课程主题

IoC模块源码阅读&AOP核心概念详解&动态代理设计模式

课程目标

7. 可以自主完成阅读Spring框架中BeanDefinition注册流程的源码
8. 可以自主完成阅读Spring框架中Bean实例创建流程的源码
9. 可以自主完成阅读Spring框架中依赖注入流程的源码
10. 可以确定aop流程的源码阅读入口
11. 搞清楚aop相关的核心概念（通知、切面、切入点等）
12. 搞清楚cglib和jdk动态代理的区别
13. 掌握cglib和jdk产生代理对象的方式
14. 掌握cglib和jdk产生代理对象的底层原理
15. 掌握cglib和jdk如何动态添加代理对象的增强功能。

课程回顾

1. BeanDefinition的注册流程相关类
    1. Resource接口和Classpath实现类
    2. XMLBeanDefinitionReader
    3. XMLBeanDefinitionDocumentReader
    4. ValueResolver
    5. BeanDefinitionRegistry（DefaultListableBeanFactory）
2. Bean创建流程中相关类有哪些
    1. BeanFactory、
    2. AbstractBeanFactory、
    3. AbstractAutowireCapableBeanFactory、
    4. DefaultListableBeanFactory
    5. SingletonBeanRegistry接口和实现类

课程内容

一、IoC源码阅读

1 基础容器的BeanDefinition注册流程源码阅读

> 原来Spring提供了一个基础容器的实现：XMLBeanFactory
>
> 但是后来这个类被遗弃了，使用DefaultListableBeanFactory来代替。

入口1

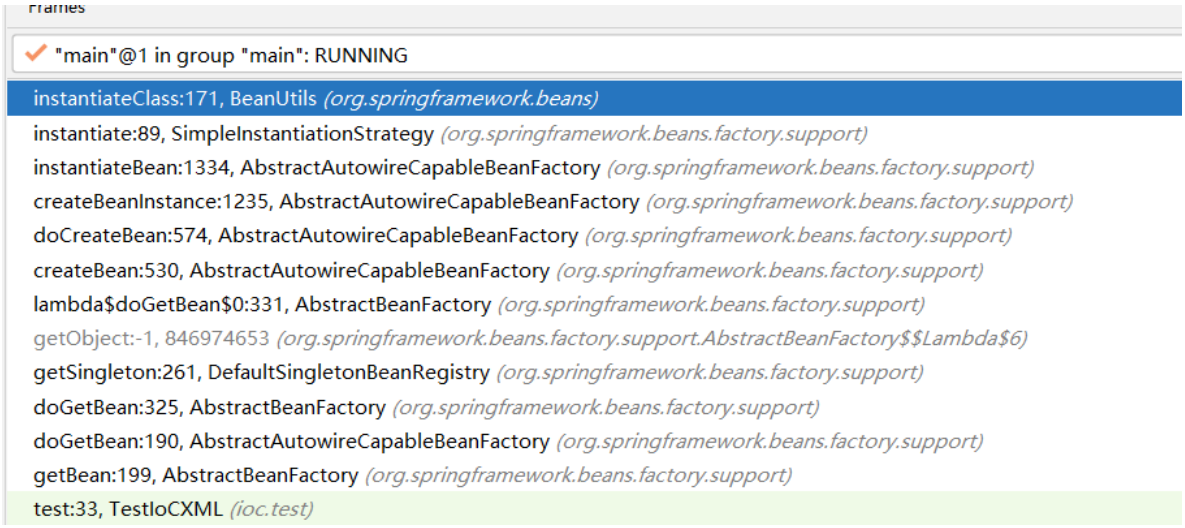
XmlBeanFactory#构造方法中

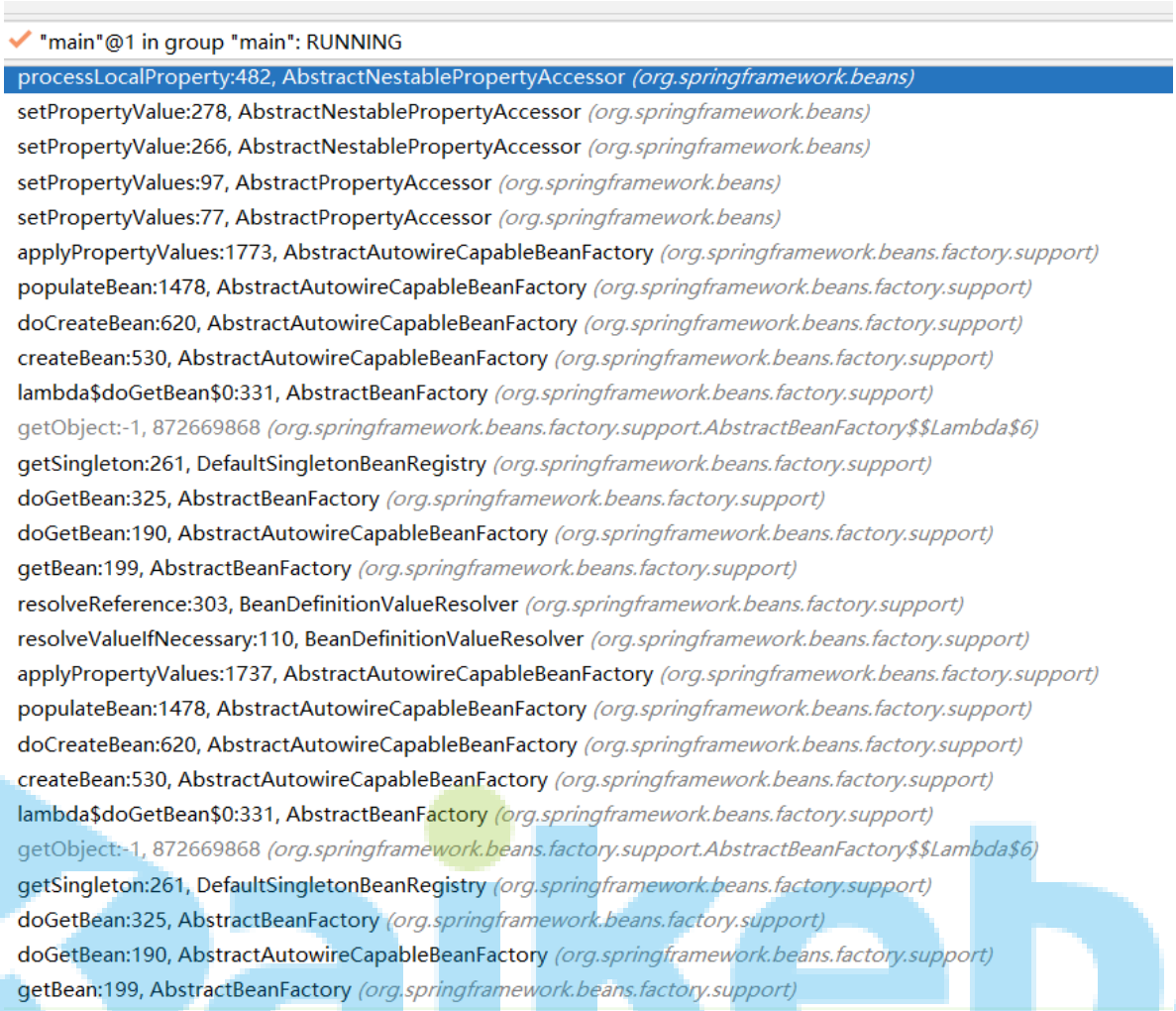
```java
@Test
    public void test11() {
        // 指定XML路径
        String path = "spring/beans.xml";
        Resource resource = new ClassPathResource(path);
        XmlBeanFactory beanFactory = new XmlBeanFactory(resource );
        // Bean实例创建流程
        DataSource dataSource = (DataSource)
beanFactory.getBean("dataSource");
        System.out.println(dataSource);
    }
```

