从代理机制到Spring AOP，这篇给你安排的明明白白的

好好学java 3天前

今日推荐

论异步编程的正确姿势：十个接口的活现在只需要一个接口就能搞定!
让SpringBoot不再需要Controller、Service、Mapper，这款开源工具绝了!
「吐血」我把 10 年的全部学习资源都分享在这里了
还在用Spring Security? 推荐你一款使用简单、功能强大的权限认证框架
干掉 navicat：这款 DB 管理工具才是y(永)y(远)d(的)s(神)

来源：*juejin.cn/post/6844903672061624327*

这篇文章准备从Java的代理机制讲到Spring的AOP。

1.代理模式

代理模式是很常见的一种设计模式，代理一词拆开来看就是代为受理，那显然是要涉及到请求被代理的委托方，提供代理的代理方，以及想要通过代理来实际联系委托方的客户三个角色。

举个生活中很常见的例子，各路的明星都会有个自己的经纪人来替自己打点各种各样的事情，这种场景下，明星本身是委托方，经纪人是代理方，明星把自己安排演出、出席见面会的时间安排权利委托给经纪人，这样当各个商家作为客户想要请明星来代言时，就只能通过经纪人来进行。

这样明星本身不用暴露身份，而经济人也可以在沟通中告知商家明星出席活动时要吃什么饭，做什么车的一些要求，省去了明星自己操心这些鸡毛蒜皮小事儿。另一方面，当经纪人也可以给多个明星提供服务，这样商家只接触一个经纪人，可以联系到不同的明星，找个适合自己公司的人选。

