链滴

Java 源码剖析——动态代理的实现原理

作者：jesministrator

原文链接：https://ld246.com/article/1511708443762

来源网站：链滴

在本篇博客中，博主将和大家一起深入分析Jdk自带的动态代理的实现原理。如果有同学对**代理模式**，**态代理**和**动态代理**这些概念比较模糊，请先阅读博主的另一篇文章《一步一步学设计模式——代理模》。

为了方便讲解，我们继续使用代理模式中的购票例子，下面是这个例子的主要代码：

- 首先我们先建立一个接口：

```
package com.wxueyuan.DesignPettern.StaticProxy;

public interface Operation {
    void buyTicket(Ticket t);
}
```

- 接着我们建立一个学生类并实现上面的接口，表示学生需要购票：

```
package com.wxueyuan.DesignPettern.StaticProxy;

public class Student implements Operation{

    @Override
    public void buyTicket(Ticket t) {
        // TODO Auto-generated method stub
        System.out.println("学生买到一张票,票价为"+t.getPrice());
    }

}
```

- 然后我们建立一个TicketOperationInvocationHandler实现InvocationHandler接口：

```
package com.wxueyuan.DesignPettern.DynamicProxy;

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

public class TicketOperationInvocationHandler implements InvocationHandler {

    //将需要代理的委托对象传入Handler中
    private Object target;

    public TicketOperationInvocationHandler(Object target) {
        this.target = target;
    }

    //获得帮助购票者买票的代理
    public Object getProxy() {
        return Proxy.newProxyInstance(Thread.currentThread()
                .getContextClassLoader(), target.getClass().getInterfaces(),
                this);
    }

    //实际上黄牛执行的购票操作
```

```
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // TODO Auto-generated method stub
        System.out.println("黄牛收取购票者的钱");
        System.out.println("黄牛连夜排队");
        Object ret = method.invoke(target, args);
        System.out.println("黄牛将票交给购票者");
        return ret;
    }

}
```

- 最后是测试类

```
package com.wxueyuan.DesignPettern.DynamicProxy;

import com.wxueyuan.DesignPettern.StaticProxy.Operation;
import com.wxueyuan.DesignPettern.StaticProxy.Student;
import com.wxueyuan.DesignPettern.StaticProxy.Ticket;

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        //学生需要购买的ticket实例
        Ticket studentTicket = new Ticket(200);

        //创建为学生买票的代理
        Operation studentProxy = (Operation) new TicketOperationInvocationHandler(new Stud
nt()).getProxy();
        studentProxy.buyTicket(studentTicket);

    }

}
```

执行结果为：

黄牛收取购票者的钱

黄牛连夜排队

购票者买到一张票,票价为200.0

黄牛将票交给学生

下面我们就一步一步地分析这个简单的动态代理例子的原理：

首先我们先看我们是如何获得学生的代理的：

```
Operation studentProxy = (Operation) new TicketOperationInvocationHandler(new Student())
getProxy();
```

其中的关键就在于我们自定义的InvocationHandler中的getProxy()方法，现在我们就进入这个方法

一下:

```
public Object getProxy() {
        return Proxy.newProxyInstance(Thread.currentThread()
                .getContextClassLoader(), target.getClass().getInterfaces(),
                this);
    }
```

这个方法的核心就是使用Proxy类的静态方法newProxyInstance()，我们具体看一下这个方法究竟在什么，我们以Jdk1.8的源码为例,首先来看一下这个方法的注释：

```
/**
     * Returns an instance of a proxy class for the specified interfaces
     * that dispatches method invocations to the specified invocation
     * handler.
     * @param   loader the class loader to define the proxy class
     * @param   interfaces the list of interfaces for the proxy class
     *          to implement
     * @param   h the invocation handler to dispatch method invocations to
```

这个方法用来返回一个实现了一个或多个指定接口的代理类的实例，这个代理类能够将其实现的方法调用传递给指定的方法调用处理器。
参数：
loader：加载这个代理类的类加载器
interfaces：这个代理类实现的所有接口
h：将方法调用传至的调用处理器

newProxyInstance()这个方法生成实例的核心代码有以下几句：

```
    //获得代理类的class类
    Class<?> cl = getProxyClass0(loader, intfs);
    ...
    //获取代理类的构造函数
    final Constructor<?> cons = cl.getConstructor(constructorParams);
    ...
    //根据构造函数生成一个代理类的实例，至于为什么代理类的构造方法中的参数是方法参数h，稍我们就会知道了
    return cons.newInstance(new Object[]{h});
```

