# 课程目标

1、掌握代理模式的应用场景和实现原理。
2、了解静态代理和动态代理的区别。
3、了解 CGLib 和 JDK Proxy 的根本区别。
4、手写实现定义的动态代理。

# 内容定位

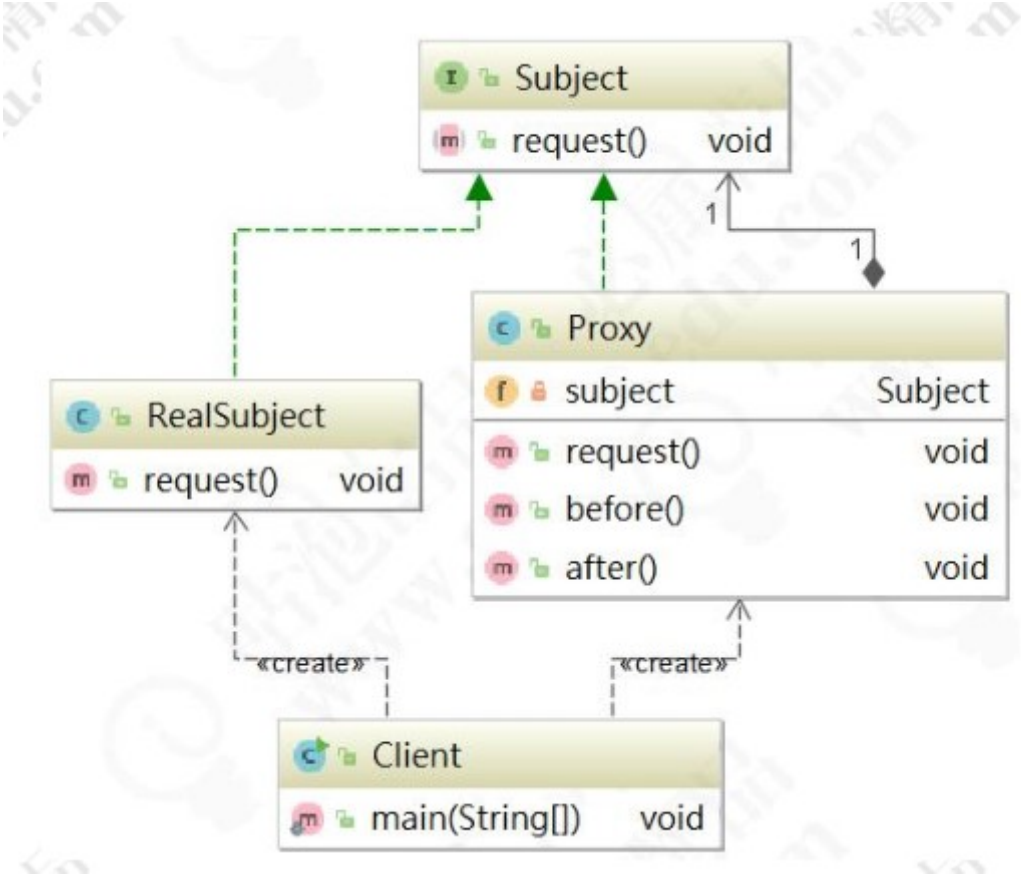
都知道 SpringAOP 是用代理模式实现，到底是怎么实现的？我们来一探究竟，并且自己仿真手写还原部分细节。

# 1. 代理模式

代理模式（Proxy Pattern）是指为其他对象提供一种代理，以控制对这个对象的访问，属于结构型模式。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

> 官方原文： Provide a surrogate or placeholder for another object to control access to it.

首先来看代理模式的通用 UML 类图：




代理模式一般包含三种角色：

> 抽象主题角色（Subject）：抽象主题类的主要职责是声明真实主题与代理的共同接口方法，该类可以是接口也可以是抽象类；
>
> 真实主题角色（RealSubject）：该类也被称为被代理类，该类定义了代理所表示的真实对象，是负责执行系统真正的逻辑业务对象；
>
> 代理主题角色（Proxy）：也被称为代理类，其内部持有 RealSubject 的引用，因此具备完全的对RealSubject 的代理权。客户端调用代理对象的方法，同时也调用被代理对象的方法，但是会在代理对象前后增加一些处理代码。

在代码中，一般代理会被理解为代码增强，实际上就是在原代码逻辑前后增加一些代码逻辑，而使调用者无感知。代理模式属于结构型模式，分为静态代理和动态代理。

## 1.1. 代理模式的应用场景

生活中的租房中介、售票黄牛、婚介、经纪人、快递、事务代理、非侵入式日志监听等，都是代理模式的实际体现。当无法或不想直接引用某个对象或访问某个对象存在困难时，可以通过也给代理对象来间接访问。使用代理模式主要有两个目的：一是保护目标对象，二是增强目标对象。

## 1.2. 代理模式的通用写法

下面是代理模式的通用代码展示，首先创建代理主题角色 ISubject 类：

```
package cn.sitedev.common;

public interface ISubject {
    void request();
}
```

创建真实主题角色 RealSubject 类：

```
package cn.sitedev.common;

public class RealSubject implements ISubject {
    @Override
    public void request() {
        System.out.println("RealSubject.request");
    }
}
```

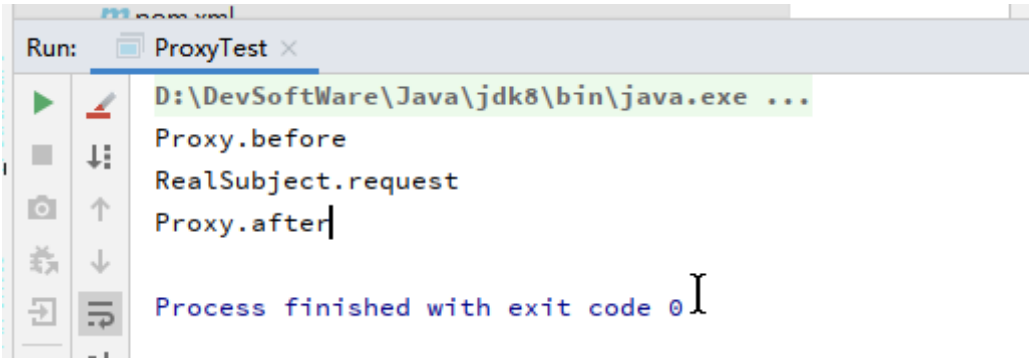
创建代理主题角色 Proxy 类：

```
package cn.sitedev.common;

public class Proxy implements ISubject {
    private ISubject subject;

    public Proxy(ISubject subject) {
        this.subject = subject;
    }

    @Override
    public void request() {
        before();
        this.subject.request();
        after();
    }
```

```
16
17      private void before() {
18          System.out.println("Proxy.before");
19      }
20
21      private void after() {
22          System.out.println("Proxy.after");
23      }
24  }
```

客户端调用代码：

```
1  package cn.sitedev.common;
2
3  public class ProxyTest {
4      public static void main(String[] args) {
5          Proxy proxy = new Proxy(new RealSubject());
6          proxy.request();
7      }
8  }
```




## 1.3. 从静态代理到动态代理

举个例子，有些人到了适婚年龄，其父母总是迫不及待地希望早点抱孙子。而现在在各种压力之下，很多人都选择晚婚晚育。于是着急的父母就开始到处为自己的子女相亲，比子女自己还着急。下面来看代码实现。创建顶层接口 IPerson 的代码如下：

```
1  package cn.sitedev.staticproxy;
2
3  public interface IPerson {
```

```
    void findLove();
}
```

儿子张三要找对象，实现 ZhangSan 类：

```
package cn.sitedev.staticproxy;

public class Zhangsan implements IPerson {
    @Override
    public void findLove() {
        System.out.println("儿子要求：肤白貌美大长腿");
    }
}
```

父亲张老三要帮儿子张三相亲，实现 Father 类：

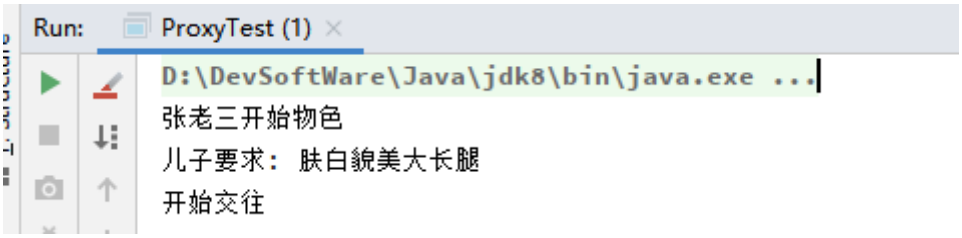
```
package cn.sitedev.staticproxy;

public class ZhangLaosan implements IPerson {
    private Zhangsan zhangsan;

    public ZhangLaosan(Zhangsan zhangsan) {
        this.zhangsan = zhangsan;
    }

    @Override
    public void findLove() {
        System.out.println("张老三开始物色");
        this.zhangsan.findLove();
        System.out.println("开始交往");
    }
}
```

来看测试代码：

```
package cn.sitedev.staticproxy;