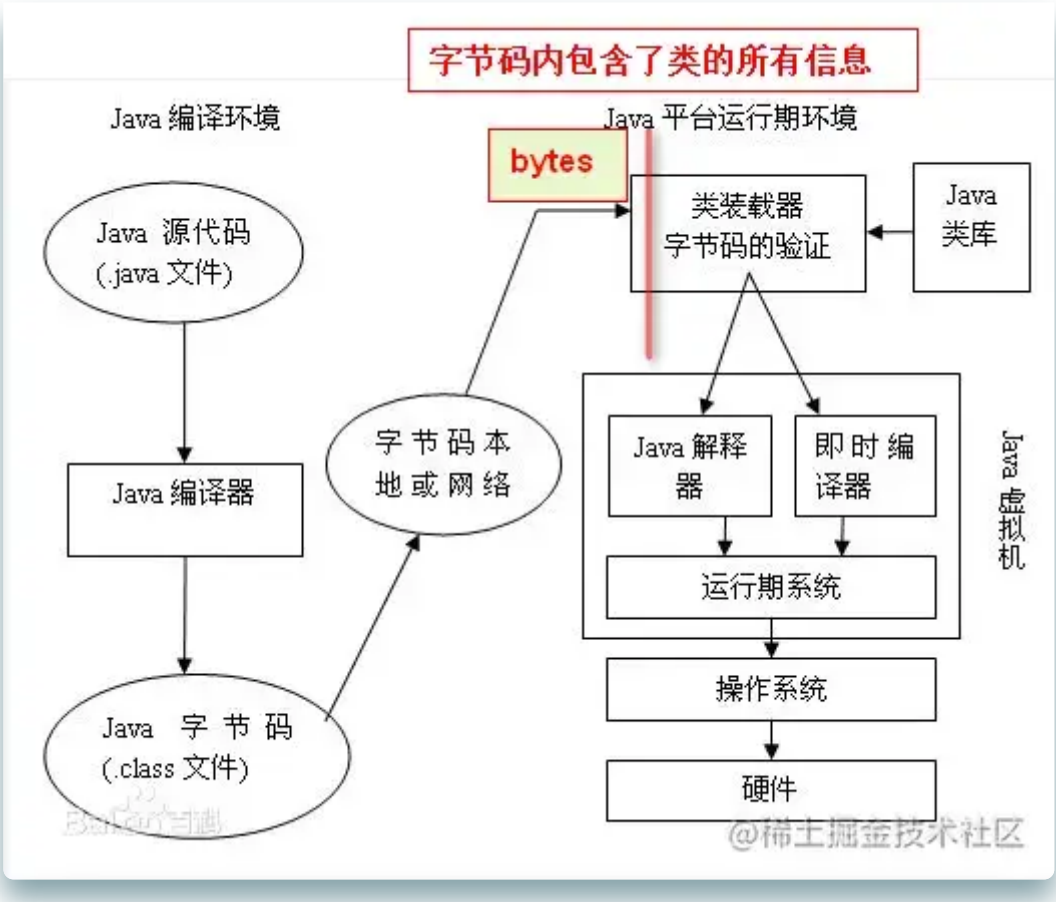
通过上面的例子，代理模式的优点就显而易见了：

优点一：可以隐藏委托类的实现；

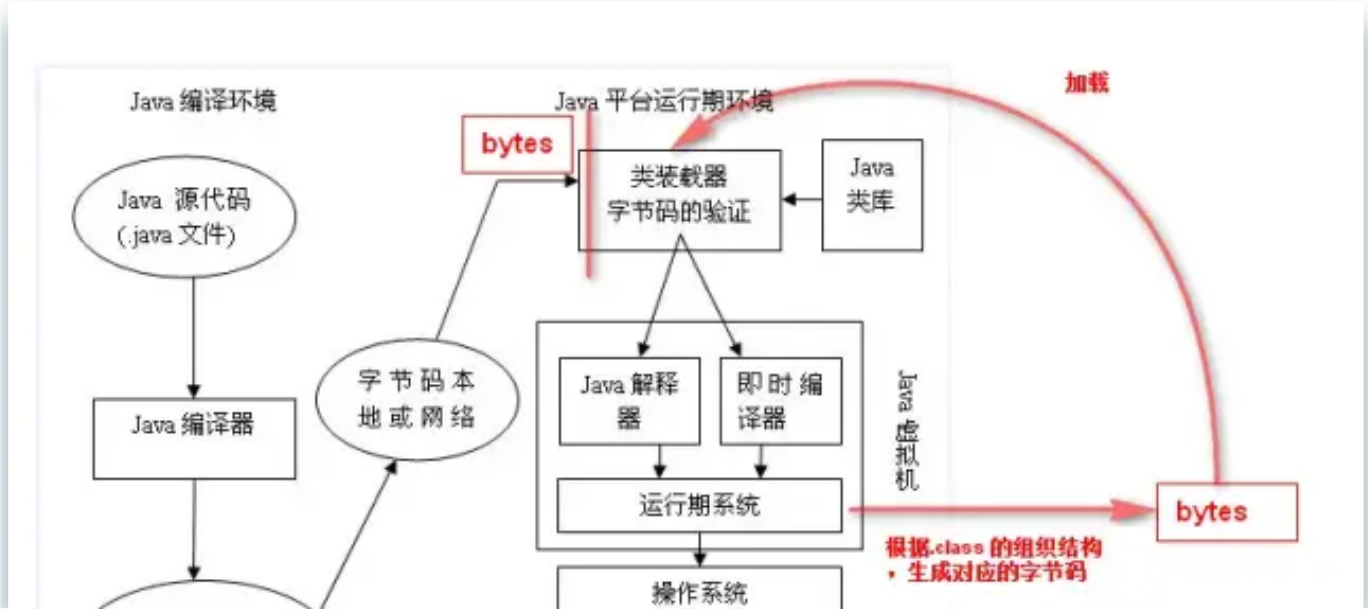
优点二：可以实现客户与委托类间的解耦，在不修改委托类代码的情况下能够做一些额外的处理。

2.字节码与代理模式

Java程序员都应该知道，Java通过Java编译器将.java源文件编译成.class字节码文件，这种.class文件是二进制文件，内容是只有JVM虚拟机能够识别的机器码，JVM虚拟机读取字节码文件，取出二进制数据，加载到内存中，解析.class文件内的信息，生成对应的Class对象，进而使Class对象创建类的具体实例来进行调用实现具体的功能。



上图说明了Java加载字节码的流程，但是Java的强大在于不仅仅可以加载在编译期生成好的字节码，还可以在运行期系统中，遵循Java编译系统组织.class文件的格式和结构，生成相应的二进制数据，然后再把这个二进制数据加载转换成对应的类，这样，就完成了在代码中，动态创建一个类的能力了，如下图流程。