从newProxyInstance()方法中的三行核心代码可以看出，如何获取代理类的class类是重中之重，因获取class类之后，我们就可以利用Java反射获取构造函数并生成实例了。由于反射并不是本篇博客的题，我们现在就来着重关注一下getProxyClass0()方法是如何工作的：

```
private static Class<?> getProxyClass0(ClassLoader loader,
                                       Class<?>... interfaces) {
        if (interfaces.length > 65535) {
            throw new IllegalArgumentException("interface limit exceeded");
        }

        //如果代理类的class在缓存中存在，则直接获取，否则的话，通过ProxyClassFactory来创建
        return proxyClassCache.get(loader, interfaces);
    }
```

这里的proxyClassCache是在Proxy类中声明的WeakCache的实例，那我们就一起来看看这个WeakCche是什么：

```
/**
 * Cache mapping pairs of (key, sub-key) -> value.
 * Keys and values are weakly but sub-keys are strongly referenced.
 * Keys are passed directly to get method which also takes a parameter.
 * Sub-keys are calculated from keys and parameters using the subKeyFactory function
 * passed to the constructor.
 * Values are calculated from keys and parameters using the valueFactory function passed
 * to the constructor.
*/

//这个WeakCache能够将一组(key,sub-key)的值映射成value的值。其中Key的值是直接通过参数传的。
//sub - key的值是通过构造方法中的subKeyFactory生成的，value的值是通过构造方法中的valueFacory生成的。
//它的构造方法是:
final class WeakCache<K,P,V> {
  ...

  public WeakCache(BiFunction<K, P, ?> subKeyFactory,BiFunction<K, P, V> valueFactory) {
    this.subKeyFactory = Objects.requireNonNull(subKeyFactory);
    this.valueFactory = Objects.requireNonNull(valueFactory);
  }

  ...
}
```

知道了WeakCache的构造方法之后，我们一起来看一下我们在getProxyClass0方法中使用到的Weakache的get方法,它的核心代码如下：

```
public V get(K key, P parameter) {
  //通过subKeyFactory的apply方法生成subKey
  Object subKey = Objects.requireNonNull(subKeyFactory.apply(key, parameter));
  ...
  //通过subKey从valuesMap中取出可能存在的缓存提供者supplier
  Supplier<V> supplier = valuesMap.get(subKey);
  ...
  //如果供应者不为空，就调用supplier.get()方法，get方法的返回值就是我们这个方法的返回值
  if (supplier != null) {
    // 这里的供应者有可能是一个factory或者是一个缓存实例
    V value = supplier.get();
    if (value != null) {
        return value;
    }
  }
  ...
  //如果factory没有成功创建，我们此时创建一个factory
  if (factory == null) {
    factory = new Factory(key, parameter, subKey, valuesMap);
  }
  ...
}
```

看到这大家也许会问，上面代码中factory成功创建之后，如果supplier就是factory,该如何通过factor的get()方法来返回我们需要的value值，这个value值又是怎么计算得到的呢？原来Factory这个类也

现了Supplier这个函数式接口，因此它也实现了自己的get方法：

```
private final class Factory implements Supplier<V> {
   @Override
       public synchronized V get() {
          ...
       V value = null;
          try {
       //通过我们的valueFactory的apply方法生成value
              value = Objects.requireNonNull(valueFactory.apply(key, parameter));
          }
       ...
       return  value;
    }
}
```

知道了WeakCache中get()方法的实现后，我们看一下在Proxy类中，是如何定义这个WeakCache类的proxyClassCache的呢?

```
//根据WeakCache的构造函数可知，KeyFactory就是在get方法中生成sub - key的subKeyFactory；
//ProxyClassFactory就是get方法中生成value的valueFactory
private static final WeakCache<ClassLoader, Class<?>[], Class<?>>
       proxyClassCache = new WeakCache<>(new KeyFactory(), new ProxyClassFactory());
```

