Java 基础提升篇：Java 技术之动态代理机制

静态代理

常规的代理模式有以下三个部分组成：

功能接口

```
interface IFunction {
    void doAThing();
}
```

功能提供者

```
class FunctionProvider implement IFunction {
    public void doAThing {
        System.out.print("do A");
    }
}
```

功能代理者

```
class Proxy implement IFunction {
    private FunctionProvider provider;
    Proxy(FunctionProvider provider) {
        this.provider = provider;
    }
    public void doAThing {
        provider.doAThing();
    }
}
```

前两者就是普通的接口和实现类，而第三个就是所谓的代理类。对于使用者而言，他会让代理类去完成某件任务，并不关心这件任务具体的跑腿者。

这就是静态代理，好处是方便调整变换具体实现类，而使用者不会受到任何影响。

不过这种方式也存在弊端：比如有多个接口需要进行代理，那么就要为每一个功能提供

者创建对应的一个代理类，那就会越来越庞大。而且，所谓的“静态”代理，意味着必须提前知道被代理的委托类。
通过下面一个例子来说明下：

统计函数耗时–静态代理实现

现在希望通过一个代理类，对我感兴趣的方法进行耗时统计，利用静态代理有如下实现：

```
interface IAFunc {
    void doA();
}
interface IBFunc {
    void doB();
}
class TimeConsumeProxy implement IAFunc, IBFunc {
    private AFunc a;
    private BFunc b;
    public(AFunc a, BFunc b) {
        this.a = a;
        this.b = b;
    }
    void doA() {
        long start = System.currentMillions();
        a.doA();
        System.out.println("耗时：" + (System.currentMillions() - start));
    }
    void doB() {
        long start = System.currentMillions();
        b.doB();
        System.out.println("耗时：" + (System.currentMillions() - start));
    }
}
```

弊端很明显，如果接口越多，每新增一个函数都要去修改这个 TimeConsumeProxy 代理类：把委托类对象传进去，实现接口，在函数执行前后统计耗时。

这种方式显然不是可持续性的，下面来看下使用动态代理的实现方式，进行对比。

动态代理

动态代理的核心思想是通过 Java Proxy 类，为传入进来的任意对象动态生成一个代理

对象，这个代理对象默认实现了原始对象的所有接口。

还是通过统计函数耗时例子来说明更加直接。

统计函数耗时–动态代理实现

```
interface IAFunc {
    void doA();
}
interface IBFunc {
    void doB();
}
class A implement IAFunc { ... }
class B implement IBFunc { ... }
class TimeConsumeProxy implements InvocationHandler {
    private Object realObject;
    public Object bind(Object realObject) {
        this.realObject = realObject;
        Object proxyObject = Proxy.newInstance(
            realObject.getClass().getClassLoader(),
            realObject.getClass().getInterfaces(),
            this
        );
        return proxyObject;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        long start = System.currentMillions();
        Object result = method.invoke(target, args);
        System.out.println("耗时：" + (System.currentMillions() - start));
        return result;
    }
}
```

具体使用时：

```
public static void main(String[] args) {
    A a = new A();
    IAFunc aProxy = (IAFunc) new TimeConsumeProxy().bind(a);
    aProxy.doA();
    B b = new B();
    IBFunc bProxy = (IBFunc) new TimeConsumeProxy().bind(b);
```

```
    bProxy.doB();
}
```

这里最大的区别就是：代理类和委托类互相透明独立，逻辑没有任何耦合，在运行时才绑定在一起。这也就是静态代理与动态代理最大的不同，带来的好处就是：无论委托类有多少个，代理类不受到任何影响，而且在编译时无需知道具体委托类。
回到动态代理本身，上面代码中最重要的就是:

```
Object proxyObject = Proxy.newInstance(
            realObject.getClass().getClassLoader(),
            realObject.getClass().getInterfaces(),
            this
        );
```