入口2

> XmlBeanDefinitionReader#loadBeanDefinitions();

```java
    @Test
    public void test1() {
        // 指定XML路径
        String path = "spring/beans.xml";
        // 创建DefaultListableBeanFactory工厂，这也就是Spring的基本容器
        DefaultListableBeanFactory beanFactory = new
DefaultListableBeanFactory();
        // 创建BeanDefinition阅读器
        XmlBeanDefinitionReader reader = new
XmlBeanDefinitionReader(beanFactory);
        // 进行BeanDefinition注册流程
        reader.loadBeanDefinitions(path);
        // Bean实例创建流程
        DataSource dataSource = (DataSource)
beanFactory.getBean("dataSource");
        System.out.println(dataSource);
    }
```





2 基础容器的Bean实例创建流程源码阅读

3 高级容器的BeanDefinition注册流程源码阅读

二、Spring容器初始化流程源码分析

1 主流程源码分析

1.1 找入口

- java程序入口

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("spring.xml");

```

- web程序入口

```xml
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring.xml</param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>

```

> 注意：不管上面哪种方式，最终都会调`AbstractApplicationContext的refresh方法`，而这个方法才是我们真正的入口。

1.2 流程解析

- AbstractApplicationContext的`refresh`方法

```java
public void refresh() throws BeansException, IllegalStateException {
        synchronized (this.startupShutdownMonitor) {
            // Prepare this context for refreshing.
            // STEP 1:  刷新预处理
            prepareRefresh();

            // Tell the subclass to refresh the internal bean factory.
            // STEP 2:
            //      a） 创建IoC容器（DefaultListableBeanFactory）
            //      b） 加载解析XML文件（最终存储到Document对象中）
            //      c） 读取Document对象，并完成BeanDefinition的加载和注册工作
            ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();

            // Prepare the bean factory for use in this context.
            // STEP 3:  对IoC容器进行一些预处理（设置一些公共属性）
            prepareBeanFactory(beanFactory);

            try {
                // Allows post-processing of the bean factory in context
subclasses.
                // STEP 4:
                postProcessBeanFactory(beanFactory);

                // Invoke factory processors registered as beans in the
context.
                // STEP 5:  调用BeanFactoryPostProcessor后置处理器对
BeanDefinition处理
                invokeBeanFactoryPostProcessors(beanFactory);

                // Register bean processors that intercept bean creation.
                // STEP 6:  注册BeanPostProcessor后置处理器
                registerBeanPostProcessors(beanFactory);

                // Initialize message source for this context.
                // STEP 7:  初始化一些消息源（比如处理国际化的i18n等消息源）
                initMessageSource();
```

```
34
35                  // Initialize event multicaster for this context.
36                  // STEP 8:  初始化应用事件广播器
37                  initApplicationEventMulticaster();
38
39                  // Initialize other special beans in specific context
    subclasses.
40                  // STEP 9:  初始化一些特殊的bean
41                  onRefresh();
42
43                  // Check for listener beans and register them.
44                  // STEP 10:  注册一些监听器
45                  registerListeners();
46
47                  // Instantiate all remaining (non-lazy-init) singletons.
48                  // STEP 11:  实例化剩余的单例bean（非懒加载方式）
49                  // 注意事项：Bean的IoC、DI和AOP都是发生在此步骤
50                  finishBeanFactoryInitialization(beanFactory);
51
52                  // Last step: publish corresponding event.
53                  // STEP 12:  完成刷新时，需要发布对应的事件
54                  finishRefresh();
55              }
56
57              catch (BeansException ex) {
58                  if (logger.isWarnEnabled()) {
59                      logger.warn("Exception encountered during context
    initialization - " +
60                              "cancelling refresh attempt: " + ex);
61                  }
62
63                  // Destroy already created singletons to avoid dangling
    resources.
64                  destroyBeans();
65
66                  // Reset 'active' flag.
67                  cancelRefresh(ex);
68
69                  // Propagate exception to caller.
70                  throw ex;
71              }
72
73              finally {
74                  // Reset common introspection caches in Spring's core,
    since we
75                  // might not ever need metadata for singleton beans
    anymore...
76                  resetCommonCaches();
77              }
78          }
79      }
80
```

2 创建BeanFactory流程源码分析

2.1 找入口

AbstractApplicationContext类的`refresh`方法：

```
// Tell the subclass to refresh the internal bean factory.
// STEP 2:
//      a） 创建IoC容器（DefaultListableBeanFactory）
//      b） 加载解析XML文件（最终存储到Document对象中）
//      c） 读取Document对象，并完成BeanDefinition的加载和注册工作
ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

```

2.2 流程解析

- 进入AbstractApplication的`obtainFreshBeanFactory`方法：

  用于创建一个新的`IoC容器`，这个`IoC容器`就是DefaultListableBeanFactory对象。

```
    protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
        // 主要是通过该方法完成IoC容器的刷新
        refreshBeanFactory();
        ConfigurableListableBeanFactory beanFactory = getBeanFactory();
        if (logger.isDebugEnabled()) {
            logger.debug("Bean factory for " + getDisplayName() + ": " +
beanFactory);
        }
        return beanFactory;
    }

```

- 进入AbstractRefreshableApplicationContext的`refreshBeanFactory`方法：
  - 销毁以前的容器
  - 创建新的`IoC容器`
  - 加载`BeanDefinition`对象注册到IoC容器中

```
protected final void refreshBeanFactory() throws BeansException {
        // 如果之前有IoC容器，则销毁
        if (hasBeanFactory()) {
            destroyBeans();
            closeBeanFactory();
        }
        try {
            // 创建IoC容器，也就是DefaultListableBeanFactory
            DefaultListableBeanFactory beanFactory = createBeanFactory();
            beanFactory.setSerializationId(getId());
            customizeBeanFactory(beanFactory);
            // 加载BeanDefinition对象，并注册到IoC容器中（重点）
            loadBeanDefinitions(beanFactory);
            synchronized (this.beanFactoryMonitor) {
                this.beanFactory = beanFactory;
            }
        }
        catch (IOException ex) {
```

```
            throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);
        }
    }

```

- 进入AbstractRefreshableApplicationContext的`createBeanFactory`方法

```
    protected DefaultListableBeanFactory createBeanFactory() {
        return new DefaultListableBeanFactory(getInternalParentBeanFactory());
    }

```

3 加载BeanDefinition流程分析

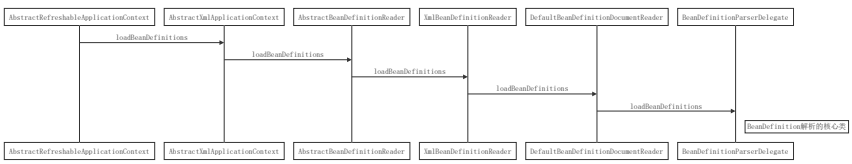
3.1 找入口

AbstractRefreshableApplicationContext类的`refreshBeanFactory`方法中第13行代码：