看到这里读者们应该知道WeakCache中的get方法是怎么工作的了吧？**它利用subKeyFactory(在Proxy类中就是KeyFactory)来生成subKey，再利用valueFactory(在Proxy类中就是ProxyClassFactor)的apply方法生成并返回value值。**那么我们一起来看一下，KeyFactory是如何生成subKey的：

```
//其实很简单就是根据参数intefaces的数量，来生成不同的subKey对象
private static final class KeyFactory
       implements BiFunction<ClassLoader, Class<?>[], Object>
   {
       @Override
       public Object apply(ClassLoader classLoader, Class<?>[] interfaces) {
          switch (interfaces.length) {
              case 1: return new Key1(interfaces[0]); // 代理类只实现了一个接口
              case 2: return new Key2(interfaces[0], interfaces[1]);//代理类实现了两个接口
              case 0: return key0;//代理类没有实现接口
              default: return new KeyX(interfaces);//代理类实现了三个及以上的接口
          }
       }
   }
```

然后是ProxyClassFactory是如何生成value,也就是我们这里需要的代理类的class类的：

```
private static final class ProxyClassFactory
       implements BiFunction<ClassLoader, Class<?>[], Class<?>>
{
  //代理类名字的前缀为$Proxy
  private static final String proxyClassNamePrefix = "$Proxy";
  //为了生成唯一的代理类名的计数器
  private static final AtomicLong nextUniqueNumber = new AtomicLong();

  @Override
  public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {
```

```
    ...

    String proxyPkg = null;     // 代理类的包名
    //对于非公共接口，代理类的包名与接口的包名相同
    for (Class<?> intf : interfaces) {
            int flags = intf.getModifiers();
            if (!Modifier.isPublic(flags)) {
                accessFlags = Modifier.FINAL;
                String name = intf.getName();
                int n = name.lastIndexOf('.');
                String pkg = ((n == -1) ? "" : name.substring(0, n + 1));
                if (proxyPkg == null) {
                    proxyPkg = pkg;
                } else if (!pkg.equals(proxyPkg)) {
                    throw new IllegalArgumentException(
                        "non-public interfaces from different packages");
                }
            }
        }

    }

    if (proxyPkg == null) {
      // 如果没有非公共的接口，就使用com.sun.proxy作为包名
      proxyPkg = ReflectUtil.PROXY_PACKAGE + ".";
    }
    //默认生成的公共代理类的全限定名为com.sun.proxy.$Proxy0,com.sun.proxy.$Proxy1,以此数字
增
    long num = nextUniqueNumber.getAndIncrement();
    String proxyName = proxyPkg + proxyClassNamePrefix + num;

    //生成代理类的字节码
    byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
        proxyName, interfaces, accessFlags);

    try {
        //根据上面产生的字节码产生Class实例并返回，至此我们终于获得了代理类的Class实例
        return defineClass0(loader, proxyName,
                proxyClassFile, 0, proxyClassFile.length);
    } catch (ClassFormatError e) {
            throw new IllegalArgumentException(e.toString());
    }
}
```

ProxyGenerator.generateProxyClass这个方法的源码并没有公开，我们可以反编译class文件，然后单看一下：

```
 public static byte[] generateProxyClass(final String var0, Class[] var1) {
    ProxyGenerator var2 = new ProxyGenerator(var0, var1);
    final byte[] var3 = var2.generateClassFile();
    // 这里根据参数配置，决定是否把生成的字节码（.class文件）保存到本地磁盘，默认是不保存的
    if(saveGeneratedFiles) {
        AccessController.doPrivileged(new PrivilegedAction() {
```

```
            public Void run() {
                try {
                    FileOutputStream var1 = new FileOutputStream(ProxyGenerator.dotToSlash(var0)
+ ".class");
                    var1.write(var3);
                    var1.close();
                    return null;
                } catch (IOException var2) {
                    throw new InternalError("I/O exception saving generated file: " + var2);
                }
            }
        });
    }
    return var3;
}
```

我们可以通过设置sun.misc.ProxyGenerator.saveGeneratedFiles这个boolean值的属性，来使方法认将class文件保存到磁盘。那么我们就来修改一下我们的Test代码，来将生成的proxy文件保存到磁上：

```
public static void main(String[] args) {
        // TODO Auto-generated method stub

    //学生需要购买的ticket实例
    Ticket studentTicket = new Ticket(200);

    //将是否在系统属性修改为true，使字节文件保存到磁盘上
    System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");
        //显示生成的代理类的全限定名
    System.out.println(Proxy.getProxyClass(Operation.class.getClassLoader(), Operation.class));
    //创建为学生买票的黄牛代理
    Operation studentProxy = (Operation) new TicketOperationInvocationHandler(new Student
)).getProxy();
    studentProxy.buyTicket(studentTicket);

    }
```