```

```
public class ProxyTest {
    public static void main(String[] args) {
        ZhangLaosan zhangLaosan = new ZhangLaosan(new Zhangsan());
        zhangLaosan.findLove();
    }
}
```

运行结果如下图所示。




但上面的场景有个弊端，就是自己父亲只会给自己的子女去物色对象，别人家的孩子是不会管的。

但社会上这项业务发展成了一个产业，出现了媒婆、婚介所等，还有各种各样的定制套餐。如果还使用静态代理成本就太高了，需要一个更加通用的解决方案，满足任何单身人士找对象的需求。这就是由静态代理升级到了动态代理。采用动态代理基本上只要是人（IPerson）就可以提供相亲服务。动态代理的底层实现一般不用我们自己亲自去实现，已经有很多现成的 API。在 Java 生态中，目前最普遍使用的是 JDK 自带的代理和 Cglib 提供的类库。下面我们首先基于 JDK 的动态代理支持如来升级一下代码。

首先，创建媒婆（婚介所）类 JdkMeipo：

```
package cn.sitedev.dynamicproxy;

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

public class JdkMeipo implements InvocationHandler {
    private IPerson target;

    public IPerson getInstance(IPerson target) {
        this.target = target;
        Class<?> clazz = target.getClass();
        return (IPerson) Proxy.newProxyInstance(clazz.getClassLoader(), clazz.ge
    }

```

```
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Thro
        before();
        Object result = method.invoke(this.target, args);
        after();
        return result;
    }

    private void before() {
        System.out.println("我是媒婆，已经收集到你的需求，开始物色");
    }

    private void after() {
        System.out.println("双方同意，开始交往");
    }
}
```

再创建一个类 ZhaoLiu：

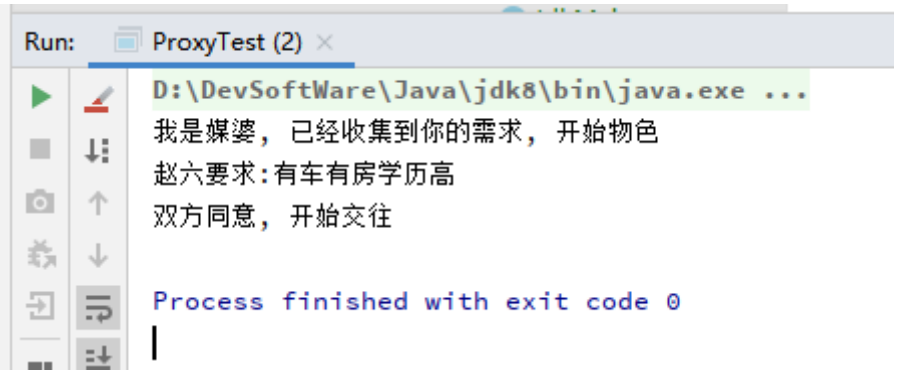
```
package cn.sitedev.dynamicproxy;

public class ZhaoLiu implements IPerson {
    @Override
    public void findLove() {
        System.out.println("赵六要求:有车有房学历高");
    }

    public void buyInsure() {

    }
}
```

测试代码如下：

```
package cn.sitedev.dynamicproxy;

public class ProxyTest {
    public static void main(String[] args) {
        JdkMeipo jdkMeipo = new JdkMeipo();
```

```
        IPerson zhaoliu = jdkMeipo.getInstance(new ZhaoLiu());
        zhaoliu.findLove();
    }
}
```

运行效果如下图所示。




## 1.4. 静态代理在业务中的应用

这里“小伙伴们”可能会觉得还是不知道如何将代理模式应用到业务场景中，我们来看一个实际的业务场景。在分布式业务场景中，通常会对数据库进行分库分表，分库分表之后使用 Java 操作时就可能需要配置多个数据源，我们通过设置数据源路由来动态切换数据源。先创建 Order 订单类：

```
package cn.sitedev.order;

import lombok.Data;

@Data
public class Order {
    private Object orderInfo;
    private Long createTime;
    private String id;
}
```

创建 OrderDao 持久层操作类：

```
package cn.sitedev.order;

public class OrderDao {
```

```
    public int insert(Order order) {
        System.out.println("OrderDao 创建 Order 成功");
        return 1;
    }
}
```

创建 IOrderService 接口：

```
package cn.sitedev.order;

public interface IOrderService {
    int createOrder(Order order);
}
```

创建 OrderService 实现类：

```
package cn.sitedev.order;

public class OrderService implements IOrderService {
    private OrderDao orderDao;

    public OrderService() {
        // 这里如果使用Spring , 应该是自动注入的
        // 为了使用方便, 我们在构造方法中将orderDao直接初始化
        this.orderDao = new OrderDao();
    }

    @Override
    public int createOrder(Order order) {
        System.out.println("OrderService调用OrderDao创建订单");
        return orderDao.insert(order);
    }
}
```

接下来使用静态代理，主要完成的功能是：根据订单创建时间自动按年进行分库。根据开闭原则，我们修改原来写好的代码逻辑，通过代理对象来完成。先创建数据源路由对象，使用 ThreadLocal 的单例实现 DynamicDataSourceEntry 类：

```java
package cn.sitedev.order;

/**
 * 动态切换数据源
 */
public class DynamicDataSourceEntry {
    /**
     * 默认数据源
     */
    public static final String DEFAULT_SOURCE = null;

    private static final ThreadLocal<String> local = new ThreadLocal<>();

    private DynamicDataSourceEntry() {
    }

    /**
     * 清空数据源
     */
    public static void clear() {
        local.remove();
    }

    /**
     * 获取当前正在使用的数据源名称
     */
    public static String get() {
        return local.get();
    }

    /**
     * 还原当前切换的数据源
     */
    public static void restore() {
        local.set(DEFAULT_SOURCE);
    }

    /**
     * '
     * 设置已知名字的数据源
     *
     * @param source
     */
```

```
    public static void set(String source) {
        local.set(source);
    }

    /**
     * 根据年份动态设置数据源
     *
     * @param year
     */
    public static void set(int year) {
        local.set("DB_" + year);
    }
}
```

创建切换数据源的代理类 OrderServiceSaticProxy：

```
package cn.sitedev.order;

import java.text.SimpleDateFormat;
import java.util.Date;

public class OrderServiceStaticProxy implements IOrderService {
    private SimpleDateFormat yearFormat = new SimpleDateFormat("yyyy");
    private IOrderService orderService;

    public OrderServiceStaticProxy(IOrderService orderService) {
        this.orderService = orderService;
    }

    @Override
    public int createOrder(Order order) {
        before();
        Long time = order.getCreateTime();
        Integer dbRouter = Integer.valueOf(yearFormat.format(new Date(time)));
        System.out.println("静态代理类自动分配到[DB_" + dbRouter + "]数据源处理数据
        DynamicDataSourceEntry.set(dbRouter);
        orderService.createOrder(order);
        after();
        return 1;
    }

```

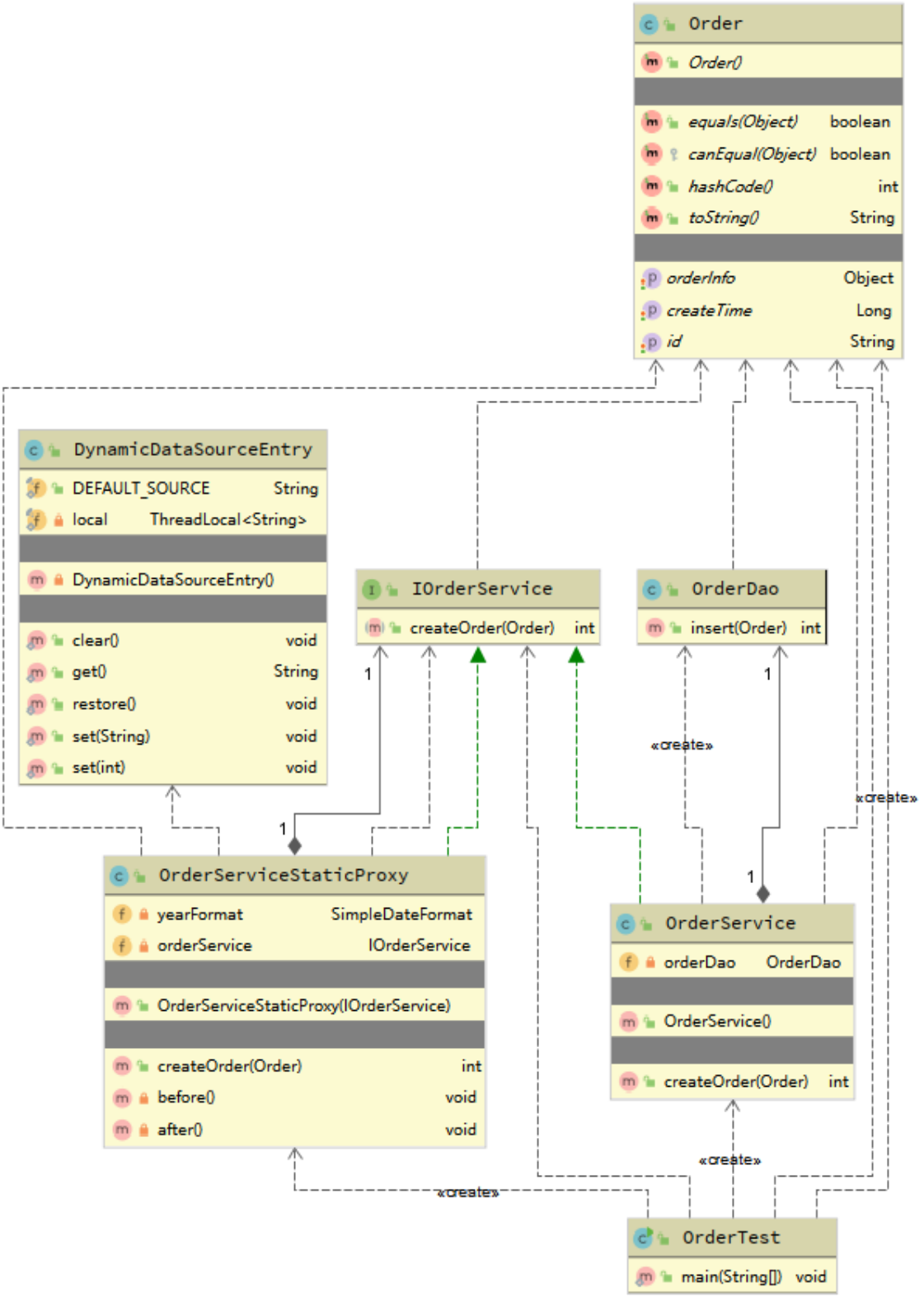
```
    private void before() {
        System.out.println("OrderServiceStaticProxy.before");
    }

    private void after() {
        System.out.println("OrderServiceStaticProxy.after");
    }
}
```