```
protected final void refreshBeanFactory() throws BeansException {
        // 如果之前有IoC容器，则销毁
        if (hasBeanFactory()) {
            destroyBeans();
            closeBeanFactory();
        }
        try {
            // 创建IoC容器，也就是DefaultListableBeanFactory
            DefaultListableBeanFactory beanFactory = createBeanFactory();
            beanFactory.setSerializationId(getId());
            customizeBeanFactory(beanFactory);
            // 加载BeanDefinition对象，并注册到IoC容器中（重点）
            loadBeanDefinitions(beanFactory);
            synchronized (this.beanFactoryMonitor) {
                this.beanFactory = beanFactory;
            }
        }
        catch (IOException ex) {
            throw new ApplicationContextException("I/O error parsing bean definition source for " + getDisplayName(), ex);
        }
    }

```

3.2 流程图



3.3 流程相关类的说明

- AbstractRefreshableApplicationContext

  主要用来对BeanFactory提供`refresh`功能。包括BeanFactory的创建和`BeanDefinition`的定义、解析、注册操作。

- AbstractXmlApplicationContext

  主要提供对于`XML资源`的加载功能。包括从Resource资源对象和资源路径中加载XML文件。

- AbstractBeanDefinitionReader

  主要提供对于`BeanDefinition`对象的读取功能。具体读取工作交给子类实现。

- XmlBeanDefinitionReader

  主要通过`DOM4J`对于`XML资源`的读取、解析功能，并提供对于`BeanDefinition`的注册功能。

- DefaultBeanDefinitionDocumentReader
- BeanDefinitionParserDelegate

3.4 流程解析

- 进入AbstractXmlApplicationContext的loadBeanDefinitions方法：
  - 创建一个XmlBeanDefinitionReader，通过阅读XML文件，真正完成BeanDefinition的加载和注册。
  - 配置XmlBeanDefinitionReader并进行初始化。
  - 委托给XmlBeanDefinitionReader去加载BeanDefinition。

```
    protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory)  throws BeansException, IOException {
        // Create a new XmlBeanDefinitionReader for the given BeanFactory.
        // 给指定的工厂创建一个BeanDefinition阅读器
        // 作用：通过阅读XML文件，真正完成BeanDefinition的加载和注册
        XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);

        // Configure the bean definition reader with this context's
        // resource loading environment.
        beanDefinitionReader.setEnvironment(this.getEnvironment());
        beanDefinitionReader.setResourceLoader(this);
        beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));

        // Allow a subclass to provide custom initialization of the reader,
        // then proceed with actually loading the bean definitions.
        initBeanDefinitionReader(beanDefinitionReader);

        // 委托给BeanDefinition阅读器去加载BeanDefinition
        loadBeanDefinitions(beanDefinitionReader);
    }

    protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws
                        BeansException, IOException {
        // 获取资源的定位
        // 这里getConfigResources是一个空实现，真正实现是调用子类的获取资源定位的方法
        // 比如：ClassPathXmlApplicationContext中进行了实现
        //      而FileSystemXmlApplicationContext没有使用该方法
        Resource[] configResources = getConfigResources();
        if (configResources != null) {
```

```
            // XML Bean读取器调用其父类AbstractBeanDefinitionReader读取定位的资源
            reader.loadBeanDefinitions(configResources);
        }
        // 如果子类中获取的资源定位为空，则获取FileSystemXmlApplicationContext构造方法中setConfigLocations方法设置的资源
        String[] configLocations = getConfigLocations();
        if (configLocations != null) {
            // XML Bean读取器调用其父类AbstractBeanDefinitionReader读取定位的资源
            reader.loadBeanDefinitions(configLocations);
        }
    }

```

- `loadBeanDefinitions`方法经过一路的兜兜转转，最终来到了XmlBeanDefinitionReader的`doLoadBeanDefinitions`方法：
  - `一个是对XML文件进行DOM解析；`
  - `一个是完成BeanDefinition对象的加载与注册`。

```
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
            throws BeanDefinitionStoreException {
        try {
            // 通过DOM4J加载解析XML文件，最终形成Document对象
            Document doc = doLoadDocument(inputSource, resource);
            // 通过对Document对象的操作，完成BeanDefinition的加载和注册工作
            return registerBeanDefinitions(doc, resource);
        }
        //省略一些catch语句
        catch (Throwable ex) {
            ......
        }
    }

```

- 此处我们暂不处理DOM4J加载解析XML的流程，我们重点分析BeanDefinition的加载注册流程
- 进入XmlBeanDefinitionReader的`registerBeanDefinitions`方法：
  - 创建DefaultBeanDefinitionDocumentReader用来解析Document对象。
  - 获得容器中已注册的BeanDefinition数量
  - 委托给DefaultBeanDefinitionDocumentReader来完成BeanDefinition的加载、注册工作。
  - 统计新注册的BeanDefinition数量

```
    public int registerBeanDefinitions(Document doc, Resource resource) throws
                                        BeanDefinitionStoreException {
        // 创建DefaultBeanDefinitionDocumentReader用来解析Document对象
        BeanDefinitionDocumentReader documentReader =
                                    createBeanDefinitionDocumentReader();
        // 获得容器中注册的Bean数量
        int countBefore = getRegistry().getBeanDefinitionCount();
        //解析过程入口，BeanDefinitionDocumentReader只是个接口
        //具体的实现过程在DefaultBeanDefinitionDocumentReader完成
```

```
        documentReader.registerBeanDefinitions(doc,
createReaderContext(resource));
        // 统计注册的Bean数量
        return getRegistry().getBeanDefinitionCount() - countBefore;
    }

```

- 进入DefaultBeanDefinitionDocumentReader的`registerBeanDefinitions`方法：
  - 获得Document的根元素标签
  - 真正实现BeanDefinition解析和注册工作

```
    public void registerBeanDefinitions(Document doc, XmlReaderContext
readerContext
    {
        this.readerContext = readerContext;
        logger.debug("Loading bean definitions");
        // 获得Document的根元素<beans>标签
        Element root = doc.getDocumentElement();
        // 真正实现BeanDefinition解析和注册工作
        doRegisterBeanDefinitions(root);
    }

```

- 进入DefaultBeanDefinitionDocumentReader`doRegisterBeanDefinitions`方法：
  - 这里使用了委托模式，将具体的BeanDefinition解析工作交给了BeanDefinitionParserDelegate去完成
  - 在解析Bean定义之前，进行自定义的解析，增强解析过程的可扩展性
  - 委托给BeanDefinitionParserDelegate,从Document的根元素开始进行BeanDefinition的解析
  - 在解析Bean定义之后，进行自定义的解析，增加解析过程的可扩展性

```
    protected void doRegisterBeanDefinitions(Element root) {
        // Any nested <beans> elements will cause recursion in this method.
In
        // order to propagate and preserve <beans> default-* attributes
correctly,
        // keep track of the current (parent) delegate, which may be null.
Create
        // the new (child) delegate with a reference to the parent for
fallback purposes,
        // then ultimately reset this.delegate back to its original
(parent) reference.
        // this behavior emulates a stack of delegates without actually
necessitating one.

        // 这里使用了委托模式，将具体的BeanDefinition解析工作交给了
BeanDefinitionParserDelegate去完成
        BeanDefinitionParserDelegate parent = this.delegate;
        this.delegate = createDelegate(getReaderContext(), root, parent);

        if (this.delegate.isDefaultNamespace(root)) {
            String profileSpec = root.getAttribute(PROFILE_ATTRIBUTE);
            if (StringUtils.hasText(profileSpec)) {
```