在代码中动态创建类的原理图

下面举一个动态生成类的实例，通过Javassist实现，Javassist是一个开源的分析、编辑和创建Java字节码的类库，我们可以使用Javasisst工具在运行时动态创建字节码并加载类，如下代码：

```
public class JavassistDemo {

    public static void main(String[] args) {
        makeNewClass();
    }

    public static Class<?> makeNewClass() {
        try {
            // 获取ClassPool
            ClassPool pool = ClassPool.getDefault();
            // 创建Student类
            CtClass ctClass = pool.makeClass("com.fufu.aop.Student");
            // 创建Student类成员变量name
            CtField name = new CtField(pool.get("java.lang.String"), "name", ctClass);
            // 设置name为私有
            name.setModifiers(Modifier.PRIVATE);
            // 将name写入class
            ctClass.addField(name, CtField.Initializer.constant("")); //写入class文件
            //增加set方法，名字为"setName"
            ctClass.addMethod(CtNewMethod.setter("setName", name));
            //增加get方法，名字为getname
            ctClass.addMethod(CtNewMethod.getter("getName", name));
            // 添加无参的构造体
            CtConstructor cons = new CtConstructor(new CtClass[] {}, ctClass);
            cons.setBody("{name = \"Brant\";}"); //相当于public Sclass(){this.name = "brant";}
            ctClass.addConstructor(cons);
            // 添加有参的构造体
            cons = new CtConstructor(new CtClass[] {pool.get("java.lang.String")}, ctClass);
            cons.setBody("{$0.name = $1;}");  //第一个传入的形参$1,第二个传入的形参$2，相当于public
            ctClass.addConstructor(cons);
```

```
            //反射调用新创建的类
            Class<?> aClass =  ctClass .toClass();
            Object student = aClass.newInstance();
            Method getter = null;
            getter = student.getClass().getMethod("getName");
            System.out.println(getter.invoke(student));

        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

介绍静态和动态加载字节码的两种方式，是为了引出下面关于两种代理方式的介绍，代理机制通过代理类创建的时间不同分为了静态代理和动态代理：

静态代理：代理类在编译阶段生成，程序运行前就已经存在，那么这种代理方式被成为静态代理，这种情况下的代理类通常都是我们在Java代码中定义的。

动态代理：代理类在程序运行时创建，也就是说，这种情况下，代理类并不是在Java代码中定义的，而是在运行时根据我们在Java代码中的“指示”动态生成的。

目前，静态代理主要有AspectJ静态代理、JDK静态代理技术、而动态代理有JDK动态代理、Cglib动态代理技术，而Spring Aop是整合使用了JDK动态代理和Cglib动态代理两种技术，下面我们结合实例一步一步介绍所有的概念。

3.静态代理

3.1 AspectJ静态代理

对于AspectJ，我们只会进行简单的了解，为后续理解打下基础，现在只需要知道下面这一句定义：

AspectJ是一个Java实现的面向切面的框架，它扩展了Java语言。AspectJ有自定义的语法，所以它有一个专门的编译器用来生成遵守Java字节编码规范的Class文件。

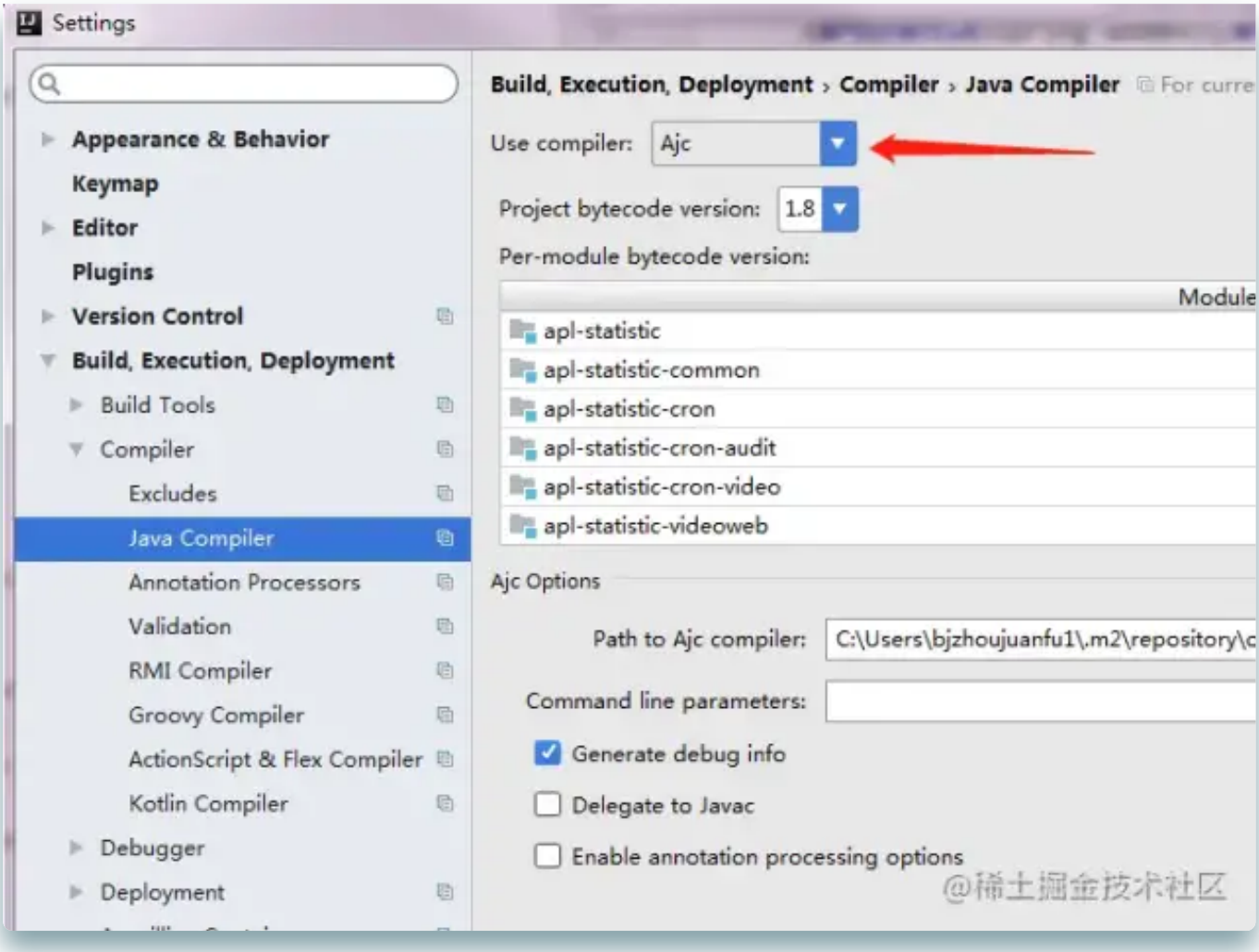
注意上面定义中的“专门的编译器”这个描述，可以看出AspectJ是典型的静态代理技术，因为是在编译时期就生成了代理类，而使用AspectJ也肯定需要指定特定的编

译器，下面我们用AspectJ来实现上面的明星和经纪人的模型。

首先在maven工程中引入AspectJ依赖：

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.8.9</version>
  </dependency>
  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjtools</artifactId>
    <version>1.8.9</version>
  </dependency>
```

然后在idea中将javac编译器改为acj编译器来支持AspectJ语法：



将明星的表演抽象成一个ShowService接口，包括了唱歌、跳舞的功能

```
public interface ShowService {
    // 歌唱表演
    void sing(String songName);
    // 舞蹈表演
    void dance();
}
```

明星类实现了ShowService接口：

```
package com.fufu.aop;

public class Star implements ShowService{
    private String name;

    @Override
    public void sing(String songName) {
        System.out.println(this.name + " sing a song: " + songName);
    }

    @Override
    public void dance() {
        System.out.println(this.name + "dance");
    }

    public Star(String name) {
        this.name = name;
    }

    public Star() {
    }

    public static void main(String[] args) {
        Star star = new Star("Eminem");
        star.sing("Mockingbird");
    }
}
```

用AspectJ语法实现一个代理AgentAspectJ：

```
package com.fufu.aop;
```

```
package com.ruru.aop;

public aspect AgentAspectJ {

    /**
     * 定义切点
     */
    pointcut sleepPointCut():call(* Star.sing(..));

    /**
     * 定义切点
     */
    pointcut eatPointCut():call(* Star.eat(..));

    /**
     * 定义前置通知
     *
     * before(参数):连接点函数{
     *      函数体
     * }
     */
    before():sleepPointCut(){
        getMoney();
    }

    /**
     * 定义后置通知
     * after(参数):连接点函数{
     *      函数体
     * }
     */
    after():sleepPointCut(){
        writeReceipt();
    }

    private void getMoney() {
        System.out.println("get money");
    }

    private void writeReceipt() {
        System.out.println("write receipt");
    }
}
```

创建一个Star并运行方法：

```
public static void main(String[] args) {
        Star star = new Star("Eminem");
        star.sing("Mockingbird");
    }
```