来看测试代码：

```
package cn.sitedev.order;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class StaticProxyTest {
    public static void main(String[] args) {
        try {
            Order order = new Order();
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");
            Date date = sdf.parse("2020/03/03");
            order.setCreateTime(date.getTime());

            IOrderService orderService = new OrderServiceStaticProxy(new OrderSe
            orderService.createOrder(order);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

运行结果如下图所示。


结果符合我们的预期。现在再来回顾一下类图，看是不是和我们最先画的一致，如下图所示。




动态代理和静态代理的基本思路是一致的，只不过动态代理功能更加强大，随着业务的扩展适应性更强。

## 1.5. 动态代理在业务中的应用

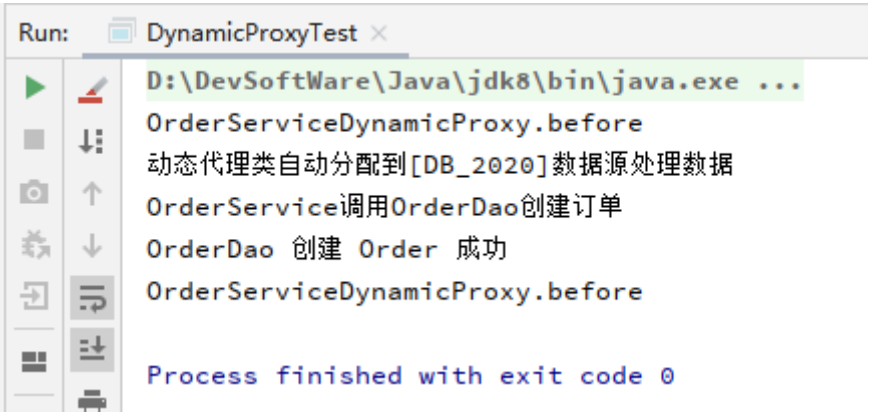
上面的案例理解了，我们再来看数据源动态路由业务，帮助“小伙伴们”加深对动态代理的印象。创建动态代理的类 OrderServiceDynamicProxy：

```java
package cn.sitedev.order;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.text.SimpleDateFormat;
import java.util.Date;

public class OrderServiceDynamicProxy implements InvocationHandler {

    private SimpleDateFormat yearFormat = new SimpleDateFormat("yyy");

    private Object target;

    public Object getInstance(Object target) {
        this.target = target;
        Class<?> clazz = target.getClass();
        return Proxy.newProxyInstance(clazz.getClassLoader(), clazz.getInterface
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Thro
        before(args[0]);
        Object result = method.invoke(target, args);
        after();
        return result;
    }

    private void before(Object target) {
        try {
            System.out.println("OrderServiceDynamicProxy.before");
            Long time = (Long) target.getClass().getMethod("getCreateTime").invo
            Integer dbRouter = Integer.valueOf(yearFormat.format(new Date(time))
            System.out.println("动态代理类自动分配到[DB_" + dbRouter + "]数据源处理
            DynamicDataSourceEntry.set(dbRouter);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
```

```
        }
    }

    private void after() {
        System.out.println("OrderServiceDynamicProxy.before");
    }
}
```

测试代码如下：

```
package cn.sitedev.order;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class DynamicProxyTest {
    public static void main(String[] args) {
        try {
            Order order = new Order();

            SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd");
            Date date = sdf.parse("2020/03/03");
            order.setCreateTime(date.getTime());

            IOrderService orderService = (IOrderService) new OrderServiceDynamic
            orderService.createOrder(order);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```




依然能够达到相同运行效果。但是，使用动态代理实现之后，我们不仅能实现 Order 的数据源动态路由，还可以实现其他任何类的数据源路由。当然，有个比较重要的约定，必须实现getCreateTime()方法，因为路由规则是根据时间来运算的。我们可以通过接口规范来达到约束的目的，在此就不再举例。

## 1.6. 手写JDK动态代理实现原理

不仅知其然，还得知其所以然。既然 JDK 动态代理功能如此强大，那么它是如何实现的呢？我们现在来探究一下原理，并模仿 JDK 动态代理动手写一个属于自己的动态代理。

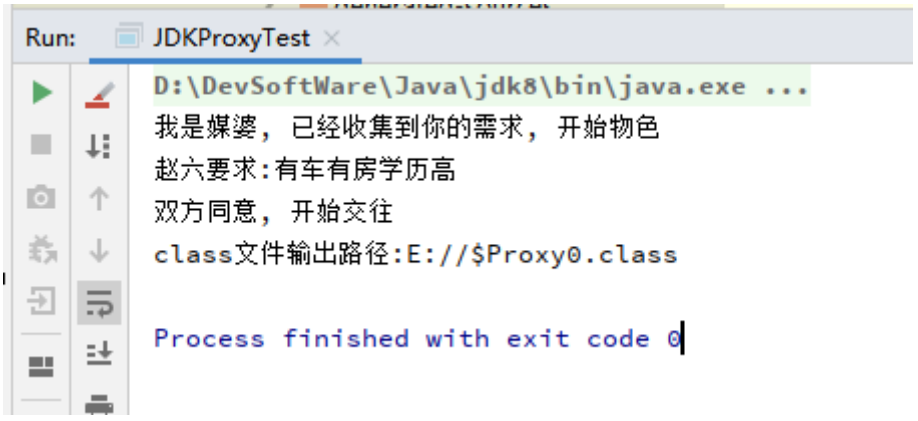
我们都知道 JDK 动态代理采用字节重组，重新生成对象来替代原始对象，以达到动态代理的目的。

JDK 动态代理生成对象的步骤如下：
（1）获取被代理对象的引用，并且获取它的所有接口，反射获取。
（2）JDK 动态代理类重新生成一个新的类，同时新的类要实现被代理类实现的所有接口。
（3）动态生成 Java 代码，新加的业务逻辑方法由一定的逻辑代码调用（在代码中体现）。
（4）编译新生成的 Java 代码.class 文件。
（5）重新加载到 JVM 中运行。

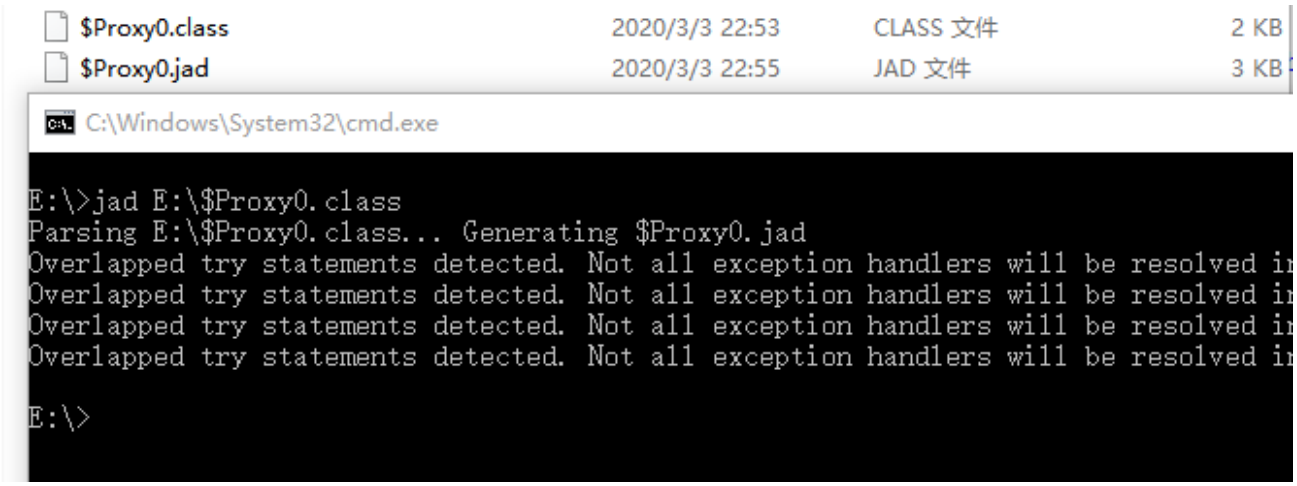
以上过程就叫字节码重组。JDK 中有一个规范，在 ClassPath 下只要是$开头的.class 文件，一般都是自动生成的。那么我们有没有办法看到代替后的对象的“真容”呢？做一个这样测试，我们将内存中的对象字节码通过文件流输出到一个新的.class 文件，然后利用反编译工具查看其源代码。

```
1 package cn.sitedev.jdkproxy;
2
3 import cn.sitedev.dynamicproxy.IPerson;
4 import cn.sitedev.dynamicproxy.JdkMeipo;
5 import cn.sitedev.dynamicproxy.ZhaoLiu;
6 import sun.misc.ProxyGenerator;
7
```