```
                String[] specifiedProfiles =
StringUtils.tokenizeToStringArray(
                        profileSpec,
BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE_DELIMITERS);
                if
(!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {
                    if (logger.isInfoEnabled()) {
                        logger.info("Skipped XML bean definition file due
to specified profiles [" + profileSpec +
                                    "] not matching: " +
getReaderContext().getResource());
                    }
                    return;
                }
            }
        }
        // 在解析Bean定义之前，进行自定义的解析，增强解析过程的可扩展性
        preProcessXml(root);
        // 委托给BeanDefinitionParserDelegate,从Document的根元素开始进行
BeanDefinition的解析
        parseBeanDefinitions(root, this.delegate);
        // 在解析Bean定义之后，进行自定义的解析，增加解析过程的可扩展性
        postProcessXml(root);

        this.delegate = parent;
    }

```

4 Bean实例化流程分析

4.1 找入口

AbstractApplicationContext类的`refresh`方法：

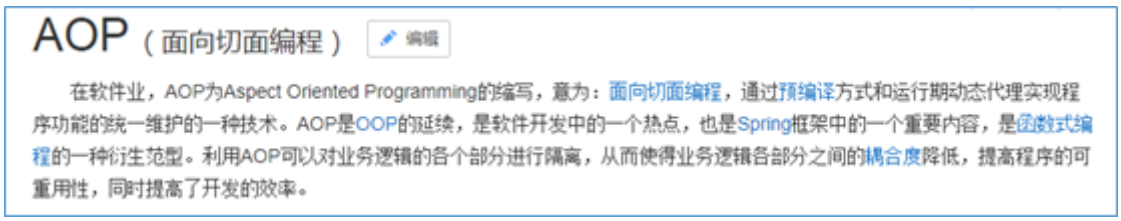
```
    // Instantiate all remaining (non-lazy-init) singletons.
    // STEP 11:  实例化剩余的单例bean（非懒加载方式）
    // 注意事项：Bean的IoC、DI和AOP都是发生在此步骤
    finishBeanFactoryInitialization(beanFactory);

```

4.4.2 流程解析

三、AOP核心概念介绍

1、什么是AOP



- 在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程
- 作用：在不修改目标类代码的前提下，可以通过AOP技术去增强目标类的功能。通过【预编译方式】或者【运行期动态代理】实现程序功能的统一维护的一种技术
    - 对目标类进行无感知的功能增强。
- AOP是一种编程范式，隶属于软工范畴，指导开发者如何组织程序结构
- AOP最早由AOP联盟的组织提出的，制定了一套规范。Spring将AOP思想引入到框架中，必须遵守AOP联盟的规范
- AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型
- 利用AOP可以对业务代码中【业务逻辑】和【系统逻辑】进行隔离，从而使得【业务逻辑】和【系统逻辑】之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

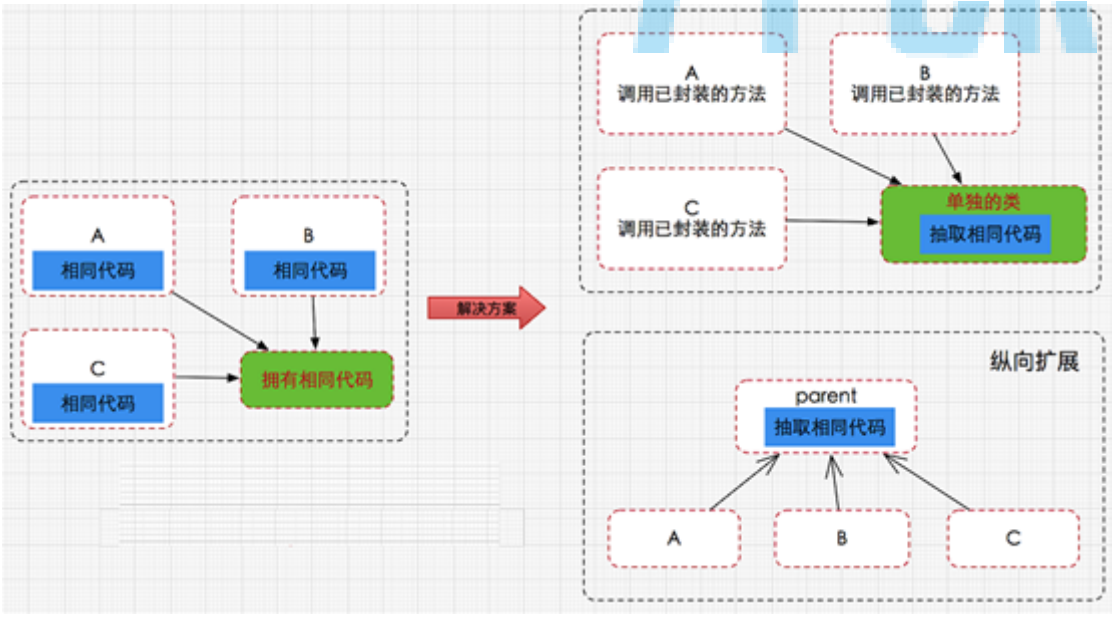
2、为什么使用AOP

- 作用：

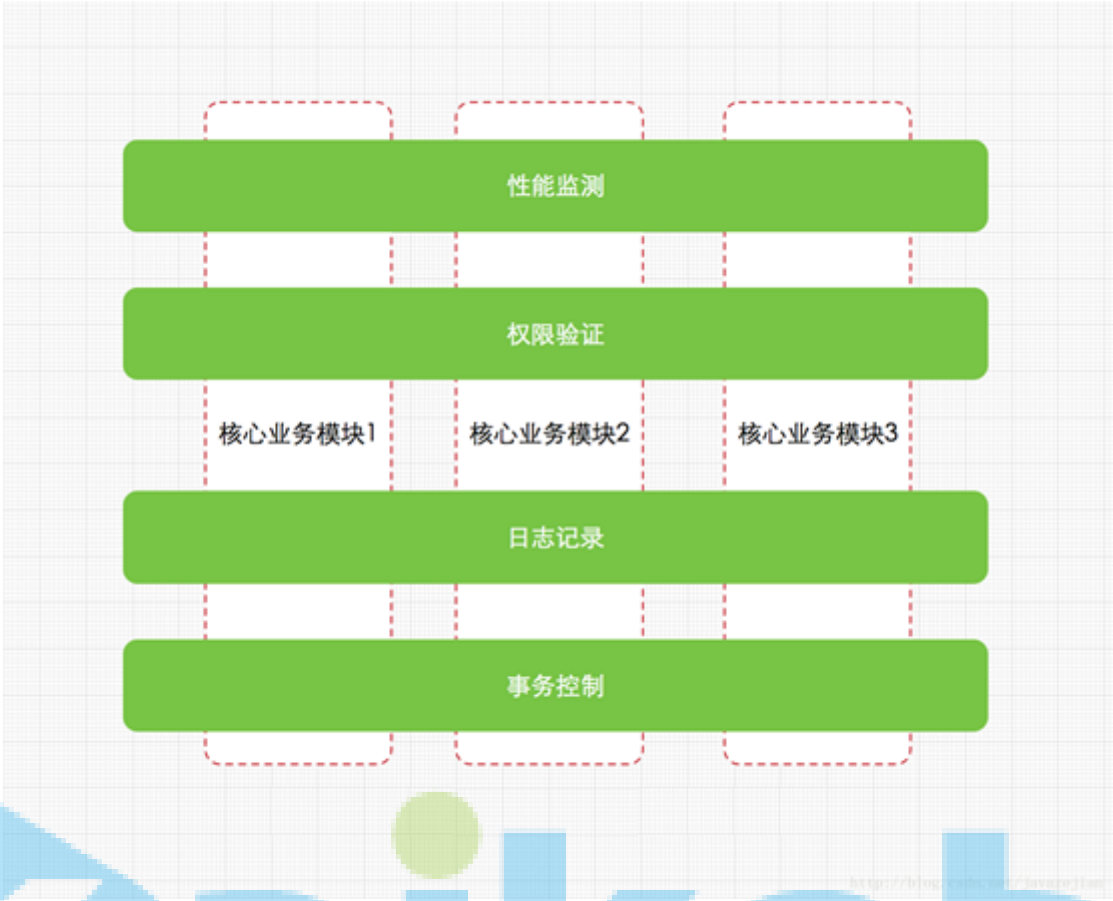
  AOP采取横向抽取机制，补充了 传统纵向继承体系（OOP）无法解决的重复性代码优化（性能监视、事务管理、安全检查、缓存），将业务逻辑和系统处理的代码（关闭连接、事务管理、操作日志记录）解耦。
- 优势：

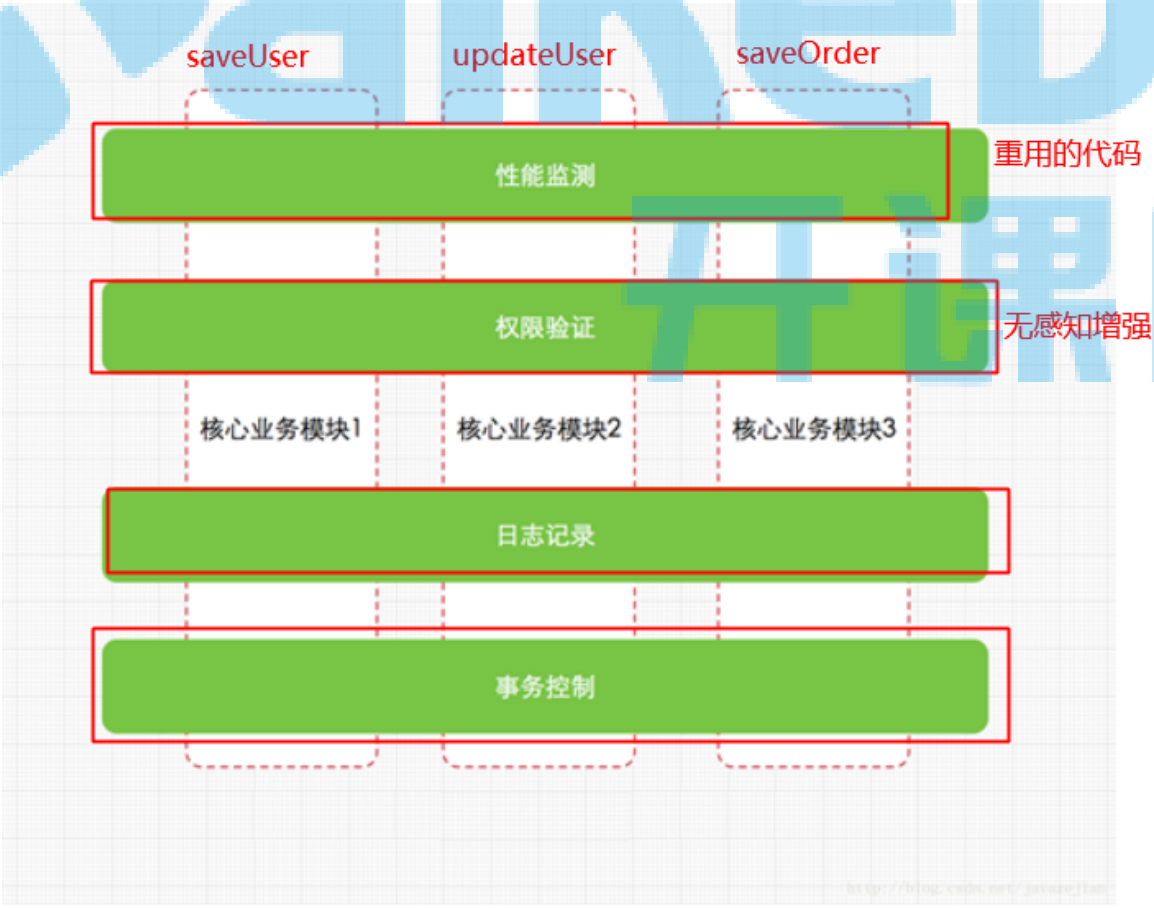
  重复性代码被抽取出来之后，维护更加方便

  不想修改原有代码前提下，可以动态横向添加共性代码。
- 纵向继承体系：



- 横向抽取机制：





3、AOP相关术语介绍

3.1 术语解释

- Joinpoint(连接点)