输出：

```
get money
Eminem sing a song: Mockingbird
write receipt
```

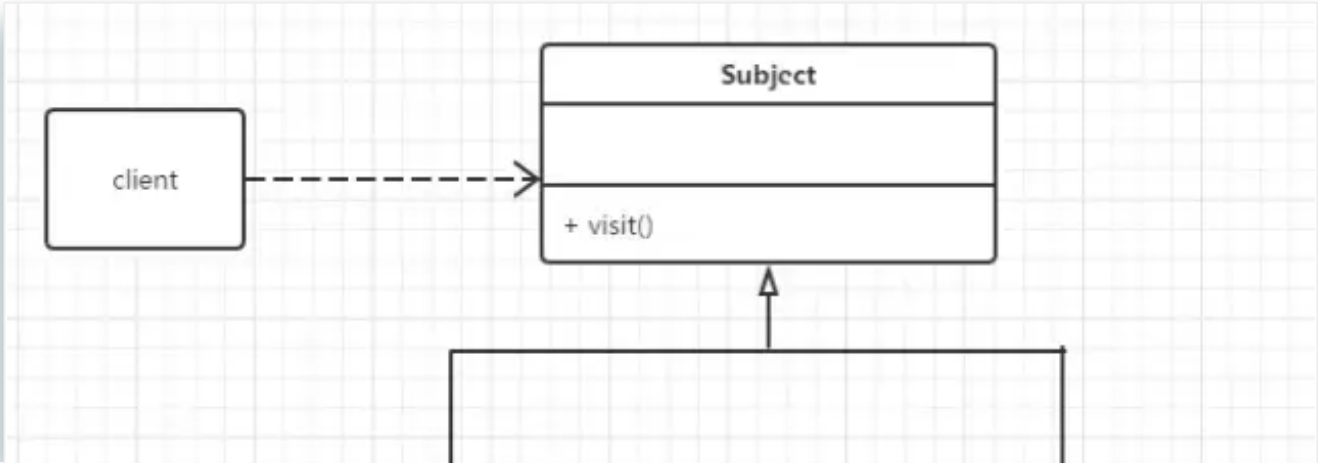
可以看到Star的sing()方法前后输出了我们在AgentAspectJ中定义的前置通知和后置通知，所以是AspectJ在编译期间，根据AgentAspectJ代码中定义的代码，生成了增强的Star类，而我们实际调用时，就会实现代理类的功能。

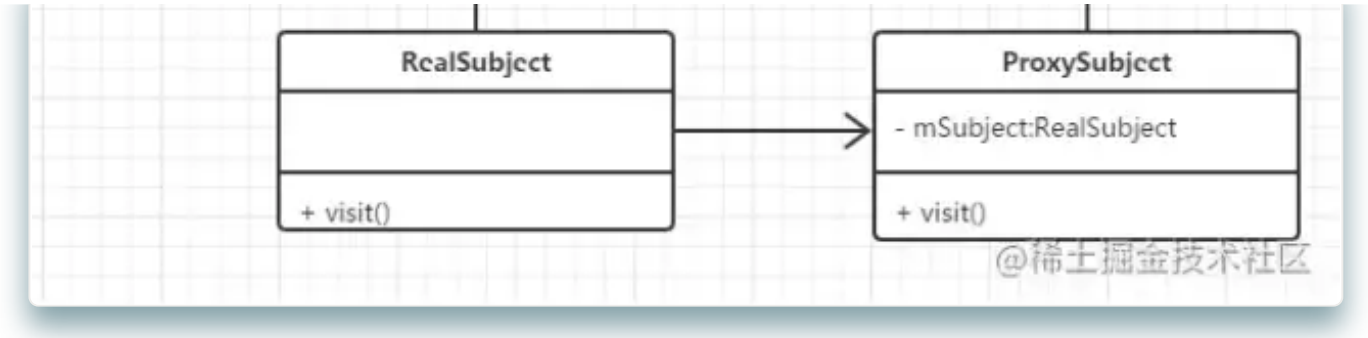
具体的AspectJ语法我们不深究，只需要知道pointcut是定义代理要代理的切入点，这里是定义了两个pointcut，分别是Star类的sing()方法和dance()方法。而before()和after()分别可以定义具体在切入点前后需要的额外操作。

总结一下，AspctJ就是用特定的编译器和语法，对类实现编译期增强，实现静态代理技术，下面我们看JDK静态代理。

3.2 JDK静态代理

通常情况下， JDK静态代理更多的是一种设计模式，JDK静态代理的代理类和委托类会实现同一接口或是派生自相同的父类，代理模式的基本类图入下：





我们接着通过把上面的明星和经纪人的例子写成代码来实现一个JDK静态代理模式。

经纪人类也实现了ShowService接口，持有了一个明星对象来提供真正的表演，并在各项表演的前后加入了经纪人需要处理的事情，如收钱、开发票等：

```
package com.fufu.aop;

/**
 * 经纪人
 */
public class Agent implements ShowService{

    private Star star;

    public Agent(Star star) {
        this.star = star;
    }

    private void getMoney() {
        System.out.println("get money");
    }

    private void writeReceipt() {
        System.out.println("write receipt");
    }
    @Override
    public void sing(String songName) {
        // 唱歌开始前收钱
        getMoney();
        // 明星开始唱歌
        star.sing(songName);
        // 唱歌结束后开发票
        writeReceipt();
    }

    @Override
```

```
    public void dance() {
        // 跳舞开始前收钱
        getMoney();
        // 明星开始跳舞
        star.dance();
        // 跳舞结束后开发票
        writeReceipt();
    }
}
```

通过经纪人来请明星表演：

```
public static void main(String[] args) {
        Agent agent = new Agent(new Star("Eminem"));
        agent.sing("Mockingbird");
}
```