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class JDKProxyTest {
    public static void main(String[] args) {
        try {
            IPerson obj = new JdkMeipo().getInstance(new ZhaoLiu());
            obj.findLove();

            // 通过反编译工具可以查看源代码
            byte[] bytes = ProxyGenerator.generateProxyClass("$Proxy0", new Clas

            String fileOut = "E://$Proxy0.class";
            System.out.println("class文件输出路径:" + fileOut);
            FileOutputStream fileOutputStream = new FileOutputStream(fileOut);
            fileOutputStream.write(bytes);
            fileOutputStream.flush();
            fileOutputStream.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```




运行以上代码，我们能在 E 盘下找到一个 `$Proxy0.class` 文件。使用 Jad 反编译，得到 `$Proxy0.jad` 文件，打开它可以看到如下内容：




```java
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)

import cn.sitedev.dynamicproxy.IPerson;
import java.lang.reflect.*;

public final class $Proxy0 extends Proxy
    implements IPerson
{

    public $Proxy0(InvocationHandler invocationhandler)
    {
        super(invocationhandler);
    }

    public final boolean equals(Object obj)
    {
        try
        {
            return ((Boolean)super.h.invoke(this, m1, new Object[] {
                obj
            })).booleanValue();
        }
        catch(Error _ex) { }
        catch(Throwable throwable)
        {
            throw new UndeclaredThrowableException(throwable);
        }
    }

```

```
    public final void findLove()
    {
        try
        {
            super.h.invoke(this, m3, null);
            return;
        }
        catch(Error _ex) { }
        catch(Throwable throwable)
        {
            throw new UndeclaredThrowableException(throwable);
        }
    }

    public final String toString()
    {
        try
        {
            return (String)super.h.invoke(this, m2, null);
        }
        catch(Error _ex) { }
        catch(Throwable throwable)
        {
            throw new UndeclaredThrowableException(throwable);
        }
    }

    public final int hashCode()
    {
        try
        {
            return ((Integer)super.h.invoke(this, m0, null)).intValue();
        }
        catch(Error _ex) { }
        catch(Throwable throwable)
        {
            throw new UndeclaredThrowableException(throwable);
        }
    }

    private static Method m1;
    private static Method m3;
    private static Method m2;
```

```
    private static Method m0;

    static
    {
        try
        {
            m1 = Class.forName("java.lang.Object").getMethod("equals", new Class
                Class.forName("java.lang.Object")
            });
            m3 = Class.forName("cn.sitedev.dynamicproxy.IPerson").getMethod("fir
            m2 = Class.forName("java.lang.Object").getMethod("toString", new Cla
            m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Cla
        }
        catch(NoSuchMethodException nosuchmethodexception)
        {
            throw new NoSuchMethodError(nosuchmethodexception.getMessage());
        }
        catch(ClassNotFoundException classnotfoundexception)
        {
            throw new NoClassDefFoundError(classnotfoundexception.getMessage());
        }
    }
}
```

我们发现，$Proxy0 继承了 Proxy 类，同时还实现了 Person 接口，而且重写了 findLove() 等方法。在静态块中用反射查找到了目标对象的所有方法，而且保存了所有方法的引用，重写的方法用反射调用目标对象的方法。“小伙伴们”此时一定会好奇：这些代码是哪里来的呢？其实是 JDK 帮我们自动生成的。现在我们不依赖 JDK，自己来动态生成源代码、动态完成编译，然后替代目标对象并执行。

创建 MyInvocationHandler 接口：

```
package cn.sitedev.myjdkProxy;

import java.lang.reflect.Method;

public interface MyInvocationHandler {
    Object invoke(Object proxy, Method method, Object[] args) throws Throwable;
}
```

创建 MyProxy 类：

```
package cn.sitedev.myjdkproxy;

import javax.tools.JavaCompiler;
import javax.tools.StandardJavaFileManager;
import javax.tools.ToolProvider;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Map;

/**
 * 用来生成源代码的工具类
  */
public class MyProxy {

  public static final String ln = "\r\n";

    public static Object newProxyInstance(MyClassLoader classLoader, Class<?>[]
                                          MyInvocationHandler invocationHandler
  try {
  // 动态生成.java文件
  String src = generateSrc(interfaces);

            // Java文件输出磁盘
  String filePath = MyProxy.class.getResource("").getPath();
            File file = new File(filePath + "$Proxy0.java");
            FileWriter fileWriter = new FileWriter(file);
            fileWriter.write(src);
            fileWriter.flush();
            fileWriter.close();

            // 把生成的.java文件编译成.class文件
  JavaCompiler javaCompiler = ToolProvider.getSystemJavaCompiler();
            StandardJavaFileManager manager = javaCompiler.getStandardFileManag
            Iterable iterable = manager.getJavaFileObjects(file);
            JavaCompiler.CompilationTask task = javaCompiler.getTask(null, mana
```

```
            task.call();
            manager.close();

            // 把编译生成的.class文件加载到jvm中
Class proxyClass = classLoader.findClass("$Proxy0");
            Constructor constructor = proxyClass.getConstructor(MyInvocationHan
            file.delete();

            // 返回字节码重组以后的新的代理对象
return constructor.newInstance(invocationHandler);
      } catch (IOException | NoSuchMethodException e) {
e.printStackTrace();
      } catch (IllegalAccessException e) {
e.printStackTrace();
      } catch (InstantiationException e) {
e.printStackTrace();
      } catch (InvocationTargetException e) {
e.printStackTrace();
      }
return null;
  }

private static String generateSrc(Class<?>[] interfaces) {
StringBuffer sb = new StringBuffer();
      sb.append("package cn.sitedev.myjdkproxy;").append(ln);
      sb.append("import cn.sitedev.myjdkproxy.IPerson;").append(ln);
      sb.append("import java.lang.reflect.*;").append(ln);
      sb.append("public class $Proxy0 implements " + interfaces[0].getName() +
      sb.append("    MyInvocationHandler h;").append(ln);
      sb.append("    public $Proxy0(MyInvocationHandler h) {").append(ln);
      sb.append("        this.h = h;").append(ln);
      sb.append("    }").append(ln);
      for (Method method : interfaces[0].getMethods()) {
Class<?>[] params = method.getParameterTypes();
            StringBuffer paramNames = new StringBuffer();
            StringBuffer paramValues = new StringBuffer();
            StringBuffer paramClasses = new StringBuffer();

            for (int i = 0; i < params.length; i++) {
Class clazz = params[i];
                String type = clazz.getName();
                String paramName = toLowerFirstCase(clazz.getSimpleName());
                paramNames.append(type + " " + paramName);
```

```
                paramValues.append(paramName);
                paramClasses.append(clazz.getName() + ".class");
                if (i > 0 && i < params.length - 1) {
 paramNames.append(",");
                    paramClasses.append(",");
                    paramValues.append(",");
                }
}  sb.append("    public " + method.getReturnType().getName() + " " + method.g
            sb.append("        try {").append(ln);
            sb.append("            Method method = " + interfaces[0].getName() 
            if (hasReturnValue(method.getReturnType())) {
 sb.append("            return ");
            } else {
 sb.append("            ");
            }
 sb.append(getCaseCode("this.h.invoke(this, method, new Object[] {" + paramVal
                    method.getReturnType()) + ";").append(ln);
            sb.append("        } catch (Exception e) {").append(ln);
            sb.append("        } catch (Throwable t) {").append(ln);
            sb.append("            throw new UndeclaredThrowableException(t);")
            sb.append("        }").append(ln);
            sb.append("        " + getReturnEmptyCode(method.getReturnType())).
            sb.append("    }").append(ln);
        }
 sb.append("}").append(ln);
        return sb.toString();
    }

 private static Map<Class, Class> mappings = new HashMap<>();

    static {
 mappings.put(int.class, Integer.class);
    }

 private static String getReturnEmptyCode(Class<?> returnClass) {
 if (mappings.containsKey(returnClass)) {
 return "return 0;";
        } else if (returnClass == void.class) {
 return "";
        } else {
 return "return null";
        }
}
```

```
  private static String getCaseCode(String code, Class<?> returnClass) {
  if (mappings.containsKey(returnClass)) {
  return "((" + mappings.get(returnClass).getName() + ")" + code + ")." + retur
                    "Value()";
        }
  return code;
    }

  private static boolean hasReturnValue(Class<?> clazz) {
  return clazz != void.class;
    }

  private static String toLowerFirstCase(String src) {
  char[] chars = src.toCharArray();
        chars[0] += 32;
        return String.valueOf(chars);
    }
}
```