```
所谓连接点是指那些被拦截到的点。在spring中,这些点指的是方法,因为spring只支持方法类型的连接点
```

- Pointcut(切入点)

```
所谓切入点是指我们要对哪些Joinpoint进行拦截的定义
```

- Advice(通知/增强)

```
所谓通知是指拦截到Joinpoint之后所要做的事情就是通知.通知分为前置通知,后置通知,异常通知,最终通知,环绕通知(切面要完成的功能)
```

- Introduction(引介)

```
引介是一种特殊的通知在不修改类代码的前提下， Introduction可以在运行期为类动态地添加一些方法或Field
```

- Target(目标对象)

```
代理的目标对象
```

- Weaving(织入)

```
是指把增强应用到目标对象来创建新的代理对象的过程
```

- Proxy（代理）

```
一个类被AOP织入增强后，就产生一个结果代理类
```

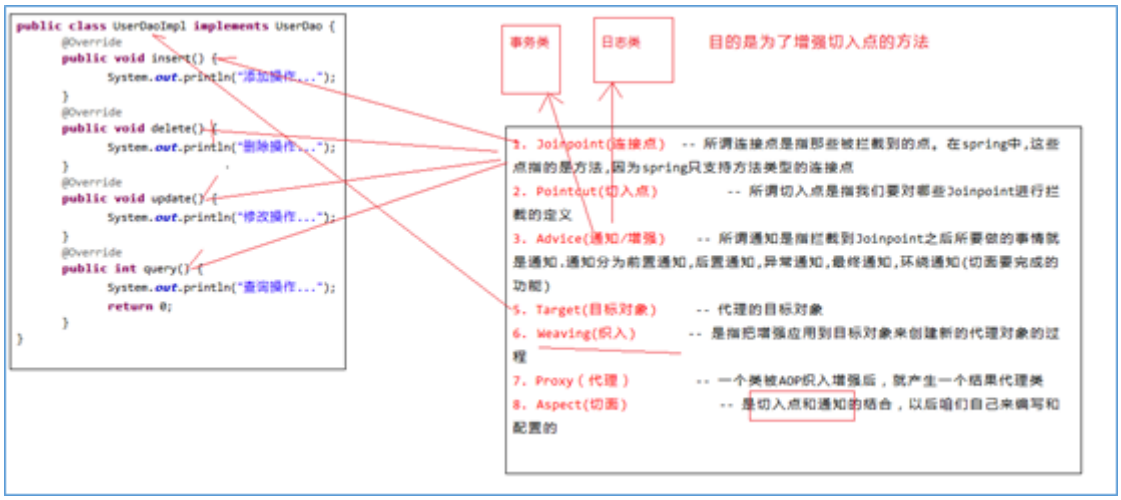
- Aspect(切面)

