添加@Component注解，只能使用@Bean 注解的方式，当然也可以使用 xml 的方式。

@Component、@Controller、@Repositor和@Service的区别?

@Component：最普通的组件，可以被注入到spring容器进行管理。

@Controller：将类标记为 Spring Web MVC 控制器。

@Service：将类标记为业务层组件。

@Repository：将类标记为数据访问组件，即DAO组件。

Spring 事务实现方式有哪些?

事务就是一系列的操作原子执行。Spring事务机制主要包括声明式事务和编程式事务。

- **编程式事务**：通过编程的方式管理事务，这种方式带来了很大的灵活性，但很难维护。
- **声明式事务**：将事务管理代码从业务方法中分离出来，通过aop进行封装。Spring声明式事务使得我们无需要去处理获得连接、关闭连接、事务提交和回滚等这些操作。使用 `@Transactional` 注解开启声明式事务。

`@Transactional` 相关属性如下：

属性	类型	描述
value	String	可选的限定描述符，指定使用的事务管理器
propagation	enum: Propagation	可选的事务传播行为设置
isolation	enum: Isolation	可选的事务隔离级别设置
readOnly	boolean	读写或只读事务，默认读写
timeout	int (in seconds granularity)	事务超时时间设置
rollbackFor	Class对象数组，必须继承自Throwable	导致事务回滚的异常类数组
rollbackForClassName	类名数组，必须继承自Throwable	导致事务回滚的异常类名字数组
noRollbackFor	Class对象数组，必须继承自Throwable	不会导致事务回滚的异常类数组
noRollbackForClassName	类名数组，必须继承自Throwable	不会导致事务回滚的异常类名字数组

有哪些事务传播行为?

在TransactionDefinition接口中定义了七个事务传播行为：

1. `PROPAGATION_REQUIRED`如果存在一个事务，则支持当前事务。如果没有事务则开启一个新的事务。如果嵌套调用的两个方法都加了事务注解，并且运行在相同线程中，则这两个方法使用相同的事务中。如果运行在不同线程中，则会开启新的事务。
2. `PROPAGATION_SUPPORTS` 如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行。
3. `PROPAGATION_MANDATORY` 如果已经存在一个事务，支持当前事务。如果不存在事务，则抛出异常`IllegalTransactionStateException`。
4. `PROPAGATION_REQUIRES_NEW` 总是开启一个新的事务。需要使用JtaTransactionManager作为事务管理器。
5. `PROPAGATION_NOT_SUPPORTED` 总是非事务地执行，并挂起任何存在的事务。需要使用JtaTransactionManager作为事务管理器。
6. `PROPAGATION_NEVER` 总是非事务地执行，如果存在一个活动事务，则抛出异常。
7. `PROPAGATION_NESTED` 如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务, 则按PROPAGATION_REQUIRED 属性执行。

PROPAGATION_NESTED 与PROPAGATION_REQUIRES_NEW的区别:

使用`PROPAGATION_REQUIRES_NEW`时，内层事务与外层事务是两个独立的事务。一旦内层事务进行了提交后，外层事务不能对其进行回滚。两个事务互不影响。

使用`PROPAGATION_NESTED`时，外层事务的回滚可以引起内层事务的回滚。而内层事务的异常并不会导致外层事务的回滚，它是一个真正的嵌套事务。

Spring事务在什么情况下会失效?

1.访问权限问题

java的访问权限主要有四种：private、default、protected、public，它们的权限从左到右，依次变大。

如果事务方法的访问权限不是定义成public，这样会导致事务失效，因为spring要求被代理方法必须是`public`的。

翻开源码，可以看到，在`AbstractFallbackTransactionAttributeSource`类的`computeTransactionAttribute`方法中有个判断，如果目标方法不是public，则返回null，即不支持事务。

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

2. 方法用final修饰

如果事务方法用final修饰，将会导致事务失效。因为spring事务底层使用了aop，也就是通过jdk动态代理或者cglib，帮我们生成了代理类，在代理类中实现的事务功能。

但如果某个方法用final修饰了，那么在它的代理类中，就无法重写该方法，而添加事务功能。

> 同理，如果某个方法是static的，同样无法通过动态代理，变成事务方法。

3.对象没有被spring管理

使用spring事务的前提是：对象要被spring管理，需要创建bean实例。如果类没有加@Controller、@Service、@Component、@Repository等注解，即该类没有交给spring去管理，那么它的方法也不会生成事务。

4.表不支持事务

如果MySQL使用的存储引擎是myisam，这样的话是不支持事务的。因为myisam存储引擎不支持事务。

5.方法内部调用

如下代码所示，update方法上面没有加`@Transactional`注解，调用有`@Transactional`注解的updateOrder 方法，updateOrder 方法上的事务会失效。