执行结果为：

class com.sun.proxy.$Proxy0

黄牛收取购票者的钱

黄牛连夜排队

学生买到一张票,票价为200.0

黄牛将票交给购票者

同时，在com/sun/proxy下生成了Proxy0.class，(注：如果我们的接类不是public的，那么我们生成的Proxy0代理类的包名会和接口类的包名相同哦)。下面我们将这个节码反编译看一下：

```
package com.sun.proxy;

import com.wxueyuan.DesignPettern.StaticProxy.Ticket;
import java.lang.reflect.UndeclaredThrowableException;
import java.lang.reflect.InvocationHandler;
```

```
import java.lang.reflect.Method;
import com.wxueyuan.DesignPettern.StaticProxy.Operation;
import java.lang.reflect.Proxy;

public final class $Proxy0 extends Proxy implements Operation
{
    private static Method m1;
    private static Method m2;
    private static Method m3;
    private static Method m0;

    public $Proxy0(final InvocationHandler invocationHandler) {
        super(invocationHandler);
    }

    public final boolean equals(final Object o) {
        try {
            return (boolean)super.h.invoke(this, $Proxy0.m1, new Object[] { o });
        }
        catch (Error | RuntimeException error) {
            throw;
        }
        catch (Throwable t) {
            throw new UndeclaredThrowableException(t);
        }
    }

    public final String toString() {
        try {
            return (String)super.h.invoke(this, $Proxy0.m2, null);
        }
        catch (Error | RuntimeException error) {
            throw;
        }
        catch (Throwable t) {
            throw new UndeclaredThrowableException(t);
        }
    }

    public final void buyTicket(final Ticket ticket) {
        try {
            super.h.invoke(this, $Proxy0.m3, new Object[] { ticket });
        }
        catch (Error | RuntimeException error) {
            throw;
        }
        catch (Throwable t) {
            throw new UndeclaredThrowableException(t);
        }
    }

    public final int hashCode() {
        try {
            return (int)super.h.invoke(this, $Proxy0.m0, null);
```

```
        }
        catch (Error | RuntimeException error) {
            throw;
        }
        catch (Throwable t) {
            throw new UndeclaredThrowableException(t);
        }
    }

    static {
        try {
            $Proxy0.m1 = Class.forName("java.lang.Object").getMethod("equals", Class.forName("
ava.lang.Object"));
            $Proxy0.m2 = Class.forName("java.lang.Object").getMethod("toString", (Class<?>[])n
w Class[0]);
            $Proxy0.m3 = Class.forName("com.wxueyuan.DesignPettern.StaticProxy.Operation").g
tMethod("buyTicket", Class.forName("com.wxueyuan.DesignPettern.StaticProxy.Ticket"));
            $Proxy0.m0 = Class.forName("java.lang.Object").getMethod("hashCode", (Class<?>[])
ew Class[0]);
        }
        catch (NoSuchMethodException ex) {
            throw new NoSuchMethodError(ex.getMessage());
        }
        catch (ClassNotFoundException ex2) {
            throw new NoClassDefFoundError(ex2.getMessage());
        }
    }
}
```

还记得我们之前的return cons.newInstance(new Object[]{h});这句代码么，因为生成的Proxy类的造函数的参数就是就是InvocationHandler，因此我们将newProxyInstance()中的参数h传递过来，来调用它的invoke()方法。

分析了这么多的源码，我们来总结一下Java动态代理的流程吧：

1. Proxy.newProxyInstance()方法返回一个代理类的实例，需要传入InvocationHandler的实例h

2. 当新的代理实例调用指定方法时，本质上是InvocationHandler实例调用invoke方法，并传入指定method类型的参数。

根据我们生成$Proxy0代理类，我们能够总结出：

1. 所有生成的代理类都继承了Proxy类，实现了需要代理的接口。正是由于java不能多继承，所以JD的动态代理不支持对实现类的代理，只支持接口的代理。

2. 提供了一个使用InvocationHandler作为参数的构造方法。这个参数是由Proxy.newProxyInstance(方法的参数传入的。当代理类实例调用某个方法时，本质上是InvocationHandler实例以该方法的metod类型作为参数调用invoke方法。

Java的动态代理其实有很多应用场景，比如Spring的AOP或者是最近很火的RPC框架，里面都涉及到动态代理的知识，因此从原理上分析一下动态代理的源码还是很有帮助的，那么这次的源码分析就到里了，我们下次再见～。