创建 MyClassLoader 类：

```
package cn.sitedev.myjdkproxy;

import lombok.Cleanup;

import java.io.*;

public class MyClassLoader extends ClassLoader {
  private File classPathFile;

    public MyClassLoader() {
  String classPath = MyClassLoader.class.getResource("").getPath();
        this.classPathFile = new File(classPath);
    }

  @Override
  public Class<?> findClass(String name) {
  String className = MyClassLoader.class.getPackage().getName() + "." + name;
        if (classPathFile != null) {
  File classFile = new File(classPathFile, name.replaceAll("\\\\.", "/") + ".cla
            if (classFile != null) {
```

```
  if (classFile.exists()) {
  try {
  @Cleanup FileInputStream fileInputStream = new FileInputStream(classFile);
                        @Cleanup ByteArrayOutputStream byteArrayOutputStream = r
                        byte[] buff = new byte[1024];
                        int len;
                        while ((len = fileInputStream.read(buff)) > 0) {
  byteArrayOutputStream.write(buff, 0, len);
                        }
  return defineClass(className, byteArrayOutputStream.toByteArray(), 0,
                                byteArrayOutputStream.size());
                    } catch (FileNotFoundException e) {
  e.printStackTrace();
                    } catch (IOException e) {
  e.printStackTrace();
                    }
 } } }  return null;
    }
}
```

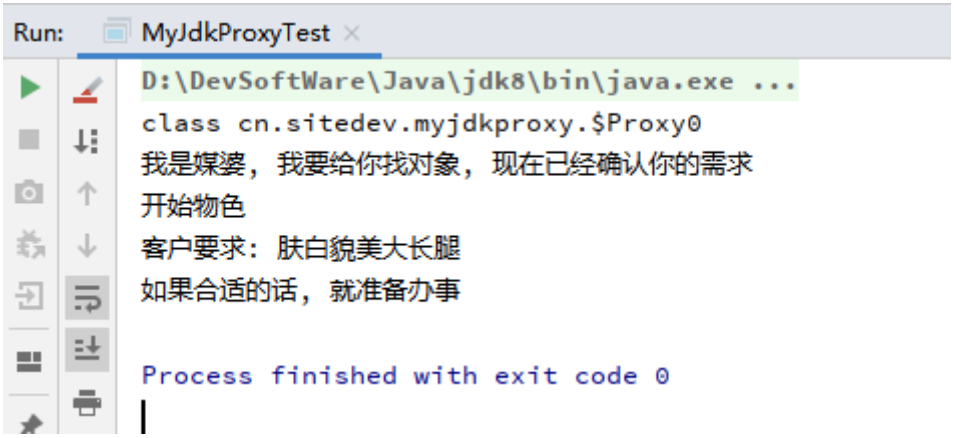
创建 MyMeipo 类：

```
package cn.sitedev.myjdkproxy;

import java.lang.reflect.Method;

public class MyMeipo implements MyInvocationHandler {
  /**
 * 被代理的对象，把引用保存下来
  */
  private Object target;

    public Object getInstance(Object target) {
  this.target = target;
        Class<?> clazz = target.getClass();
        return MyProxy.newProxyInstance(new MyClassLoader(), clazz.getInterfaces
    }

  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwa
  before();
```

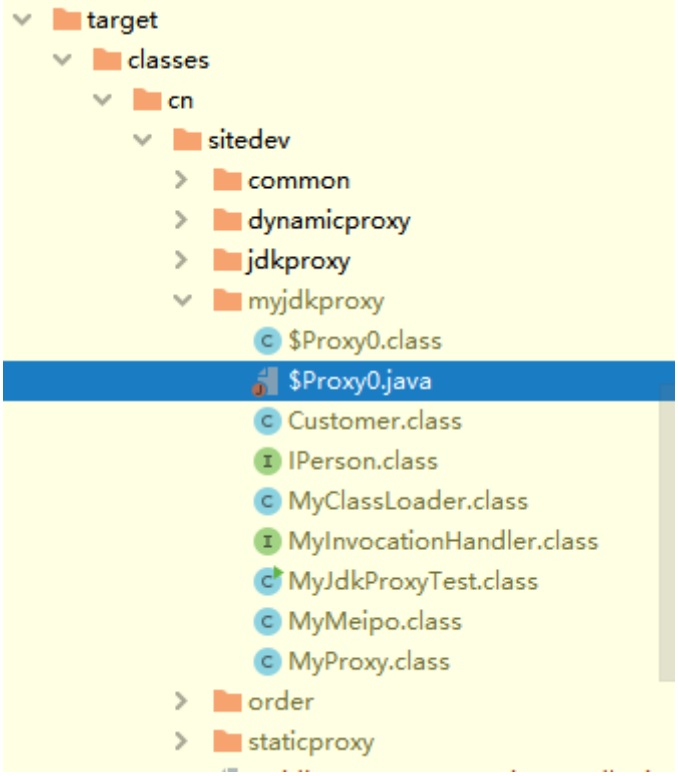
```
        method.invoke(target, args);
        after();
        return null;
    }

  private void before() {
  System.out.println("我是媒婆，我要给你找对象，现在已经确认你的需求");
        System.out.println("开始物色");
    }

  private void after() {
  System.out.println("如果合适的话，就准备办事");
    }
}
```

客户端测试代码如下：

```
package cn.sitedev.myjdkproxy;

public interface IPerson {
    void findLove();
}
////////////////////////////////////
package cn.sitedev.myjdkproxy;

public class Customer implements IPerson {
    @Override
    public void findLove() {
        System.out.println("客户要求：肤白貌美大长腿");
    }
}
////////////////////////////////////
package cn.sitedev.myjdkproxy;

public class MyJdkProxyTest {
    public static void main(String[] args) {
        IPerson person = (IPerson) new MyMeipo().getInstance(new Customer());
        System.out.println(person.getClass());
        person.findLove();
    }
}
```




生成后的 `$Proxy0.java` 文件内容如下:




```java
package cn.sitedev.myjdkproxy;
import cn.sitedev.myjdkproxy.IPerson;
import java.lang.reflect.*;
public class $Proxy0 implements cn.sitedev.myjdkproxy.IPerson{
    MyInvocationHandler h;
    public $Proxy0(MyInvocationHandler h) {
        this.h = h;
    }
    public void findLove() {
        try {
            Method method = cn.sitedev.myjdkproxy.IPerson.class.getMethod("findL
            this.h.invoke(this, method, new Object[] {});
```

```
        } catch (Exception e) {
        } catch (Throwable t) {
            throw new UndeclaredThrowableException(t);
        }

    }
}
```

到此，手写 JDK 动态代理就完成了。“小伙伴们”是不是又多了一个面试用的“撒手锏”呢？

## 1.7. CGLib代理调用API及原理分析

首先引入依赖:

```
        <dependency>
            <groupId>cglib</groupId>
            <artifactId>cglib-nodep</artifactId>
            <version>2.2</version>
        </dependency>
```

简单看一下 CGLib 代理的使用，还是以媒婆为例，创建 CglibMeipo 类：

```
package cn.sitedev.cn.sitedev.cglibproxy;

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

import java.lang.reflect.Method;

public class CglibMeipo implements MethodInterceptor {
    public Object getInstance(Class<?> clazz) {
        Enhancer enhancer = new Enhancer();
        // 要把哪个设置为即将生成的新类的父类
        enhancer.setSuperclass(clazz);
        enhancer.setCallback(this);

        return enhancer.create();
    }
```

```
18
19      @Override
20      public Object intercept(Object o, Method method, Object[] objects, MethodPro
21          // 业务的增强
22          before();
23          Object result = methodProxy.invokeSuper(o, objects);
24          after();
25          return result;
26      }
27
28      private void before() {
29          System.out.println("我是媒婆:我要给你找对象，现在已经确认你的需求");
30          System.out.println("开始物色");
31      }
32
33      private void after() {
34          System.out.println("如果合适的话，就准备办事");
35      }
36  }
```

