a. 连接点（Joinpoint）：程序执行的某个特定位置（如：某个方法调用前、调用后，方法抛出异常后）。一个类或一段程序代码拥有一些具有边界性质的特定点，这些代码中的特定点就是连接点。Spring仅支持方法的连接点。

b. 切点（Pointcut）：如果连接点相当于数据中的记录，那么切点相当于查询条件，一个切点可以匹配多个连接点。Spring AOP的规则解析引擎负责解析切点所设定的查询条件，找到对应的连接点。

c. 增强（Advice）：增强是织入到目标类连接点上的一段程序代码。Spring提供的增强接口都是带方位名的，比如：BeforeAdvice、AfterReturningAdvice、ThrowsAdvice等。很多资料上将增强译为“通知”，这明显是个词不达意的翻译，让很多程序员困惑了许久。

说明： Advice在国内的很多书面资料中都被翻译成"通知"，但是很显然这个翻译无法表达其本质，有少量的读物上将这个词翻译为"增强"，这个翻译是对Advice较为准确的诠释，我们通过AOP将横切关注功能加到原有的业务逻辑上，这就是对原有业务逻辑的一种增强，这种增强可以是前置增强、后置增强、返回后增强、抛异常时增强和包围型增强。

d. 引介（Introduction）：引介是一种特殊的增强，它为类添加一些属性和方法。这样，即使一个业务类原本没有实现某个接口，通过引介功能，可以动态的为该业务类添加接口的实现逻辑，让业务类成为这个接口的实现类。

e. 织入（Weaving）：织入是将增强添加到目标类具体连接点上的过程，AOP有三种织入方式：①编译期织入：需要特殊的Java编译器（例如AspectJ的ajc）；②装载期织入：要求使用特殊的类加载器，在装载类的时候对类进行增强；③运行时织入：在运行时为目标类生成代理实现增强。Spring采用了动态代理的方式实现了运行时织入，而AspectJ采用了编译期织入和装载期织入的方式。

f. 切面（Aspect）：切面是由切点和增强（引介）组成的，它包括了对横切关注功能的定义，也包括了对连接点的定义。

补充：代理模式是GoF提出的23种设计模式中最为经典的模式之一，代理模式是对象的结构模式，它给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。简单的说，代理对象可以完成比原对象更多的职责，当需要为原对象添加横切关注功能时，就可以使用原对象的代理对象。我们在打开Office系列的Word文档时，如果文档中有插图，当文档刚加载时，文档中的插图都只是一个虚框占位符，等用户真正翻到某页要查看该图片时，才会真正加载这张图，这其实就是对代理模式的使用，代替真正图片的虚框就是一个虚拟代理；Hibernate的load方法也是返回一个虚拟代理对象，等用户真正需要访问对象的属性时，才向数据库发出SQL语句获得真实对象。

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

从JDK 1.3开始，Java提供了动态代理技术，允许开发者在运行时创建接口的代理实例，主要包括Proxy类和InvocationHandler接口。下面的例子使用动态代理为ArrayList编写一个代理，在添加和删除元素时，在控制台打印添加或删除的元素以及ArrayList的大小：

package chimomo.learning.java.designpattern.proxy.listproxy;

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.util.List;

/\*\*

\* @author Created by Chimomo

\*/

public class ListProxy<T> implements InvocationHandler {

private List<T> target;

**public ListProxy(List<T> target) {**

**this.target = target;**

**}**

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

Object retVal = null;

System.out.println("[" + method.getName() + ": " + args[0] + "]");

retVal = method.invoke(target, args);

System.out.println("[size=" + target.size() + "]");

return retVal;

}

package chimomo.learning.java.designpattern.proxy.listproxy;

import java.lang.reflect.Proxy;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* @author Created by Chimomo

\*/

public class ListProxyTest {

@SuppressWarnings("unchecked")

public static void main(String[] args) {

List<String> list = new ArrayList<>();

Class<?> clazz = list.getClass();

ListProxy<String> myProxy = new ListProxy<>(list);

**List<String> newList = (List<String>) Proxy.newProxyInstance(clazz.getClassLoader(), clazz.getInterfaces(), myProxy);**

newList.add("apple");

newList.add("banana");

newList.add("orange");

newList.remove("banana");

}

}

// Output:

/\*

[add: apple]

[size=1]

[add: banana]

[size=2]

[add: orange]

[size=3]

[remove: banana]

[size=2]

\*/

说明：使用Java的动态代理有一个局限性就是代理的类必须要实现接口，虽然面向接口编程是每个优秀的Java程序都知道的规则，但现实往往不尽如人意，对于没有实现接口的类如何为其生成代理呢？继承！继承是最经典的扩展已有代码能力的手段，虽然继承常常被初学者滥用，但继承也常常被进阶的程序员忽视。**CGLib采用非常底层的字节码生成技术，通过为一个类创建子类来生成代理，它弥补了Java动态代理的不足**，**因此Spring中动态代理和CGLib都是创建代理的重要手段，对于实现了接口的类就用动态代理为其生成代理类，而没有实现接口的类就用CGLib通过继承的方式为其创建代理。**

**Source link:**<https://blog.csdn.net/troubleshooter/article/details/78467637>

Java中代理的实现一般分为三种：JDK静态代理、JDK动态代理以及CGLIB动态代理。在Spring的AOP实现中，主要应用了JDK动态代理以及CGLIB动态代理。但是本文着重介绍JDK动态代理机制，CGLIB动态代理后面会接着探究。

代理一般实现的模式为JDK静态代理：创建一个接口，然后创建被代理的类实现该接口并且实现该接口中的抽象方法。之后再创建一个代理类，同时使其也实现这个接口。在代理类中持有一个被代理对象的引用，而后在代理类方法中调用该对象的方法。

其实就是代理类为被代理类预处理消息、过滤消息并在此之后将消息转发给被代理类，之后还能进行消息的后置处理。**代理类和被代理类通常会存在关联关系(即上面提到的持有的被带离对象的引用)，代理类本身不实现服务，而是通过调用被代理类中的方法来提供服务。**

我们可以看出，使用JDK静态代理很容易就完成了对一个类的代理操作。但是JDK静态代理的缺点也暴露了出来：由于代理只能为一个类服务，如果需要代理的类很多，那么就需要编写大量的代理类，比较繁琐。

To read list:

Dynamic proxy

<https://www.jianshu.com/p/471c80a7e831>