```
是切入点和通知的结合，以后咱们自己来编写和配置的
```

- Advisor（通知器、顾问）

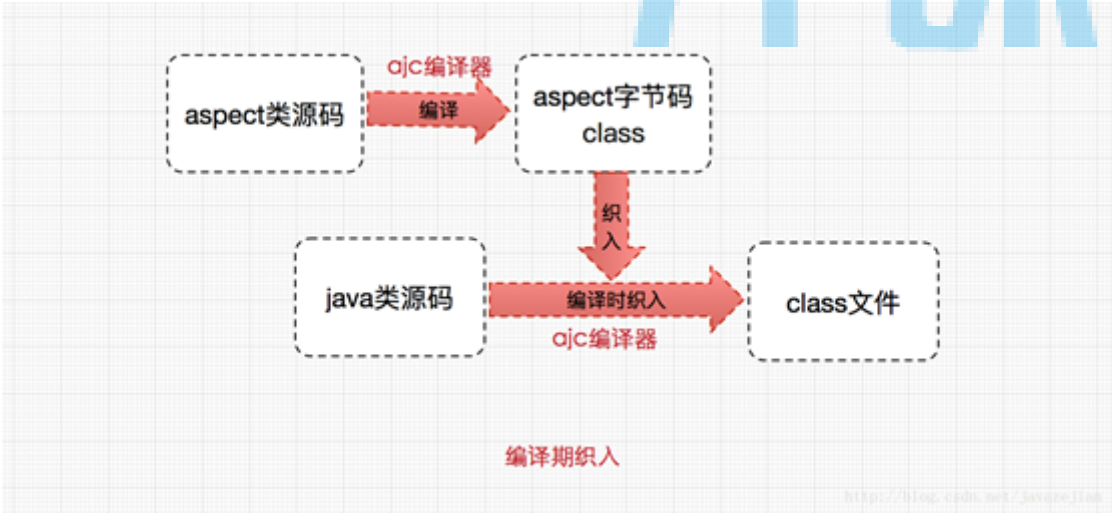
```
和Aspect很相似
```

3.2 图示说明



4、AOP实现之AspectJ（了解）

- AspectJ是一个Java实现的AOP框架，它能够对java代码进行AOP编译（一般在编译期进行），让java代码具有AspectJ的AOP功能（当然需要特殊的编译器）
- 可以这样说AspectJ是目前实现AOP框架中最成熟，功能最丰富的语言。更幸运的是，AspectJ与java程序完全兼容，几乎是无缝关联，因此对于有java编程基础的工程师，上手和使用都非常容易。
- 了解AspectJ应用到java代码的过程（这个过程称为织入），对于织入这个概念，可以简单理解为aspect(切面)应用到目标函数(类)的过程。
- 对于织入这个过程，一般分为动态织入和静态织入，动态织入的方式是在运行时动态将要增强的代码织入到目标类中，这样往往是通过动态代理技术完成的，如Java JDK的动态代理(Proxy，底层通过反射实现)或者CGLIB的动态代理(底层通过继承实现)，Spring AOP采用的就是基于运行时增强的代理技术
- ApectJ采用的就是静态织入的方式。ApectJ主要采用的是编译期织入，在这个期间使用AspectJ的acj编译器(类似javac)把aspect类编译成class字节码后，在java目标类编译时织入，即先编译aspect类再编译目标类。

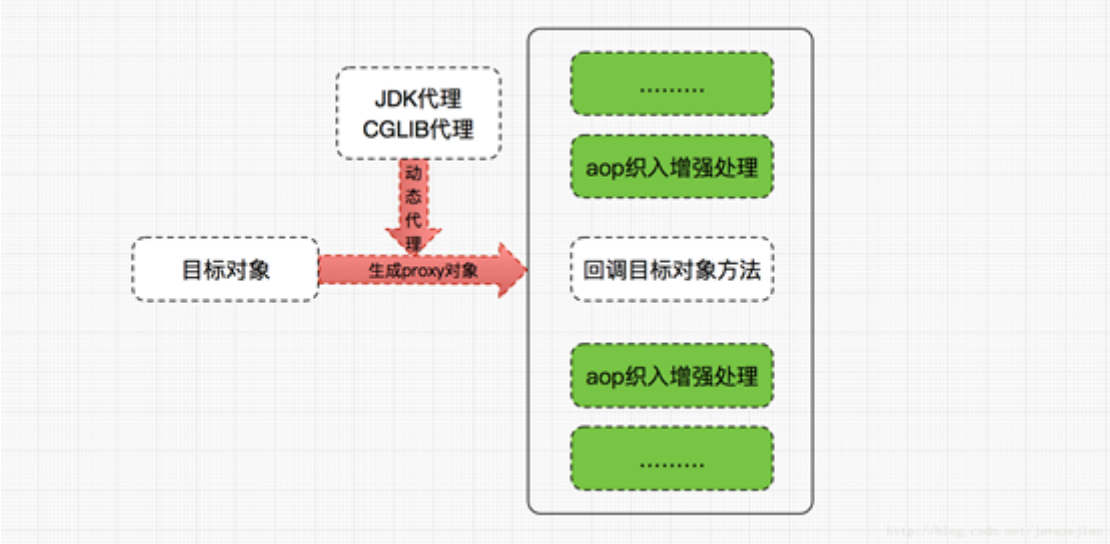


5、AOP实现之Spring AOP(了解)

5.1 实现原理分析

- Spring AOP是通过动态代理技术实现的
- 而动态代理是基于反射设计的。（关于反射的知识，请自行学习）

- 动态代理技术的实现方式有两种：基于接口的JDK动态代理和基于继承的CGLib动态代理。



1）JDK动态代理

目标对象必须实现接口

```
// JDK代理对象工厂&代理对象方法调用处理器
public class JDKProxyFactory implements InvocationHandler {

    // 目标对象的引用
    private Object target;

    // 通过构造方法将目标对象注入到代理对象中
    public JDKProxyFactory(Object target) {
        super();
        this.target = target;
    }

    /**
     * @return
     */
    public Object getProxy() {

        // 如何生成一个代理类呢？
        // 1、编写源文件
        // 2、编译源文件为class文件
        // 3、将class文件加载到JVM中(ClassLoader)
        // 4、将class文件对应的对象进行实例化（反射）

        // Proxy是JDK中的API类
        // 第一个参数：目标对象的类加载器
        // 第二个参数：目标对象的接口
        // 第二个参数：代理对象的执行处理器
        Object object =
Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(),
                this);

        return object;
    }

    /**
```

```
     * 代理对象会执行的方法
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Method method2 = target.getClass().getMethod("saveUser", null);
        Method method3 = Class.forName("com.sun.proxy.$Proxy4").getMethod("saveUser", null);
        System.out.println("目标对象的方法:" + method2.toString());
        System.out.println("目标接口的方法:" + method.toString());
        System.out.println("代理对象的方法:" + method3.toString());
        System.out.println("这是jdk的代理方法");
        // 下面的代码，是反射中的API用法
        // 该行代码，实际调用的是[目标对象]的方法
        // 利用反射，调用[目标对象]的方法
        Object returnValue = method.invoke(target, args);

        return returnValue;
    }
}
```

2）Cglib动态代理

- 目标对象不需要实现接口
- 底层是通过继承目标对象产生代理子对象（代理子对象中继承了目标对象的方法，并可以对该方法进行增强）

```
public class CgLibProxyFactory implements MethodInterceptor {

    /**
     * @param clazz
     * @return
     */
    public Object getProxyByCgLib(Class clazz) {
        // 创建增强器
        Enhancer enhancer = new Enhancer();
        // 设置需要增强的类的类对象
        enhancer.setSuperclass(clazz);
        // 设置回调函数
        enhancer.setCallback(this);
        // 获取增强之后的代理对象
        return enhancer.create();
    }

    /***
     * Object proxy:这是代理对象，也就是[目标对象]的子类
     * Method method:[目标对象]的方法
     * Object[] arg:参数
     * MethodProxy methodProxy: 代理对象的方法
     */
    @Override
    public Object intercept(Object proxy, Method method, Object[] arg, MethodProxy methodProxy) throws Throwable {
        // 因为代理对象是目标对象的子类
        // 该行代码，实际调用的是父类目标对象的方法
```

```
        System.out.println("这是cglib的代理方法");

        // 通过调用子类[代理类]的invokeSuper方法，去实际调用[目标对象]的方法
        Object returnValue = methodProxy.invokeSuper(proxy, arg);
        // 代理对象调用代理对象的invokeSuper方法，而invokeSuper方法会去调用目标类的invoke方法完成目标对象的调用

        return returnValue;
    }
}
```

ASM API使用：

```
ClassWriter classWriter = new ClassWriter(0);
classWriter.visit(Opcodes.V1_8, Opcodes.ACC_PUBLIC, className, null,
        "java/lang/Object", null);
MethodVisitor initVisitor = classWriter.visitMethod(Opcodes.ACC_PUBLIC, "<init>",
        "()V", null, null);
initVisitor.visitCode();//访问开始
initVisitor.visitVarInsn(Opcodes.ALOAD, 0);//this指针入栈
initVisitor.visitMethodInsn(Opcodes.INVOKESPECIAL, "java/lang/Object", "<init>",
        "()V");//调用构造函数
initVisitor.visitInsn(Opcodes.RETURN);
initVisitor.visitMaxs(1, 1);//设置栈长、本地变量数

```

5.2 使用

- 其使用ProxyFactoryBean创建：
- 使用 `<aop:advisor>` 定义通知器的方式实现AOP则需要通知类实现Advice接口
- 增强（通知）的类型有：