输出：

```
get money
Eminem sing a song: Mockingbird
write receipt
```

以上就是一个典型的静态代理的实例，很简单但是也能说明问题，我们来看看静态代理的优缺点：

优点： 业务类可以只关注自身逻辑，可以重用，通过代理类来增加通用的逻辑处理。

缺点：

1. 代理对象的一个接口只服务于一种类型的对象，如果要代理的类很多，势必要为每一个类都进行代理，静态代理在程序规模稍大时就无法胜任了。
2. 如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度

另外，如果要按照上述的方法使用代理模式，那么真实角色(委托类)必须是事先已经存在的，并将其作为代理对象的内部属性。但是实际使用时，一个真实角色必须对应一个代理角色，如果大量使用会导致类的急剧膨胀；此外，如果事先并不知道真实角色（委托类），该如何使用代理呢？这些问题可以通过Java的动态代理类来解决。

4.动态代理

动态代理类的源码是在程序运行期间由JVM根据反射等机制动态的生成，所以不存在代理类的字节码文件。代理类和委托类的关系是在程序运行时确定。

4.1 动态代理思路

想弄明白动态代理类实现的思路是什么，我们还需用从静态代理的存在的问题入手，因为毕竟动态代理是为了解决静态代理存在问题而出现的，回过头来看静态代理的问题：

1. 类膨胀：每个代理类都是一个需要程序员编写的具体类，不现实。
2. 方法级代理：代理类和实现类都实现相同接口，导致代理类每个方法都需要进行代理，你有几个方法我就要有几个，编码复杂，无法维护。

动态代理如何解决：

1. 第一个问题很容易回答，类似使用Javasisst的例子，在代码中动态的创建代理类的字节码，然后获取到代理类对象。
2. 第二问题就要引出InvocationHandler了，为了构造出具有通用性和简单性的代理类，可以将所有的触发真实角色动作交给一个触发的管理器，让这个管理器统一地管理触发。这种管理器就是InvocationHandler。静态代理中，代理类无非是在前后加入特定逻辑后，调用对应的实现类的方法，sleep()对应sleep()，run()对应run()，而在Java中，方法Method也是一个对象，所以，动态代理类可以将对自己的所有调用作为Method对象都交给InvocationHandler处理，InvocationHandler根据是什么Method调用具体实现类的不同方法，InvocationHandler负责增加代理逻辑和调用具体的实现类的方法。

也就是说，动态代理类还是和实现类实现相同的接口，但是动态代理类是根据实现类实现的接口动态生成，不需要使用者关心，另外动态代理类的所有方法调用，统一交给InvocationHandler，不用处理实现类每个接口的每个方法。

在这种模式之中：代理Proxy和RealSubject应该实现相同的功能，这一点相当重要。（我这里说的功能，可以理解为某个类的public方法）

在面向对象的编程之中，如果我们想要约定Proxy和RealSubject可以实现相同的功能，有两种方式：

a.一个比较直观的方式，就是定义一个功能接口，然后让Proxy 和RealSubject来实现这个接口。
b.还有比较隐晦的方式，就是通过继承。因为如果Proxy继承自RealSubject，这样

Proxy则拥有了RealSubject的功能，Proxy还可以通过重写RealSubject中的方法，来实现多态。

其中JDK中提供的创建动态代理的机制，是以a这种思路设计的，而cglib则是以b思路设计的。

4.1 JDK动态代理（通过接口）

先来看一个具体的例子，还是以上边明星和经纪人的模型为例，这样方便对比理解：

将明星的表演抽象成一个ShowService接口，包括了唱歌、跳舞的功能：

```
package com.fufu.aop;

public interface ShowService {
    // 歌唱表演
    void sing(String songName);
    // 舞蹈表演
    void dance();
}
```

明星类实现了ShowService接口：

```
package com.fufu.aop;

/**
 * 明星类
 */
public class Star implements ShowService{
    private String name;

    @Override
    public void sing(String songName) {
        System.out.println(this.name + " sing a song: " + songName);
    }

    @Override
    public void dance() {
        System.out.println(this.name + "dance");
```

```
    }

    public Star(String name) {
        this.name = name;
    }

    public Star() {
    }
}
```

实现一个代理类的请求处理器，处理对具体类的所有方法的调用：

```
package com.fufu.aop;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class InvocationHandlerImpl implements InvocationHandler {

    ShowService target;

    public InvocationHandlerImpl(ShowService target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // 表演开始前收钱
        getMoney();
        // 明星开始唱歌
        Object invoke = method.invoke(target, args);
        // 表演结束后开发票
        writeReceipt();

        return invoke;
    }

    private void getMoney() {
        System.out.println("get money");
    }

    private void writeReceipt() {
        System.out.println("write receipt");
```

```
    }
}
```

通过JDK动态代理机制实现一个动态代理：

```
package com.fufu.aop;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;

public class JDKProxyDemo {

    public static void main(String[] args) {
        // 1.创建被代理的具体类
        Star star = new Star("Eminem");
        // 2.获取对应的ClassLoader
        ClassLoader classLoader = star.getClass().getClassLoader();
        // 3.获取被代理对象实现的所有接口
        Class[] interfaces = star.getClass().getInterfaces();
        // 4.设置请求处理器，处理所有方法调用
        InvocationHandler invocationHandler = new InvocationHandlerImpl(star);

        /**
         * 5.根据上面提供的信息，创建代理对象 在这个过程中，
         *    a.JDK会通过根据传入的参数信息动态地在内存中创建和.class文件等同的字节码
         *    b.然后根据相应的字节码转换成对应的class，
         *    c.然后调用newInstance()创建实例
         */
        Object o = Proxy.newProxyInstance(classLoader, interfaces, invocationHandler);
        ShowService showService = (ShowService)o;
        showService.sing("Mockingbird");
    }
}
```

我们从代理的创建入手，看看JDK的动态代理都做了什么：

```
Object o = Proxy.newProxyInstance(classLoader, interfaces, invocationHandler);
```