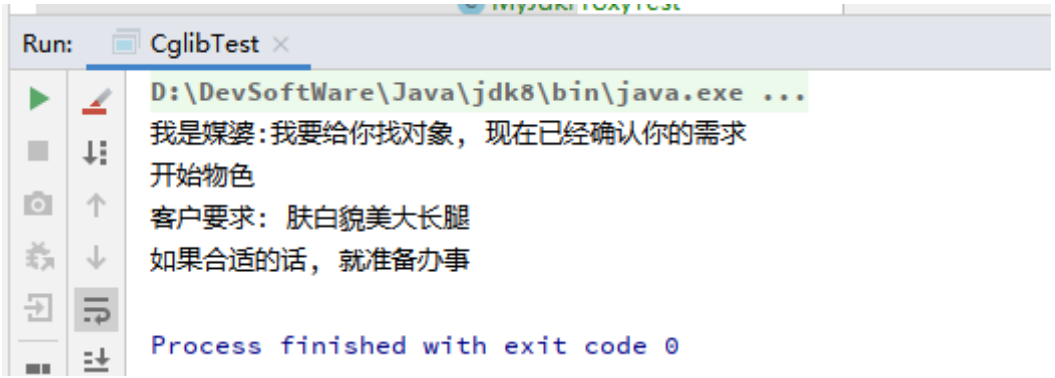
创建单身客户类 Customer：

```
1  package cn.sitedev.cn.sitedev.cglibproxy;
2
3  public class Customer {
4      public void findLove() {
5          System.out.println("客户要求：肤白貌美大长腿");
6      }
7  }
```

有个小细节，CGLib 代理的目标对象不需要实现任何接口，它是通过动态继承目标对象实现动态代理的。来看测试代码：

```
1  package cn.sitedev.cn.sitedev.cglibproxy;
2
3  public class CglibTest {
4      public static void main(String[] args) {
5          Customer customer = (Customer) new CglibMeipo().getInstance(Customer.cla
6          customer.findLove();
```

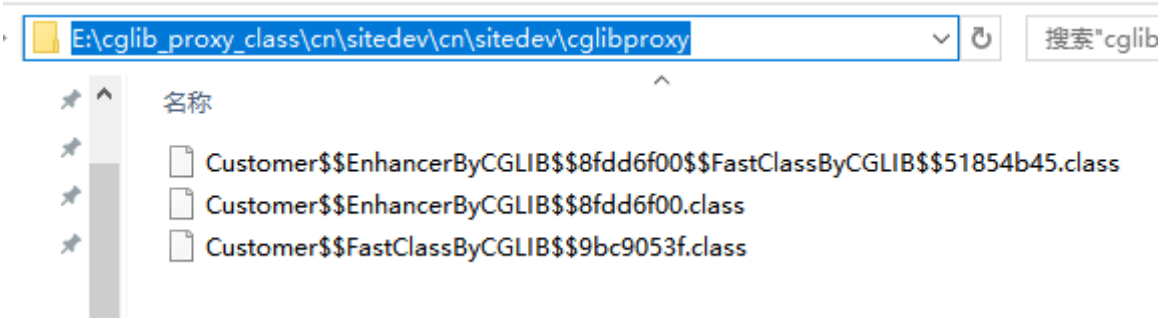
```
    }
}
```




CGLib 代理的实现原理又是怎样的呢？我们可以在测试代码中加上一句代码，将 CGLib 代理后的.class 文件写入磁盘，然后反编译来一探究竟，代码如下：

```
package cn.sitedev.cn.sitedev.cglibproxy;

import net.sf.cglib.core.DebuggingClassWriter;

public class CglibTest {
    public static void main(String[] args) {
        // 利用Cglib的代理类可以将内存中的.class文件写入本地磁盘中
        System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY, "E://cg
        Customer customer = (Customer) new CglibMeipo().getInstance(Customer.cla
        customer.findLove();
    }
}
```

重新执行代码，我们会发现在 E://cglib_proxy_class 目录下多了三个.class 文件，如下图所示。




通过调试跟踪发现，`Customer$$EnhancerByCGLIB$$8fdd6f00.class` 就是 CGLib 代理生成的代理类，继承了 Customer 类。

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

package cn.sitedev.cn.sitedev.cglibproxy;

import java.lang.reflect.Method;
import net.sf.cglib.core.ReflectUtils;
import net.sf.cglib.core.Signature;
import net.sf.cglib.proxy.Callback;
import net.sf.cglib.proxy.Factory;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

public class Customer$$EnhancerByCGLIB$$8fdd6f00 extends Customer implements Fa
    private boolean CGLIB$BOUND;
    private static final ThreadLocal CGLIB$THREAD_CALLBACKS;
    private static final Callback[] CGLIB$STATIC_CALLBACKS;
    private MethodInterceptor CGLIB$CALLBACK_0;
    private static final Method CGLIB$findLove$0$Method;
    private static final MethodProxy CGLIB$findLove$0$Proxy;
    private static final Object[] CGLIB$emptyArgs;
    private static final Method CGLIB$finalize$1$Method;
    private static final MethodProxy CGLIB$finalize$1$Proxy;
    private static final Method CGLIB$equals$2$Method;
    private static final MethodProxy CGLIB$equals$2$Proxy;
    private static final Method CGLIB$toString$3$Method;
    private static final MethodProxy CGLIB$toString$3$Proxy;
    private static final Method CGLIB$hashCode$4$Method;
    private static final MethodProxy CGLIB$hashCode$4$Proxy;
    private static final Method CGLIB$clone$5$Method;
    private static final MethodProxy CGLIB$clone$5$Proxy;

    static void CGLIB$STATICHOOK1() {
        CGLIB$THREAD_CALLBACKS = new ThreadLocal();
        CGLIB$emptyArgs = new Object[0];
        Class var0 = Class.forName("cn.sitedev.cn.sitedev.cglibproxy.Customer$$
        Class var1;
        CGLIB$findLove$0$Method = ReflectUtils.findMethods(new String[]{"findLo
        CGLIB$findLove$0$Proxy = MethodProxy.create(var1, var0, "()V", "findLov
        Method[] var10000 = ReflectUtils.findMethods(new String[]{"finalize", "
```

```
        CGLIB$finalize$1$Method = var10000[0];
        CGLIB$finalize$1$Proxy = MethodProxy.create(var1, var0, "()V", "finaliz
        CGLIB$equals$2$Method = var10000[1];
        CGLIB$equals$2$Proxy = MethodProxy.create(var1, var0, "(Ljava/lang/Obje
        CGLIB$toString$3$Method = var10000[2];
        CGLIB$toString$3$Proxy = MethodProxy.create(var1, var0, "()Ljava/lang/S
        CGLIB$hashCode$4$Method = var10000[3];
        CGLIB$hashCode$4$Proxy = MethodProxy.create(var1, var0, "()I", "hashCod
        CGLIB$clone$5$Method = var10000[4];
        CGLIB$clone$5$Proxy = MethodProxy.create(var1, var0, "()Ljava/lang/Obje
    }

    final void CGLIB$findLove$0() {
        super.findLove();
    }

    public final void findLove() {
        MethodInterceptor var10000 = this.CGLIB$CALLBACK_0;
        if (var10000 == null) {
            CGLIB$BIND_CALLBACKS(this);
            var10000 = this.CGLIB$CALLBACK_0;
        }

        if (var10000 != null) {
            var10000.intercept(this, CGLIB$findLove$0$Method, CGLIB$emptyArgs, 
        } else {
            super.findLove();
        }
    }

    final void CGLIB$finalize$1() throws Throwable {
        super.finalize();
    }

    protected final void finalize() throws Throwable {
        MethodInterceptor var10000 = this.CGLIB$CALLBACK_0;
        if (var10000 == null) {
            CGLIB$BIND_CALLBACKS(this);
            var10000 = this.CGLIB$CALLBACK_0;
        }

        if (var10000 != null) {
            var10000.intercept(this, CGLIB$finalize$1$Method, CGLIB$emptyArgs, 
```

```
        } else {
            super.finalize();
        }
    }

    final boolean CGLIB$equals$2(Object var1) {
        return super.equals(var1);
    }

    public final boolean equals(Object var1) {
        MethodInterceptor var10000 = this.CGLIB$CALLBACK_0;
        if (var10000 == null) {
            CGLIB$BIND_CALLBACKS(this);
            var10000 = this.CGLIB$CALLBACK_0;
        }

        if (var10000 != null) {
            Object var2 = var10000.intercept(this, CGLIB$equals$2$Method, new O
            return var2 == null ? false : (Boolean)var2;
        } else {
            return super.equals(var1);
        }
    }

    final String CGLIB$toString$3() {
        return super.toString();
    }

    public final String toString() {
        MethodInterceptor var10000 = this.CGLIB$CALLBACK_0;
        if (var10000 == null) {
            CGLIB$BIND_CALLBACKS(this);
            var10000 = this.CGLIB$CALLBACK_0;
        }

        return var10000 != null ? (String)var10000.intercept(this, CGLIB$toStri
    }

    final int CGLIB$hashCode$4() {
        return super.hashCode();
    }

    public final int hashCode() {
```

```
129         MethodInterceptor var10000 = this.CGLIB$CALLBACK_0;
130         if (var10000 == null) {
131             CGLIB$BIND_CALLBACKS(this);
132             var10000 = this.CGLIB$CALLBACK_0;
133         }
134 
135         if (var10000 != null) {
136             Object var1 = var10000.intercept(this, CGLIB$hashCode$4$Method, CGL
137             return var1 == null ? 0 : ((Number)var1).intValue();
138         } else {
139             return super.hashCode();
140         }
141     }
142 
143     final Object CGLIB$clone$5() throws CloneNotSupportedException {
144         return super.clone();
145     }
146 
147     protected final Object clone() throws CloneNotSupportedException {
148         MethodInterceptor var10000 = this.CGLIB$CALLBACK_0;
149         if (var10000 == null) {
150             CGLIB$BIND_CALLBACKS(this);
151             var10000 = this.CGLIB$CALLBACK_0;
152         }
153 
154         return var10000 != null ? var10000.intercept(this, CGLIB$clone$5$Method
155     }
156 
157     public static MethodProxy CGLIB$findMethodProxy(Signature var0) {
158         String var10000 = var0.toString();
159         switch(var10000.hashCode()) {
160         case -1574182249:
161             if (var10000.equals("finalize()V")) {
162                 return CGLIB$finalize$1$Proxy;
163             }
164             break;
165         case -508378822:
166             if (var10000.equals("clone()Ljava/lang/Object;")) {
167                 return CGLIB$clone$5$Proxy;
168             }
169             break;
170         case 1192015562:
171             if (var10000.equals("findLove()V")) {
```