```
- 前置通知：org.springframework.aop.MethodBeforeAdvice

- 后置通知：org.springframework.aop.AfterReturningAdvice

- 环绕通知：org.aopalliance.intercept.MethodInterceptor

- 异常通知：org.springframework.aop.ThrowsAdvice
```

四、基于AspectJ的AOP使用

其实就是指的Spring + AspectJ整合，不过Spring已经将AspectJ收录到自身的框架中了，并且底层织入依然是采取的动态织入方式。

1 添加依赖

```
        <!-- 基于AspectJ的aop依赖 -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>aopalliance</groupId>
            <artifactId>aopalliance</artifactId>
            <version>1.0</version>
        </dependency>

```

2 编写目标类和目标方法

编写接口和实现类（目标对象）

```
UserService接口

UserServiceImpl实现类
```

配置目标类，将目标类交给spring IoC容器管理

```
<context:component-scan base-package="sourcecode.ioc" />
```

3 使用XML实现

1）实现步骤

编写通知（增强类，一个普通的类）

```
public class MyAdvice {

    public void log(){
        System.out.println("记录日志...");
    }
}
```

配置通知，将通知类交给spring IoC容器管理

```
<!-- 配置通知、增强 -->
<bean name="myAdvice" class="cn.spring.advice.MyAdvice"></bean>
```

配置AOP 切面

```xml
<!-- 配置通知、增强 -->
<bean name="myAdvice" class="cn.spring.advice.MyAdvice"></bean>

<!-- AOP配置 -->
<aop:config>
    <aop:aspect ref="myAdvice">
        <!-- method：指定要增强的方法，也就是指定通知类中的增强功能方法 -->
        <!-- pointcut：指定切入点，需要通过表达式来指定 -->
        <aop:before method="log"
            pointcut="execution(void cn.spring.dao.UserDaoImpl.insert())" />
    </aop:aspect>
</aop:config>
```

2）切入点表达式

切入点表达式的格式：

```
execution([修饰符] 返回值类型 包名.类名.方法名(参数))
```

表达式格式说明：

- execution：必须要
- 修饰符：可省略
- 返回值类型：必须要，但是可以使用*通配符
- 包名 ：

```
** 多级包之间使用.分割

** 包名可以使用*代替，多级包名可以使用多个*代替

** 如果想省略中间的包名可以使用 ..
```

- 类名

```
** 可以使用*代替

** 也可以写成*DaoImpl
```

- 方法名：

```
** 也可以使用*好代替

** 也可以写成add*
```

- 参数：

```
** 参数使用*代替

** 如果有多个参数，可以使用 ..代替
```

3）通知类型

通知类型（五种）：前置通知、后置通知、最终通知、环绕通知、异常抛出通知。

前置通知：

```
    * 执行时机：目标对象方法之前执行通知
    * 配置文件：<aop:before method="before" pointcut-ref="myPointcut"/>
    * 应用场景：方法开始时可以进行校验
```

后置通知：

```
    * 执行时机：目标对象方法之后执行通知，有异常则不执行了
    * 配置文件：<aop:after-returning method="afterReturning" pointcut-ref="myPointcut"/>
    * 应用场景：可以修改方法的返回值
```

最终通知：

```
    * 执行时机：目标对象方法之后执行通知，有没有异常都会执行
    * 配置文件：<aop:after method="after" pointcut-ref="myPointcut"/>
    * 应用场景：例如像释放资源
```

环绕通知：

```
    * 执行时机：目标对象方法之前和之后都会执行。
    * 配置文件：<aop:around method="around" pointcut-ref="myPointcut"/>
    * 应用场景：事务、统计代码执行时机
```

异常抛出通知：

```
    * 执行时机：在抛出异常后通知
    * 配置文件：<aop:after-throwing method=" afterThrowing " pointcut-ref="myPointcut"/>
    * 应用场景：包装异常
```

4 使用注解实现

1）实现步骤

编写切面类（注意不是通知类，因为该类中可以指定切入点）

```java
/**
 * 切面类（通知+切入点）
 *
 * @author think
 *
 */
// @Aspect：标记该类是一个切面类
@Component("myAspect")
@Aspect
public class MyAspect {

    //@Before：标记该方法是一个前置通知
    //value：切入点表达式
    @Before(value = "execution(* *..*.*DaoImpl.*(..))")
    public void log() {
        System.out.println("记录日志...");
    }
}
```

配置切面类

```xml
<context:component-scan base-package="com.kkb.spring"/>
```

开启AOP自动代理

```xml
<!-- AOP基于注解的配置-开启自动代理 -->
<aop:aspectj-autoproxy />
```

2）环绕通知注解配置

@Around

```
    作用：
        把当前方法看成是环绕通知。属性：
    value：
        用于指定切入点表达式，还可以指定切入点表达式的引用。
```

```java
@Around(value = "execution(* *.*(..))")
public Object aroundAdvice(ProceedingJoinPoint joinPoint) {
    // 定义返回值
    Object rtValue = null;
    try {
        // 获取方法执行所需的参数
        Object[] args = joinPoint.getArgs();
        // 前置通知：开启事务beginTransaction();
        // 执行方法
        rtValue = joinPoint.proceed(args);
        // 后置通知：提交事务commit();
    } catch (Throwable e) {
        // 异常通知：回滚事务rollback(); e.printStackTrace();
    } finally {
        // 最终通知：释放资源release();
    }
    return rtValue;

}
```

3）定义通用切入点

使用@PointCut注解在切面类中定义一个通用的切入点，其他通知可以引用该切入点

```java
// @Aspect：标记该类是一个切面类
@Aspect
public class MyAspect {

    // @Before：标记该方法是一个前置通知
    // value：切入点表达式
    // @Before(value = "execution(* *..*.*DaoImpl.*(..))")
    @Before(value = "MyAspect.fn()")
    public void log() {
        System.out.println("记录日志...");
    }

    // @Before(value = "execution(* *..*.*DaoImpl.*(..))")
    @Before(value = "MyAspect.fn()")
    public void validate() {
        System.out.println("进行后台校验...");
    }

    // 通过@Pointcut定义一个通用的切入点
    @Pointcut(value = "execution(* *..*.*DaoImpl.*(..))")
    public void fn() {
    }
}
```

5 纯注解方式

```
@Configuration
@ComponentScan(basePackages="com.kkb")
@EnableAspectJAutoProxy
public class SpringConfiguration {
}
```

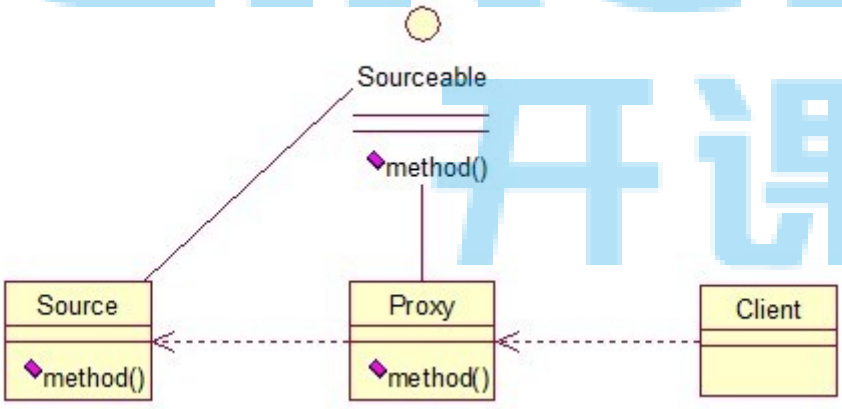
五、代理模式

其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作。代理又分为动态代理和静态代理

1 静态代理

> 比如我们在租房子的时候回去找中介，为什么呢？
>
> 因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。
>
> 再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。

先来看看关系图：



根据上文的阐述，代理模式就比较容易的理解了，我们看下代码：