因为发生了自身调用，调用该类自己的方法，而没有经过 Spring 的代理类，只有在外部调用事务才会生效。

```
@Service
public class OrderServiceImpl implements OrderService {

    public void update(Order order) {
        this.updateOrder(order);
    }

    @Transactional
    public void updateOrder(Order order) {
        // update order
    }
}
```

解决方法：

1、再声明一个service，将内部调用改为外部调用

2、使用编程式事务

3、使用AopContext.currentProxy()获取代理对象

```
@Servcie
public class OrderServiceImpl implements OrderService {

   public void update(Order order) {
        ((OrderService)AopContext.currentProxy()).updateOrder(order);
   }

    @Transactional
    public void updateOrder(Order order) {
        // update order
    }
}
```

6.未开启事务

如果是spring项目，则需要在配置文件中手动配置事务相关参数。如果忘了配置，事务肯定是不会生效的。

如果是springboot项目，那么不需要手动配置。因为springboot已经在`DataSourceTransactionManagerAutoConfiguration`类中帮我们开启了事务。

7.吞了异常

有时候事务不会回滚，有可能是在代码中手动catch了异常。因为开发者自己捕获了异常，又没有手动抛出，把异常吞掉了，这种情况下spring事务不会回滚。

如果想要spring事务能够正常回滚，必须抛出它能够处理的异常。如果没有抛异常，则spring认为程序是正常的。

Spring怎么解决循环依赖的问题?

构造器注入的循环依赖：Spring处理不了，直接抛出`BeanCurrentlylnCreationException`异常。

单例模式下属性注入的循环依赖：通过三级缓存处理循环依赖。

非单例循环依赖：无法处理。

下面分析单例模式下属性注入的循环依赖是怎么处理的：

首先，Spring单例对象的初始化大略分为三步：

1. `createBeanInstance`：实例化bean，使用构造方法创建对象，为对象分配内存。
2. `populateBean`：进行依赖注入。
3. `initializeBean`：初始化bean。

Spring为了解决单例的循环依赖问题，使用了三级缓存：

`singletonObjects`：完成了初始化的单例对象map，bean name --> bean instance

`earlySingletonObjects`：完成实例化未初始化的单例对象map，bean name --> bean instance

`singletonFactories`： 单例对象工厂map，bean name --> ObjectFactory，单例对象实例化完成之后会加入singletonFactories。

在调用createBeanInstance进行实例化之后，会调用addSingletonFactory，将单例对象放到singletonFactories中。

```
protected void addSingletonFactory(String beanName, ObjectFactory<?>
singletonFactory) {
    Assert.notNull(singletonFactory, "Singleton factory must not be null");
    synchronized (this.singletonObjects) {
        if (!this.singletonObjects.containsKey(beanName)) {
            this.singletonFactories.put(beanName, singletonFactory);
            this.earlySingletonObjects.remove(beanName);
            this.registeredSingletons.add(beanName);
        }
    }
}
```

假如A依赖了B的实例对象，同时B也依赖A的实例对象。

1. A首先完成了实例化，并且将自己添加到singletonFactories中
2. 接着进行依赖注入，发现自己依赖对象B，此时就尝试去get(B)
3. 发现B还没有被实例化，对B进行实例化
4. 然后B在初始化的时候发现自己依赖了对象A，于是尝试get(A)，尝试一级缓存singletonObjects和二级缓存earlySingletonObjects没找到，尝试三级缓存singletonFactories，由于A初始化时将自己添加到了singletonFactories，所以B可以拿到A对象，然后将A从三级缓存中移到二级缓存中
5. B拿到A对象后顺利完成了初始化，然后将自己放入到一级缓存singletonObjects中
6. 此时返回A中，A此时能拿到B的对象顺利完成自己的初始化

由此看出，属性注入的循环依赖主要是通过将实例化完成的bean添加到singletonFactories来实现的。而使用构造器依赖注入的bean在实例化的时候会进行依赖注入，不会被添加到singletonFactories中。比如A和B都是通过构造器依赖注入，A在调用构造器进行实例化的时候，发现自己依赖B，B没有被实例化，就会对B进行实例化，此时A未实例化完成，不会被添加到singtonFactories。而B依赖于A，B会去三级缓存寻找A对象，发现不存在，于是又会实例化A，A实例化了两次，从而导致抛异常。

总结：1、利用缓存识别已经遍历过的节点； 2、利用Java引用，先提前设置对象地址，后完善对象。

Spring启动过程

1. 读取web.xml文件。
2. 创建 ServletContext，为 ioc 容器提供宿主环境。
3. 触发容器初始化事件，调用 contextLoaderListener.contextInitialized()方法，在这个方法会初始化一个应用上下文WebApplicationContext，即 Spring 的 ioc 容器。ioc 容器初始化完成之后，会被存储到 ServletContext 中。
4. 初始化web.xml中配置的Servlet。如DispatcherServlet，用于匹配、处理每个servlet请求。

Spring 的单例 Bean 是否有并发安全问题?