1. Proxy.newProxyInstance()获取Star类的所有接口列表（第二个参数：interfaces）
2. 确定要生成的代理类的类名，默认为：com.sun.proxy.$ProxyXXXX

3. 根据需要实现的接口信息，在代码中动态创建该Proxy类的字节码；
4. 将对应的字节码转换为对应的class对象；
5. 创建InvocationHandler实例handler，用来处理Proxy所有方法调用
6. Proxy的class对象以创建的handler对象为参数（第三个参数：invocationHandler），实例化一个Proxy对象

而对于InvocationHandler，我们需要实现下列的invoke方法：

```
public Object invoke(Object proxy, Method method, Object[] args)
```

在调用代理对象中的每一个方法时，在代码内部，都是直接调用了InvocationHandler的invoke方法，而invoke方法根据代理类传递给自己的method参数来区分是什么方法。

可以看出，Proxy.newProxyInstance()方法生成的对象也是实现了ShowService接口的，所以可以在代码中将其强制转换为ShowService来使用，和静态代理到达了同样的效果。我们可以用下面代码把生成的代理类的字节码保存到磁盘里，然后反编译看看JDK生成的动态代理类的结构。

```
package com.fufu.aop;

import sun.misc.ProxyGenerator;

import java.io.FileOutputStream;
import java.io.IOException;

public class ProxyUtils {

    public static void main(String[] args) {
        Star star = new Star("Eminem");
        generateClassFile(star.getClass(), "StarProxy");
    }

    public static void generateClassFile(Class clazz, String proxyName) {

        //根据类信息和提供的代理类名称，生成字节码
        byte[] classFile = ProxyGenerator.generateProxyClass(proxyName, clazz.getInterfaces())
        String paths = clazz.getResource(".").getPath();
        System.out.println(paths);
```

```
        FileOutputStream out = null;

        try {
            //保留到硬盘中
            out = new FileOutputStream(paths + proxyName + ".class");
            out.write(classFile);
            out.flush();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                out.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

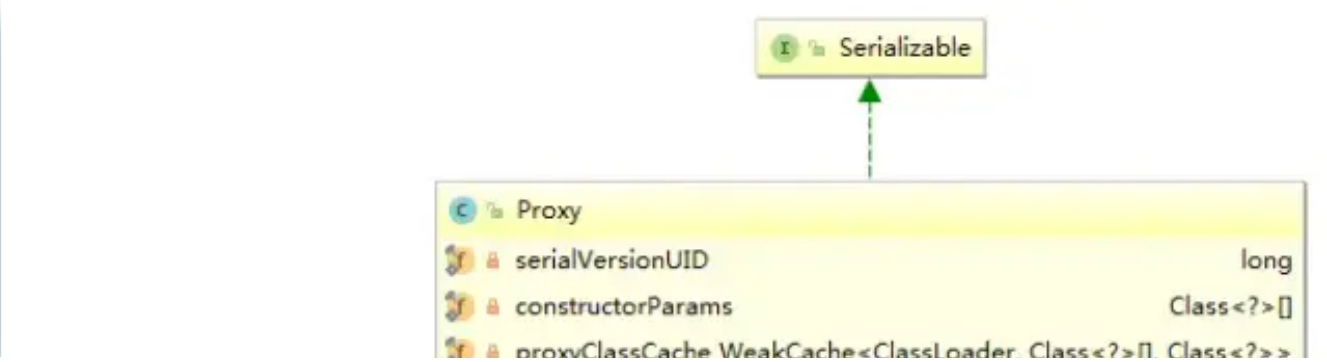
反编译StarPoxy.class文件后得到：

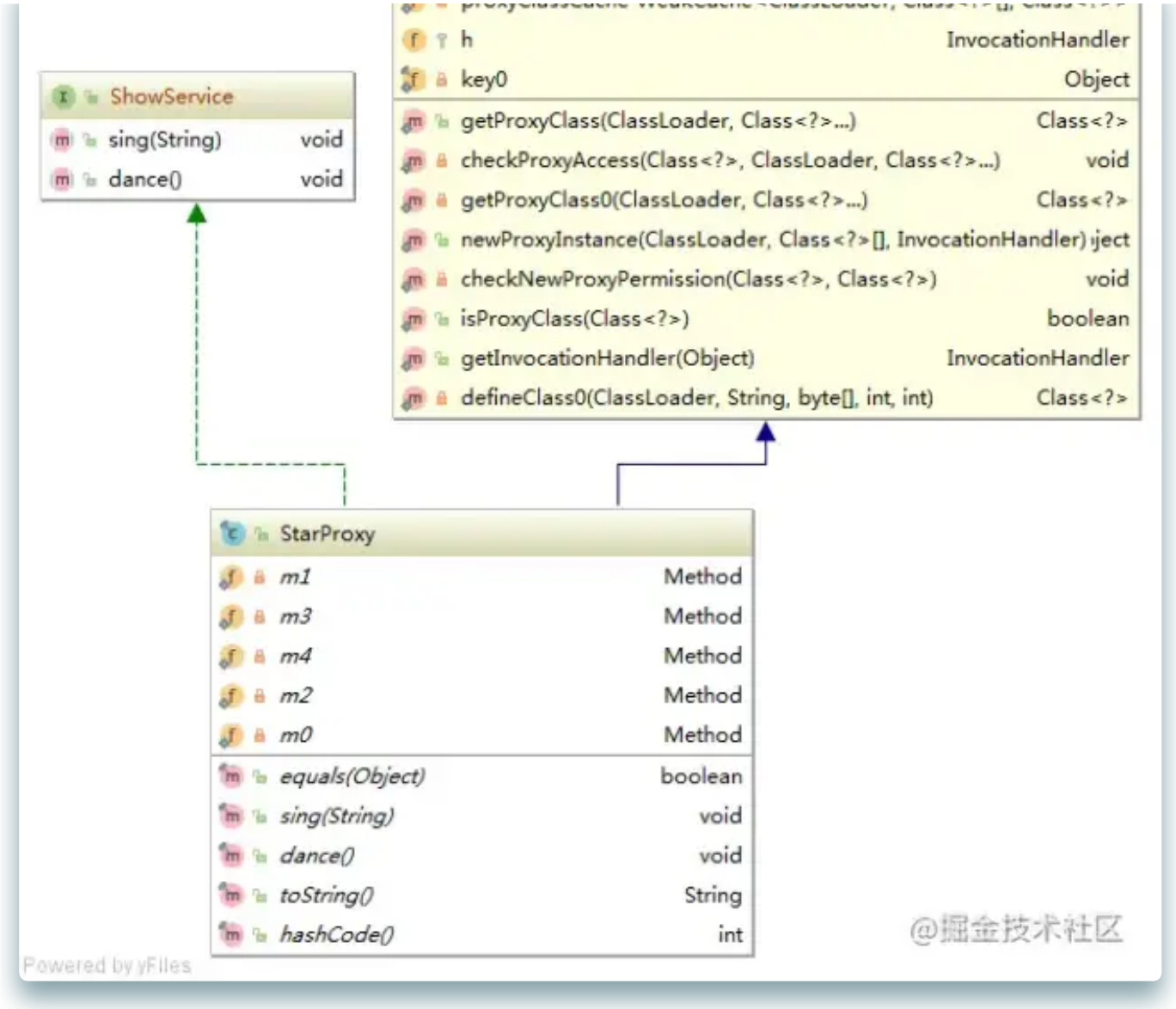
```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by Fernflower decompiler)
//