```
public interface Sourceable {
    public void method();
}
```

```java
public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}
```

```java
public class Proxy implements Sourceable {
    private Source source;
    public Proxy(){
        super();
        this.source = new Source();
    }
    @Override
    public void method() {
        before();
        source.method();
        atfer();
    }
    private void atfer() {
        System.out.println("after proxy!");
    }
    private void before() {
        System.out.println("before proxy!");
    }
}
```

测试类：

```java
public class ProxyTest {
    public static void main(String[] args) {
        Sourceable source = new Proxy();
        source.method();
    }
}
```

输出：

```
before proxy!
the original method!
after proxy!
```

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1. 修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。
2. 就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

2 动态代理

JDK动态代理和Cglib动态代理的区别：

> 1.JDK动态代理是Java自带的，cglib动态代理是第三方jar包提供的。
> 2.JDK动态代理是针对【拥有接口的目标类】进行动态代理的，而Cglib是非final类都可以进行动态代理。但是Spring优先使用JDK动态代理。
> 3.JDK动态代理实现的逻辑是目标类和代理类都实现同一个接口，目标类和代理类是平级的。而Cglib动态代理实现的逻辑是给目标类生个孩子（子类，也就是代理类），目标类和代理类是父子继承关系。
> 4.JDK动态代理在早期的JDK1.6左右性能比cglib差，但是在JDK1.8以后cglib和jdk的动态代理性能基本上差不多。反而jdk动态代理性能更加的优越。

动态代理主要关注两个点：

> 代理对象如何创建的底层原理？
>
> 代理对象如何执行的原理分析？

1）JDK动态代理

基于接口去实现的动态代理

```
// JDK代理对象工厂&代理对象方法调用处理器
public class JDKProxyFactory implements InvocationHandler {

    // 目标对象的引用
    private Object target;

    // 通过构造方法将目标对象注入到代理对象中
    public JDKProxyFactory(Object target) {
        super();
        this.target = target;
    }

    /**
     * @return
     */
    public Object getProxy() {

        // 如何生成一个代理类呢？
        // 1、编写源文件
        // 2、编译源文件为class文件
        // 3、将class文件加载到JVM中(ClassLoader)
        // 4、将class文件对应的对象进行实例化（反射）

        // Proxy是JDK中的API类
        // 第一个参数：目标对象的类加载器
        // 第二个参数：目标对象的接口
        // 第二个参数：代理对象的执行处理器
        Object object =
Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(),
                this);
```

```
30
31          return object;
32      }
33
34      /**
35       * 代理对象会执行的方法
36       */
37      @Override
38      public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
39          Method method2 = target.getClass().getMethod("saveUser", null);
40          Method method3 =
Class.forName("com.sun.proxy.$Proxy4").getMethod("saveUser", null);
41          System.out.println("目标对象的方法:" + method2.toString());
42          System.out.println("目标接口的方法:" + method.toString());
43          System.out.println("代理对象的方法:" + method3.toString());
44          System.out.println("这是jdk的代理方法");
45          // 下面的代码，是反射中的API用法
46          // 该行代码，实际调用的是[目标对象]的方法
47          // 利用反射，调用[目标对象]的方法
48          Object returnValue = method.invoke(target, args);
49
50          return returnValue;
51      }
52  }
```

2）CGLib动态代理(ASM)

是通过子类继承父类的方式去实现的动态代理，不需要接口。

```
1   public class CgLibProxyFactory implements MethodInterceptor {
2
3       /**
4        * @param clazz
5        * @return
6        */
7       public Object getProxyByCgLib(Class clazz) {
8           // 创建增强器
9           Enhancer enhancer = new Enhancer();
10          // 设置需要增强的类的类对象
11          enhancer.setSuperclass(clazz);
12          // 设置回调函数
13          enhancer.setCallback(this);
14          // 获取增强之后的代理对象
15          return enhancer.create();
16      }
17
18      /***
19       * Object proxy:这是代理对象，也就是[目标对象]的子类
20       * Method method:[目标对象]的方法
21       * Object[] arg:参数
22       * MethodProxy methodProxy：代理对象的方法
23       */
24      @Override
```

```
    public Object intercept(Object proxy, Method method, Object[] arg,
MethodProxy methodProxy) throws Throwable {
        // 因为代理对象是目标对象的子类
        // 该行代码，实际调用的是父类目标对象的方法
        System.out.println("这是cglib的代理方法");

        // 通过调用子类[代理类]的invokeSuper方法，去实际调用[目标对象]的方法
        Object returnValue = methodProxy.invokeSuper(proxy, arg);
        // 代理对象调用代理对象的invokeSuper方法，而invokeSuper方法会去调用目标类的
invoke方法完成目标对象的调用

        return returnValue;
    }
}
```

##

1.基础容器源码阅读：基于XML的BeanDefinition注册流程

入口1：XmlBeanFactory#构造方法

|--【XmlBeanDefinitionReader】#loadBeanDefinitions

|--#doLoadBeanDefinitions

|--#doLoadDocument

|--#registerBeanDefinitions

|--【DefaultBeanDefinitionDocumentReader】#registerBeanDefinitions

|--#doRegisterBeanDefinitions

|--#parseBeanDefinitions

|--【BeanDefinitionParserDelegate】#parseCustomElement：解析自定义标签

|--#parseDefaultElement：解析不带有冒号的标签，比如说bean标签

|--#processBeanDefinition

|--【BeanDefinitionParserDelegate】#parseBeanDefinitionElement：经过几个重载方法

|--#createBeanDefinition：创建出来【GenericBeanDefinition】

入口2：XmlBeanDefinitionReader#loadBeanDefinitions

2.高级容器源码阅读：基于注解的BeanDefinition注册流程

入口：AnnotationConfigApplicationContext#构造方法

|--#scan

|--ClassPathBeanDefinitionScanner#scan：扫描classpath路径下的注解（@Controller、@Component、@Repository、@Service）

|--#doScan

|--【ClassPathScanningCandidateComponentProvider】#scanCandidateComponents

|--【ScannedGenericBeanDefinition】

3.高级容器源码阅读

入口：ClassPathXmlApplicationContext#构造方法

|--【AbstractApplicationContext】#refresh

|--【AbstractRefreshableApplicationContext】#refreshBeanFactory

```
        |--#createBeanFactory:【DefaultListableBeanFactory】
        |--【AbstractXmlApplicationContext】#loadBeanDefinitions
            |--【XmlBeanDefinitionReader】#loadBeanDefinitions
```

4.基础容器：创建bean实例流程

```
入口：AbstractBeanFactory#getBean
    |--#doGetBean
        |--【DefaultSingletonBeanRegistry】#getSingleton
        |--#getObjectForBeanInstance
        |--#getMergedLocalBeanDefinition
        |--【AbstractAutowireCapableBeanFactory】#createBean
            |--#doCreateBean
                |--#createBeanInstance：Bean的实例化
                    |--#obtainFromSupplier
                    |--#instantiateUsingFactoryMethod
                    |--#instantiateBean
                |--#populateBean：依赖注入
                |--#initializeBean：Bean的初始化
```

```
1  //springmvc：页面参数传递
2
3  user.id = &user.name=
4
5  user.name
6  orders[0]
7  orders["key"]
```