通过 Proxy 工具，把真实委托类转换成了一个代理类，最开始提到了一个代理模式的三要素：功能接口、功能提供者、功能代理者；在这里对应的就是：

```
realObject.getClass().getInterfaces()，realObject，TimeConsumeProxy。
```

其实动态代理并不复杂，通过一个 Proxy 工具，为委托类的接口自动生成一个代理对象，后续的函数调用都通过这个代理对象进行发起，最终会执行到 InvocationHandler#invoke 方法，在这个方法里除了调用真实委托类对应的方法，还可以做一些其他自定义的逻辑，比如上面的运行耗时统计等。

探索动态代理实现机制

抛出几个问题：

上面生成的代理对象 Object proxyObject 究竟是个什么东西？为什么它可以转型成 IAFunc，还能调用 doA() 方法？

这个 proxyObject 是怎么生成出来的？它是一个 class 吗？
下面我先给出答案，再一步步探究这个答案是如何来的。
问题一: proxyObject 究竟是个什么 -> 动态生成的 $Proxy0.class 文件

在调用 Proxy.newInstance 后，Java 最终会为委托类 A 生成一个真实的 class 文件：$Proxy0.class，而 proxyObject 就是这个 class 的一个实例。
猜一下，这个 $Proxy0.class 类长什么样呢，包含了什么方法呢？回看下刚刚的代码：

```
IAFunc aProxy = (IAFunc) new TimeConsumeProxy().bind(a);
aProxy.doA();
```

推理下，显然这个 $Proxy0.class 实现了 IAFunc 接口，同时它内部也实现了 doA() 方法，而且重点是：这个 doA() 方法在运行时会执行到 TimeConsumeProxy#invoke() 方法里。

重点来了！下面我们来看下这个 $Proxy0.class 文件，把它放进 IDE 反编译下，可以看到如下内容，来验证下刚刚的猜想：

```
final class $Proxy0 extends Proxy implements IAFunc {
```

```
    private static Method m1;
    private static Method m3;
    private static Method m2;
    private static Method m0;
    public $Proxy0(InvocationHandler var1) throws  {
        super(var1);
    }
    public final boolean equals(Object var1) throws  {
        // 省略
    }
    public final void doA() throws  {
        try {
            // 划重点
            super.h.invoke(this, m3, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }
    public final String toString() throws  {
        // 省略
    }
    public final int hashCode() throws  {
        // 省略
    }
    static {
        try {
            // 划重点
            m3 = Class.forName("proxy.IAFunc").getMethod("doA", new Class[0]);
            m1 = Class.forName("java.lang.Object").getMethod("equals", new
Class[]{Class.forName("java.lang.Object")});
            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());
        }
```

```
    }
}
```

没错，刚刚的猜想都中了！实现了 IAFunc 接口和 doA() 方法，不过，doA()里是什么鬼？

```
super.h.invoke(this, m3, (Object[])null);
```

回看下，TimeConsumeProxy 里面的 invoke 方法，它的函数签名是啥？

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

没错，doA()里做的就是调用 TimeConsumeProxy#invoke() 方法。那么也就是说，下面这段代码执行流程如下：

```
IAFunc aProxy = (IAFunc) new TimeConsumeProxy().bind(a);
aProxy.doA();
```

基于传入的委托类 A，生成一个$Proxy0.class 文件；
创建一个 $Proxy0.class 对象，转型为 IAFunc 接口；
调用 aProxy.doA() 时，自动调用 TimeConsumeProxy 内部的 invoke 方法。

问题二：proxyObject 是怎么一步步生成出来的 -> $Proxy0.class 文件生成流程

刚刚从末尾看了结果，现在我们回到代码的起始端来看：

```
Object proxyObject = Proxy.newInstance(
            realObject.getClass().getClassLoader(),
            realObject.getClass().getInterfaces(),
            this
        );
```