当多个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，这时多个线程会并发执行该请求对应的业务逻辑，如果业务逻辑有对单例状态的修改（体现为此单例的成员属性），则必须考虑线程安全问题。

无状态bean和有状态bean

- 有实例变量的bean，可以保存数据，是非线程安全的。
- 没有实例变量的bean，不能保存数据，是线程安全的。

在Spring中无状态的Bean适合用单例模式，这样可以共享实例提高性能。有状态的Bean在多线程环境下不安全，一般用 `Prototype` 模式或者使用 `ThreadLocal` 解决线程安全问题。

Spring Bean如何保证并发安全?

Spring的Bean默认都是单例的，某些情况下，单例是并发不安全的。

以 `Controller` 举例，假如我们在 `Controller` 中定义了成员变量。当多个请求来临，进入的都是同一个单例的 `Controller` 对象，并对此成员变量的值进行修改操作，因此会互相影响，会有并发安全的问题。

应该怎么解决呢?

为了让多个HTTP请求之间不互相影响，可以采取以下措施：

1、单例变原型

对 web 项目，可以 `Controller` 类上加注解 @`Scope("prototype")` 或 `@Scope("request")`，对非 web 项目，在 `Component` 类上添加注解 `@Scope("prototype")` 。

这种方式实现起来非常简单，但是很大程度上增大了 Bean 创建实例化销毁的服务器资源开销。

2、尽量避免使用成员变量

在业务允许的条件下，可以将成员变量替换为方法中的局部变量。这种方式个人认为是最恰当的。

3、使用并发安全的类

如果非要在单例Bean中使用成员变量，可以考虑使用并发安全的容器，如 `ConcurrentHashMap`、`ConcurrentHashSet` 等等，将我们的成员变量包装到这些并发安全的容器中进行管理即可。

4、分布式或微服务的并发安全

如果还要进一步考虑到微服务或分布式服务的影响，方式3便不合适了。这种情况下可以借助于可以共享某些信息的分布式缓存中间件，如Redis等。这样即可保证同一种服务的不同服务实例都拥有同一份共享信息了。

@Async注解的原理

当我们调用第三方接口或者方法的时候，我们不需要等待方法返回才去执行其它逻辑，这时如果响应时间过长，就会极大的影响程序的执行效率。所以这时就需要使用异步方法来并行执行我们的逻辑。在springboot中可以使用@Async注解实现异步操作。

使用@Async注解实现异步操作的步骤：

1.首先在启动类上添加 @EnableAsync 注解。

```
@Configuration
@EnableAsync
public class App {
    public static void main(String[] args) {
         ApplicationContext ctx = new
             AnnotationConfigApplicationContext(App.class);
        MyAsync service = ctx.getBean(MyAsync.class);
        System.out.println(service.getClass());
        service.async1();
        System.out.println("main thread finish...");
    }
}
```

2.在对应的方法上添加@Async注解。

```
@Component
public class MyAsync {
    @Async
    public void asyncTest() {
        try {
            TimeUnit.SECONDS.sleep(20);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("asyncTest...");
    }
}
```

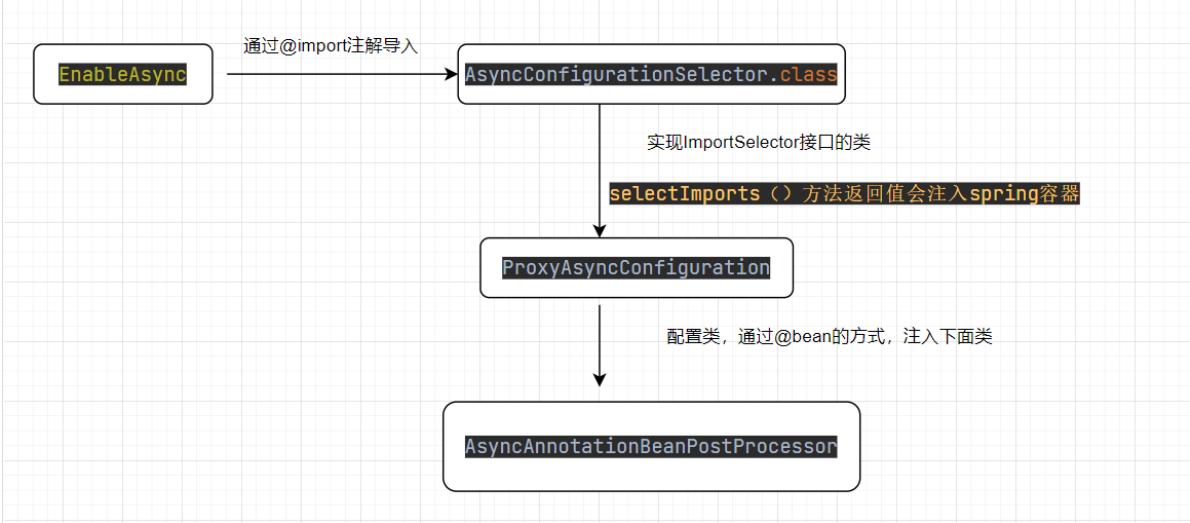
运行代码，控制台输出：

```
main thread finish...
asyncTest...
```