import com.fufu.aop.ShowService;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;

// 动态代理类StarPoxy实现了ShowService接口
public final class StarProxy extends Proxy implements ShowService {
    // 加载接口中定义的所有方法
    private static Method m1;
    private static Method m3;
    private static Method m4;
    private static Method m2;
    private static Method m0;
```

```
//构造函数接入InvocationHandler，也就是持有了InvocationHandler对象h
public StarProxy(InvocationHandler var1) throws  {
    super(var1);
}

public final boolean equals(Object var1) throws  {
    try {
        return ((Boolean)super.h.invoke(this, m1, new Object[]{var1})).booleanValue();
    } catch (RuntimeException | Error var3) {
        throw var3;
    } catch (Throwable var4) {
        throw new UndeclaredThrowableException(var4);
    }
}

// 自动生成的sing()方法，实际调用InvocationHandler对象h的invoke方法，传入m3参数对象代表sing()方法
public final void sing(String var1) throws  {
    try {
        super.h.invoke(this, m3, new Object[]{var1});
    } catch (RuntimeException | Error var3) {
        throw var3;
    } catch (Throwable var4) {
        throw new UndeclaredThrowableException(var4);
    }
}

//同理生成dance()方法
public final void dance() throws  {
    try {
        super.h.invoke(this, m4, (Object[])null);
    } catch (RuntimeException | Error var2) {
        throw var2;
    } catch (Throwable var3) {
        throw new UndeclaredThrowableException(var3);
    }
}

public final String toString() throws  {
    try {
        return (String)super.h.invoke(this, m2, (Object[])null);
    } catch (RuntimeException | Error var2) {
        throw var2;
    } catch (Throwable var3) {
```

```
            throw new UndeclaredThrowableException(var3);
        }
    }

    public final int hashCode() throws  {
        try {
            return ((Integer)super.h.invoke(this, m0, (Object[])null)).intValue();
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    // 加载接口中定义的所有方法
    static {
        try {
            m1 = Class.forName("java.lang.Object").getMethod("equals", new Class[]{Class.forNa
            m3 = Class.forName("com.fufu.aop.ShowService").getMethod("sing", new Class[]{Class
            m4 = Class.forName("com.fufu.aop.ShowService").getMethod("dance", new Class[0]);
            m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[0]);
            m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]);
        } catch (NoSuchMethodException var2) {
            throw new NoSuchMethodError(var2.getMessage());
        } catch (ClassNotFoundException var3) {
            throw new NoClassDefFoundError(var3.getMessage());
        }
    }
}
```

通过上面反编译后的代码可以看出，JDK生成的动态代理类实现和具体类相同的接口，并持有InvocationHandler对象（InvocationHandler对象又持有具体类），调用动态代理类中方法，会触发传入InvocationHandler的invoke()方法，通过method参数，来区分调用的是什么具体的方法，具体如下图所示：





4.2 CGLIB动态代理（通过继承）

JDK中提供的生成动态代理类的机制有个鲜明的特点是：

某个类必须有实现的接口，而生成的代理类也只能代理某个类接口定义的方法，比如：如果上面例子的Star实现了继承自ShowService接口的方法外，另外实现了方法play(),则在产生的动态代理类中不会有这个方法了！更极端的情况是：如果某个类没有实现接口，那么这个类就不能用JDK产生动态代理了！

幸好我们有cglib，“CGLIB（Code Generation Library），是一个强大的，高性能，高质量的Code生成类库，它可以在运行期扩展Java类与实现Java接口。”

cglib 创建某个类A的动态代理类的模式是：

1. 查找A上的所有非final 的public类型的方法定义；
2. 将这些方法的定义转换成字节码；
3. 将组成的字节码转换成相应的代理的class对象；

4. 实现 MethodInterceptor接口，用来处理对代理类上所有方法的请求（这个接口和JDK动态代理InvocationHandler的功能和角色是一样的）

有了上边JDK动态代理的例子，cglib的理解起来就简单了，还是先以实例说明，ShowService接口和Star类都复用之前的不变：

实现 MethodInterceptor接口：

```
package com.fufu.aop;

import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class MethodInterceptorImpl implements MethodInterceptor {

    @Override
    public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy
        // 表演开始前收钱
        getMoney();
        // 明星开始唱歌
        Object invoke = methodProxy.invokeSuper(o, objects);
        // 表演结束后开发票
        writeReceipt();

        return invoke;
    }

    private void getMoney() {
        System.out.println("get money");
    }

    private void writeReceipt() {
        System.out.println("write receipt");
    }
}
```

创建动态代理：