准备好，开始发车读源码了。我会截取重要的代码并加上注释。

先看 Proxy.newInstance():

```
public static Object newProxyInstance(ClassLoader loader,
                                      Class<?>[] interfaces,
                                      InvocationHandler h) {
    // 复制要代理的接口
    final Class<?>[] intfs = interfaces.clone();
    // 重点：生成 $Proxy0.class 文件并通过 ClassLoader 加载进来
    Class<?> cl = getProxyClass0(loader, intfs);
    // 对 $Proxy0.class 生成一个实例，就是 `proxyObject`
    final Constructor<?> cons = cl.getConstructor(constructorParams);
    return cons.newInstance(new Object[]{h});
}
```

再来看 getProxyClass0 的具体实现：ProxyClassFactory 工厂类：

```
@Override
public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {
    // 参数为 ClassLoader 和要代理的接口
```

```
    Map<Class<?>, Boolean> interfaceSet = new
IdentityHashMap<>(interfaces.length);
    // 1. 验证 ClassLoader 和接口有效性
    for (Class<?> intf : interfaces) {
       // 验证 classLoader 正确性
        Class<?> interfaceClass = Class.forName(intf.getName(), false, loader);
        if (interfaceClass != intf) {
            throw new IllegalArgumentException(
                intf + " is not visible from class loader");
        }
        // 验证传入的接口 class 有效
        if (!interfaceClass.isInterface()) { ... }
        // 验证接口是否重复
        if (interfaceSet.put(interfaceClass, Boolean.TRUE) != null) { ... }
    }
    // 2. 创建包名及类名 $Proxy0.class
    proxyPkg = ReflectUtil.PROXY_PACKAGE + ".";
    long num = nextUniqueNumber.getAndIncrement();
    String proxyName = proxyPkg + proxyClassNamePrefix + num;
    // 3. 创建 class 字节码内容
    byte[] proxyClassFile = ProxyGenerator.generateProxyClass(proxyName,
interfaces, accessFlags);
    // 4. 基于字节码和类名，生成 Class<?> 对象
    return defineClass0(loader, proxyName, proxyClassFile, 0,
proxyClassFile.length);
}
```

再看下第三步生成 class 内容 ProxyGenerator.generateProxyClass：

```
// 添加 hashCode equals toString 方法
addProxyMethod(hashCodeMethod, Object.class);
addProxyMethod(equalsMethod, Object.class);
addProxyMethod(toStringMethod, Object.class);
// 添加委托类的接口实现
for (int i = 0; i < interfaces.length; i++) {
    Method[] methods = interfaces[i].getMethods();
    for (int j = 0; j < methods.length; j++) {
        addProxyMethod(methods[j], interfaces[i]);
    }
}
// 添加构造函数
methods.add(this.generateConstructor());
```

这里构造好了类的内容：添加必要的函数，实现接口，构造函数等，下面就是要写入上一步看到的 $Proxy0.class 了。

```
ByteArrayOutputStream bout = new ByteArrayOutputStream();
DataOutputStream dout = new DataOutputStream(bout);
dout.writeInt(0xCAFEBABE);
...
dout.writeShort(ACC_PUBLIC | ACC_FINAL | ACC_SUPER);
...
return bout.toByteArray();
```

到这里就生成了第一步看到的 $Proxy0.class 文件了，完成闭环，讲解完成！

动态代理小结

通过上面的讲解可以看出，动态代理可以随时为任意的委托类进行代理，并可以在 InvocationHandler#invoke 拿到运行时的信息，并可以做一些切面处理。

在动态代理背后，其实是为一个委托类动态生成了一个 $Proxy0.class 的代理类，该代理类会实现委托类的接口，并把接口调用转发到 InvocationHandler#invoke 上，最终调用到真实委托类的对应方法。

动态代理机制把委托类和代理类进行了隔离，提高了扩展性。

Java 动态代理与 Python 装饰器

这是 Java 语言提供的一个有意思的语言特性，而其实 Python 里也提供了一种类似的特性：装饰器，可以达到类似的面相切面编程思想，下次有空再把两者做下对比，这次先到这。