证明asyncTest方法异步执行了。

原理：

我们在主启动类上贴了一个@EnableAsync注解，才能使用@Async生效。@EnableAsync的作用是通过@import导入了AsyncConfigurationSelector。在AsyncConfigurationSelector的selectImports方法将ProxyAsyncConfiguration定义为Bean注入容器。在ProxyAsyncConfiguration中通过@Bean的方式注入AsyncAnnotationBeanPostProcessor类。



代码如下：

```
@Import(AsyncConfigurationSelector.class)
public @interface EnableAsync {
}

public class AsyncConfigurationSelector extends
AdviceModeImportSelector<EnableAsync> {
    public String[] selectImports(AdviceMode adviceMode) {
        switch (adviceMode) {
            case PROXY:
                return new String[] { ProxyAsyncConfiguration.class.getName() };
            //...
        }
```

```
    }
}

public class ProxyAsyncConfiguration extends AbstractAsyncConfiguration {
    @Bean(name = TaskManagementConfigUtils.ASYNC_ANNOTATION_PROCESSOR_BEAN_NAME)
    public AsyncAnnotationBeanPostProcessor asyncAdvisor() {
        //创建postProcessor
        AsyncAnnotationBeanPostProcessor bpp = new
AsyncAnnotationBeanPostProcessor();
        //...
    }
}
```

AsyncAnnotationBeanPostProcessor往往期创建了一个增强器AsyncAnnotationAdvisor。在AsyncAnnotationAdvisor的buildAdvice方法中，创建了AnnotationAsyncExecutionInterceptor。

```
public class AsyncAnnotationBeanPostProcessor extends
AbstractBeanFactoryAwareAdvisingPostProcessor {
    @Override
    public void setBeanFactory(BeanFactory beanFactory) {
        super.setBeanFactory(beanFactory);
        //创建一个增强器
        AsyncAnnotationAdvisor advisor = new
AsyncAnnotationAdvisor(this.executor, this.exceptionHandler);
        //...
        advisor.setBeanFactory(beanFactory);
        this.advisor = advisor;
    }
}


public class AsyncAnnotationAdvisor extends AbstractPointcutAdvisor implements
BeanFactoryAware {
    public AsyncAnnotationAdvisor(
            @Nullable Supplier<Executor> executor, @Nullable
Supplier<AsyncUncaughtExceptionHandler> exceptionHandler) {
        //增强方法
        this.advice = buildAdvice(executor, exceptionHandler);
        this.pointcut = buildPointcut(asyncAnnotationTypes);
    }

    // 委托给AnnotationAsyncExecutionInterceptor拦截器
    protected Advice buildAdvice(
            @Nullable Supplier<Executor> executor, @Nullable
Supplier<AsyncUncaughtExceptionHandler> exceptionHandler) {
        //拦截器
        AnnotationAsyncExecutionInterceptor interceptor = new
AnnotationAsyncExecutionInterceptor(null);
        interceptor.configure(executor, exceptionHandler);
        return interceptor;
    }
}
```

AnnotationAsyncExecutionInterceptor继承自AsyncExecutionInterceptor，间接实现了MethodInterceptor。该拦截器的实现的invoke方法把原来方法的调用提交到新的线程池执行，从而实现了方法的异步。

```
public class AsyncExecutionInterceptor extends AsyncExecutionAspectSupport
implements MethodInterceptor, Ordered {
    public Object invoke(final MethodInvocation invocation) throws Throwable {
        //...
        //构建放到AsyncTaskExecutor执行Callable Task
        Callable<Object> task = () -> {
            //...
        };
        //提交到新的线程池执行
        return doSubmit(task, executor, invocation.getMethod().getReturnType());
    }
}
```

由上面分析可以看到，@Async注解其实是通过代理的方式来实现异步调用的。

那使用@Async有什么要注意的呢?

1.使用@Aysnc的时候最好配置一个线程池Executor以让线程复用节省资源，或者为SimpleAsyncTaskExecutor设置基于线程池实现的ThreadFactory，在否则会默认使用SimpleAsyncTaskExecutor，该executor会在每次调用时新建一个线程。

2.调用本类的异步方法是不会起作用的。这种方式绕过了代理而直接调用了方法，@Async注解会失效。