```
                return CGLIB$findLove$0$Proxy;
            }
            break;
        case 1826985398:
            if (var10000.equals("equals(Ljava/lang/Object;)Z")) {
                return CGLIB$equals$2$Proxy;
            }
            break;
        case 1913648695:
            if (var10000.equals("toString()Ljava/lang/String;")) {
                return CGLIB$toString$3$Proxy;
            }
            break;
        case 1984935277:
            if (var10000.equals("hashCode()I")) {
                return CGLIB$hashCode$4$Proxy;
            }
        }

        return null;
    }

    public Customer$$EnhancerByCGLIB$$8fdd6f00() {
        CGLIB$BIND_CALLBACKS(this);
    }

    public static void CGLIB$SET_THREAD_CALLBACKS(Callback[] var0) {
        CGLIB$THREAD_CALLBACKS.set(var0);
    }

    public static void CGLIB$SET_STATIC_CALLBACKS(Callback[] var0) {
        CGLIB$STATIC_CALLBACKS = var0;
    }

    private static final void CGLIB$BIND_CALLBACKS(Object var0) {
        Customer$$EnhancerByCGLIB$$8fdd6f00 var1 = (Customer$$EnhancerByCGLIB$$
        if (!var1.CGLIB$BOUND) {
            var1.CGLIB$BOUND = true;
            Object var10000 = CGLIB$THREAD_CALLBACKS.get();
            if (var10000 == null) {
                var10000 = CGLIB$STATIC_CALLBACKS;
                if (var10000 == null) {
                    return;
```

```
215                 }
216             }
217 
218             var1.CGLIB$CALLBACK_0 = (MethodInterceptor)((Callback[])var10000)[0
219         }
220 
221     }
222 
223     public Object newInstance(Callback[] var1) {
224         CGLIB$SET_THREAD_CALLBACKS(var1);
225         Customer$$EnhancerByCGLIB$$8fdd6f00 var10000 = new Customer$$EnhancerBy
226         CGLIB$SET_THREAD_CALLBACKS((Callback[])null);
227         return var10000;
228     }
229 
230     public Object newInstance(Callback var1) {
231         CGLIB$SET_THREAD_CALLBACKS(new Callback[]{var1});
232         Customer$$EnhancerByCGLIB$$8fdd6f00 var10000 = new Customer$$EnhancerBy
233         CGLIB$SET_THREAD_CALLBACKS((Callback[])null);
234         return var10000;
235     }
236 
237     public Object newInstance(Class[] var1, Object[] var2, Callback[] var3) {
238         CGLIB$SET_THREAD_CALLBACKS(var3);
239         Customer$$EnhancerByCGLIB$$8fdd6f00 var10000 = new Customer$$EnhancerBy
240         switch(var1.length) {
241         case 0:
242             var10000.<init>();
243             CGLIB$SET_THREAD_CALLBACKS((Callback[])null);
244             return var10000;
245         default:
246             throw new IllegalArgumentException("Constructor not found");
247         }
248     }
249 
250     public Callback getCallback(int var1) {
251         CGLIB$BIND_CALLBACKS(this);
252         MethodInterceptor var10000;
253         switch(var1) {
254         case 0:
255             var10000 = this.CGLIB$CALLBACK_0;
256             break;
257         default:
```

```
            var10000 = null;
        }

        return var10000;
    }

    public void setCallback(int var1, Callback var2) {
        switch(var1) {
        case 0:
            this.CGLIB$CALLBACK_0 = (MethodInterceptor)var2;
        default:
        }
    }

    public Callback[] getCallbacks() {
        CGLIB$BIND_CALLBACKS(this);
        return new Callback[]{this.CGLIB$CALLBACK_0};
    }

    public void setCallbacks(Callback[] var1) {
        this.CGLIB$CALLBACK_0 = (MethodInterceptor)var1[0];
    }

    static {
        CGLIB$STATICHOOK1();
    }
}
```

我们重写了 Customer 类的所有方法，通过代理类的源码可以看到，代理类会获得所有从父类继承来的方法，并且会有 MethodProxy 与之对应，比如 `Method CGLIB$findLove$0$Method`、`MethodProxy CGLIB$findLove$0$Proxy` 这些方法在代理类的 findLove()方法中都有调用。

```
    // 代理方法(methodProxy.invokeSuper()方法会调用)
    final void CGLIB$findLove$0() {
        super.findLove();
    }

    // 被代理方法(methodProxy.invoke()方法会调用, 这就是为什么在拦截器中调用methodPr
    public final void findLove() {
        MethodInterceptor var10000 = this.CGLIB$CALLBACK_0;
```

```
        if (var10000 == null) {
            CGLIB$BIND_CALLBACKS(this);
            var10000 = this.CGLIB$CALLBACK_0;
        }

        if (var10000 != null) {
            // 调用拦截器
            var10000.intercept(this, CGLIB$findLove$0$Method, CGLIB$emptyArgs, C
        } else {
            super.findLove();
        }
    }
```

调用过程为：代理对象调用 this.findLove()方法→调用拦截器
→ `methodProxy.invokeSuper` → `CGLIB$findLove$0` →被代理对象 findLove()方法。

此时，我们发现拦截器 MethodInterceptor 中就是由 MethodProxy 的 invokeSuper()方法调用代理方法的，MethodProxy 非常关键，我们分析一下它具体做了什么。

```
public class MethodProxy {
    private Signature sig1;
    private Signature sig2;
    private CreateInfo createInfo;

    private final Object initLock = new Object();
    private volatile FastClassInfo fastClassInfo;

    public static MethodProxy create(Class c1, Class c2, String desc, String nam
        MethodProxy proxy = new MethodProxy();
        proxy.sig1 = new Signature(name1, desc);
        proxy.sig2 = new Signature(name2, desc);
        proxy.createInfo = new CreateInfo(c1, c2);
        return proxy;
    }
    ...


    private static class CreateInfo
    {
        Class c1;
        Class c2;
```

```
        NamingPolicy namingPolicy;
        GeneratorStrategy strategy;
        boolean attemptLoad;

        public CreateInfo(Class c1, Class c2)
        {
            this.c1 = c1;
            this.c2 = c2;
            AbstractClassGenerator fromEnhancer = AbstractClassGenerator.getCurr
            if (fromEnhancer != null) {
                namingPolicy = fromEnhancer.getNamingPolicy();
                strategy = fromEnhancer.getStrategy();
                attemptLoad = fromEnhancer.getAttemptLoad();
            }
        }
    }
    ...
```

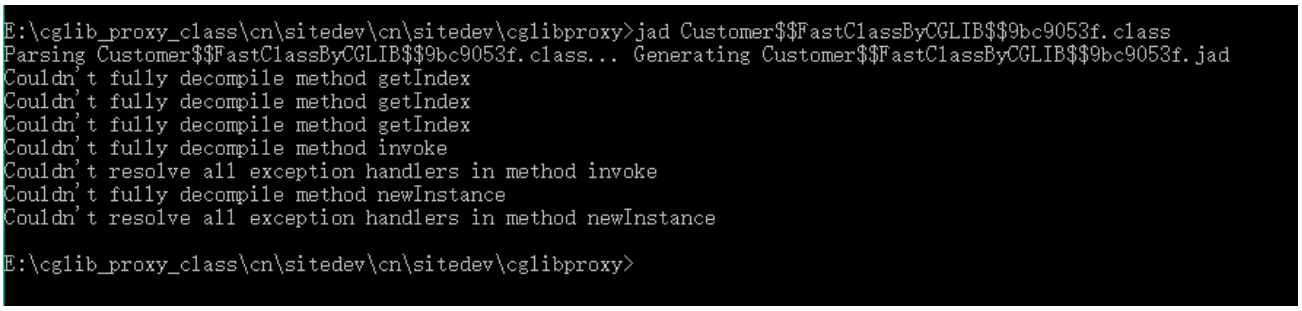
继续看 invokeSuper()方法：

```
    public Object invokeSuper(Object obj, Object[] args) throws Throwable {
        try {
            init();
            FastClassInfo fci = fastClassInfo;
            return fci.f2.invoke(fci.i2, obj, args);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        }
    }