```
package com.fufu.aop;
```

```
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;

public class CglibProxyDemo {

    public static void main(String[] args) {
        Star star = new Star("Eminem");

        MethodInterceptor methodInterceptor = new MethodInterceptorImpl();

        //cglib 中加强器，用来创建动态代理
        Enhancer enhancer = new Enhancer();
        //设置要创建动态代理的类
        enhancer.setSuperclass(star.getClass());
        // 设置回调，这里相当于是对于代理类上所有方法的调用，都会调用CallBack，而Callback则需要实行inte
        enhancer.setCallback(methodInterceptor);

        ShowService showService = (ShowService) enhancer.create();
        showService.sing("Mockingbird");
    }
}
```

通过以上实例可以看出，Cglib通过继承实现动态代理，具体类不需要实现特定的接口，而且代理类可以调用具体类的非接口方法，更加灵活。

5.Spring AOP

5.1 概念

AOP的具体概念就不再说了，网上一搜一大把，这篇文章主要介绍Spring AOP低层使用的代理技术，因为平时在使用Spring AOP时，很多人都是copy配置，对上面介绍的这些技术概念并不清楚。

Spring AOP采用的是动态代理，在运行期间对业务方法进行增强，所以不会生成新类，对于动态代理技术，Spring AOP提供了对JDK动态代理的支持以及CGLib的支持，然而什么时候用哪种代理呢？

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，spring会自动在JDK动态代理和CGLIB之间转换

目前来看，Spring貌似和AspectJ没半毛钱关系，那为什么在许多应用了Spring AOP的项目中都出现了@AspectJ的注解呢？Spring是应用的动态代理，怎么会还和AspectJ有关系呢，原因是Spring AOP基于注解配置的情况下，需要依赖于AspectJ包的标准注解，但是不需要额外的编译以及AspectJ的织入器，而基于XML配置不需要，所以Spring AOP只是复用了AspectJ的注解，并没有其他依赖AspectJ的地方。

当Spring需要使用@AspectJ注解支持时，需要在Spring配置文件中如下配置：

```
<aop:aspectj-autoproxy/>
```

而关于第二点强制使用CGLIB，可以通过在Spring的配置文件如下配置实现：

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

proxy-target-class属性值决定是基于接口的还是基于类的代理被创建。如果proxy-target-class 属性值被设置为true，那么基于类的代理将起作用（这时需要cglib库）。如果proxy-target-class属值被设置为false或者这个属性被省略，那么标准的JDK 基于接口的代理。

所以，虽然使用了Aspect的Annotation，但是并没有使用它的编译器和织入器。其实现原理是JDK动态代理或Cglib，在运行时生成代理类。

已经写了这么多了，下面再贴两个Spring AOP的demo代码吧，分别是基于XML和注解的：

5.2 基于XML

切面类：

```
package com.fufu.spring.aop;

import org.springframework.stereotype.Component;
```

```java
/**
 * 基于XML的Spring AOP
 */
@Component
public class AgentAdvisorXML {

    public void getMoney() {
        System.out.println("get money");
    }

    public void writeReceipt() {
        System.out.println("write receipt");
    }
}
```

配置文件：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean id="star" class="com.fufu.proxy.Star">
        <property name="name" value="Eminem"/>
    </bean>

    <bean id="agentAdvisorXML" class="com.fufu.spring.aop.AgentAdvisorXML"/>

     <!--Spring基于Xml的切面-->
     <aop:config>
         <!-- 定义切点函数 -->
         <aop:pointcut id="singPointCut" expression="execution(* com.fufu.proxy.Star.sing(..))
```

```
        <!-- 定义切面 order 定义优先级,值越小优先级越大-->
        <aop:aspect ref="agentAdvisorXML" order="0">
            <!--前置通知-->
            <aop:before method="getMoney" pointcut-ref="singPointCut"/>
            <!--后置通知-->
            <aop:after method="writeReceipt" pointcut-ref="singPointCut"/>
        </aop:aspect>
    </aop:config>

</beans>
```

测试类：

```
package com.fufu.spring.aop;

import com.fufu.proxy.ShowService;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext

        Object star = applicationContext.getBean("star");

        ShowService showService = (ShowService)star;
        showService.sing("Mockingbird");
    }
}
```

5.3 基于注解

切面类：

```
package com.fufu.spring.aop;

import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
```

```java
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

/**
 * 基于注解的Spring AOP
 */
@Aspect
@Component
public class AgentAdvisor {

    @Before(value = "execution(* com.fufu.proxy.ShowService.sing(..))")
    public void getMoney() {
        System.out.println("get money");
    }

    @After(value = "execution(* com.fufu.proxy.ShowService.sing(..))")
    public void writeReceipt() {
        System.out.println("write receipt");
    }
}
```

配置文件：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop.xsd">

    <context:component-scan base-package="com.fufu.proxy, com.fufu.spring.aop"/>

    <aop:aspectj-autoproxy  proxy-target-class="true"/>

</beans>
```

测试类：

```
package com.fufu.spring.aop;

import com.fufu.proxy.ShowService;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext

        Object star = applicationContext.getBean("star");

        ShowService showService = (ShowService)star;
        showService.sing("Mockingbird");
    }
}
```

6.总结

以上内容，虽然比较浅显易懂，但是可以对Java代理机制和Spring AOP会有一个全面的理解，如有错误，欢迎指正。

推荐文章

1、一款高颜值的 SpringBoot+JPA 博客项目
2、超优 Vue+Element+Spring 中后端解决方案
3、推荐几个支付项目！
4、推荐一个 Java 企业信息化系统
5、一款基于 Spring Boot 的现代化社区（论坛/问答/社交网络/博客）