    ...
    private static class FastClassInfo
    {
        FastClass f1;
        FastClass f2;
        int i1;
        int i2;
    }
```

上面的代码调用就是获取代理类对应的 FastClass，并执行代理方法。还记得之前生成的三个.class文件吗？

`Customer$$EnhancerByCGLIB$$8fdd6f00$$FastClassByCGLIB$$51854b45.class` 就是代理类的 FastClass，`Customer$$FastClassByCGLIB$$9bc9053f.class` 就是被代理类的 FastClass。

CGLib 代理执行代理方法的效率之所以比 JDK 的高，是因为 CGlib 采用了 FastClass 机制，它的原理简单来说就是：为代理类和被代理类各生成一个类，这个类会为代理类或被代理类的方法分配一个index（int 类型）；这个 index 当作一个入参，FastClass 就可以直接定位要调用的方法并直接进行调用，省去了反射调用，所以调用效率比 JDK 代理通过反射调用高。下面我们反编译一个 FastClass 看看：




```
// Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   <generated>

package cn.sitedev.cn.sitedev.cglibproxy;

import java.lang.reflect.InvocationTargetException;
import net.sf.cglib.core.Signature;
import net.sf.cglib.reflect.FastClass;

// Referenced classes of package cn.sitedev.cn.sitedev.cglibproxy:
//            Customer

public class Customer$$FastClassByCGLIB$$9bc9053f extends FastClass
{

    public int getIndex(Signature signature)
    {
        String s = signature.toString();
        s;
        s.hashCode();
```

```
        JVM INSTR lookupswitch 10: default 204
        ...
        JVM INSTR pop ;
        return -1;
    }

    ...

    // 根据index直接定位执行方法
    public Object invoke(int i, Object obj, Object aobj[])
        throws InvocationTargetException
    {
        (Customer)obj;
        i;
        JVM INSTR tableswitch 0 9: default 152
    //                   0 60
    //                   1 65
    //                   2 70
    //                   3 93
    //                   4 107
    //                   5 122
    //                   6 126
    //                   7 138
    //                   8 142
    //                   9 147;
           goto _L1 _L2 _L3 _L4 _L5 _L6 _L7 _L8 _L9 _L10 _L11
_L2:
        findLove();
        return null;
_L3:
        wait();
        return null;
_L4:
        ((Number)aobj[0]).longValue();
        ((Number)aobj[1]).intValue();
        wait();
        return null;
_L5:
        ((Number)aobj[0]).longValue();
        wait();
        return null;
_L6:
        aobj[0];
```

```
        equals();
        JVM INSTR new #100 <Class Boolean>;
        JVM INSTR dup_x1 ;
        JVM INSTR swap ;
        Boolean();
        return;
_L7:
        toString();
        return;
_L8:
        hashCode();
        JVM INSTR new #107 <Class Integer>;
        JVM INSTR dup_x1 ;
        JVM INSTR swap ;
        Integer();
        return;
_L9:
        getClass();
        return;
_L10:
        notify();
        return null;
_L11:
        notifyAll();
        return null;
        JVM INSTR new #75  <Class InvocationTargetException>;
        JVM INSTR dup_x1 ;
        JVM INSTR swap ;
        InvocationTargetException();
        throw ;
_L1:
        throw new IllegalArgumentException("Cannot find matching method/construc
    }
    ...
```

FastClass 并不是跟代理类一起生成的，而是在第一次执行 MethodProxy 的 invoke()或 invokeSuper()方法时生成的，并放在了缓存中。

```
    // MethodProxy的invoke()或invokeSuper()方法都调用了init()方法
    private void init()
    {
```

```
        if (fastClassInfo == null)
        {
            synchronized (initLock)
            {
                if (fastClassInfo == null)
                {
                    CreateInfo ci = createInfo;

                    FastClassInfo fci = new FastClassInfo();
                    fci.f1 = helper(ci, ci.c1); // 如果在缓存中就取出;如果没有在缓
                    fci.f2 = helper(ci, ci.c2);
                    fci.i1 = fci.f1.getIndex(sig1); // 获取方法的index
                    fci.i2 = fci.f2.getIndex(sig2);
                    fastClassInfo = fci;
                }
            }
        }
    }
```

至此，CGLib 代理的原理我们就基本搞清楚了，对代码细节有兴趣的“小伙伴”可以自行深入研究。

# 1.8. CGLib和JDK动态代理对比

（1）JDK 动态代理实现了被代理对象的接口，CGLib 代理继承了被代理对象。
（2）JDK 动态代理和 CGLib 代理都在运行期生成字节码，JDK 动态代理直接写 Class 字节码，CGLib代理使用 ASM 框架写 Class 字节码，CGlib 代理实现更复杂，生成代理类比 JDK 动态代理效率低。
（3）JDK 动态代理调用代理方法是通过反射机制调用的，CGLib 代理是通过 FastClass 机制直接调用方法的，CGLib 代理的执行效率更高。

# 1.9. 代理模式于Spring生态

## 1.9.1. 代理模式在 Spring 中的应用

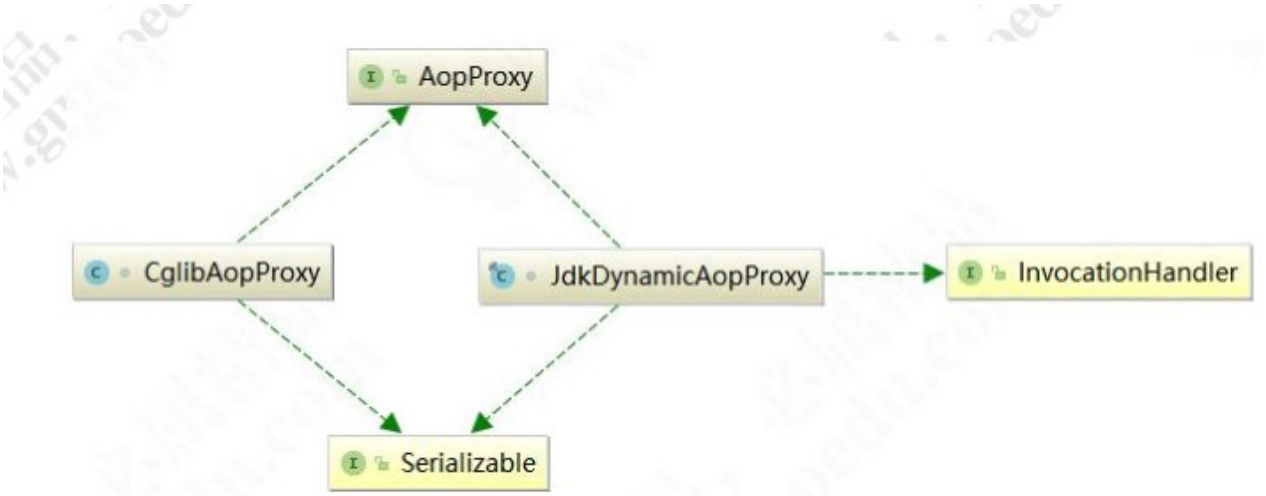
先看 ProxyFactoryBean 核心方法 getObject()，源码如下

```
public Object getObject() throws BeansException {
    initializeAdvisorChain();
```

```
    if (isSingleton()) {
        return getSingletonInstance();
    }
    else {
        if (this.targetName == null) {
            logger.warn("Using non-singleton proxies with singleton targets is c
            "Enable prototype proxies by setting the 'targetName' property.");
        }
        return newPrototypeInstance();
    }
}
```

在 getObject()方法中，主要调用 getSingletonInstance()和 newPrototypeInstance()。在 Spring的配置中如果不做任何设置，那么 Spring 代理生成的 Bean 都是单例对象。如果修改 scope，则每次创建一个新的原型对象。newPrototypeInstance()里面的逻辑比较复杂，我们后面再做深入研究，这里先做简单了解。

Spring 利用动态代理实现 AOP 时有两个非常重要的类：JdkDynamicAopProxy 类和CglibAopProxy 类，来看一下类图，如下图所示。




## 1.9.2. Spring中的代理选择原则

（1）当 Bean 有实现接口时，Spring 就会用 JDK 动态代理。
（2）当 Bean 没有实现接口时，Spring 会选择 CGLib 代理。
（3）Spring 可以通过配置强制使用 CGLib 代理，只需在 Spring 的配置文件中加入如下代码：

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

## 1.10. 静态代理和动态代理的本质区别

（1）静态代理只能通过手动完成代理操作，如果被代理类增加了新的方法，代理类需要同步增加，违背开闭原则。
（2）动态代理采用在运行时动态生成代码的方式，取消了对被代理类的扩展限制，遵循开闭原则。
（3）若动态代理要对目标类的增强逻辑进行扩展，结合策略模式，只需要新增策略类便可完成，无须修改代理类的代码。

## 1.11. 代理模式的优缺点

代理模式具有以下优点：
（1）代理模式能将代理对象与真实被调用目标对象分离。
（2）在一定程度上降低了系统的耦合性，扩展性好。
（3）可以起到保护目标对象的作用。
（4）可以增强目标对象的功能。

当然，代理模式也有缺点：
（1）代理模式会造成系统设计中类的数量增加。
（2）在客户端和目标对象中增加一个代理对象，会导致请求处理速度变慢。
（3）增加了系统的复杂度。