动态代理详解

动态代理：

指的是在Java运行时生成代理类。在运行时，你能获取到产生的代理类的.class文件。如果想看代理类的具体内容，则需要通过反编译将.class文件转换成.java文件。

动态代理解决的问题：

不改变目标方法的代码，实现对目标方法的增强。

> 具有这种增强的效果是因为实际调用方法的是代理类对象，而不是目标类对象，调用的也不是目标类中的方法，而是代理类中的方法。如果用的是JDK动态代理，代理类实现了目标类实现的所有接口，所以目标类实现的所有接口方法，代理类都实现了。代理类中实现的每一个接口方法的内容为：增强 + 这个接口方法在目标类中实现的方法的内容。

动态代理的实现方式：

使用JDK代理：

这种实现方式是基于接口实现的。用这种方法生成代理类的时候，需要先传入目标类实现的所有接口，生成的代理类会实现所有接口。代理类中的方法都是在接口中定义的方法，不存在没有在接口中定义的方法。这就意味着目标类中能被增强的方法只有在接口中定义的方法，目标类中没有在接口中定义的方法是不能被增强的。代理类中的每个方法的内容为：增强 + 与这个方法对应的接口方法在目标类中实现的方法的内容。（代理类中的每个方法的内容是：InvocationHandler对象调用invoke方法。invoke方法里面包含了①增强方法②目标类中对应的方法。）

示例：

```
final class JdkDynamicAopProxy implements AopProxy, InvocationHandler,
Serializable {

    @Override
    public Object getProxy(@Nullable ClassLoader classLoader) {
        if (logger.isTraceEnabled()) {
            logger.trace("Creating JDK dynamic proxy: " +
this.advised.getTargetSource());
        }
        return Proxy.newProxyInstance(classLoader, this.proxiedInterfaces,
this);
    }

    //其余代码略

}
```

注：JdkDynamicAopProxy的getProxy()方法会调用Proxy.newProxyInstance()方法。

```
public class Proxy implements java.io.Serializable {

    @SuppressWarnings("serial") // Not statically typed as Serializable
    protected InvocationHandler h;

    public static Object newProxyInstance(ClassLoader loader,  Class<?>[]
interfaces, InvocationHandler h) {
                                        ...
    }

    //其余代码略

}
```

```
public interface InvocationHandler {

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

    //其余代码略
}
```

- 第一个参数：Object proxy。代理对象。设计这个参数只是为了后期的方便，如果想在invoke方法中使用代理对象的话，尽管通过这个参数来使用。
- 第二个参数：Method method。传入的是目标方法对应到接口中的那个方法。也就是说传入的是接口中的方法。
- 第三个参数：Object[] args。目标方法调用时要传的参数。

invoke方法内部调用目标方法的过程：invoke方法里面会获取到目标对象target,然后利用反射让这个target对象调用method参数这个方法。
在JdkDynamicAopProxy类中的invoke方法中有一行语句：AopUtils.invokeJoinpointUsingReflection(target, method, argsToUse);可以看出这个过程。(JdkDynamicAopProxy类中的invoke方法中获取目标对象的语句为：target = targetSource.getTarget();)

注：
调用Proxy.newProxyInstance()方法时，是将目标类实现的接口们传给interfaces参数。
Proxy.newProxyInstance方法根据传入的第二个参数interfaces动态生成一个代理类(继承了Proxy类)，
这个代理类实现了interfaces中的接口。newProxyInstance()返回的对象就是这个代理类生成的代理类的对象。
Proxy.newProxyInstance方法传入的第一个参数loader是用于加载生成的代理类的(传入用目标类对象获取的类加载器)。

一个代理类的.class反编译为.java后，示例如下：

```
package com.sun.proxy;

import com.atguigu.spring.aop.proxy.ArithmeticCalculator;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;

public final class $Proxy0 extends Proxy implements ArithmeticCalculator {
    private static Method m1;
```

```
    private static Method m2;
    private static Method m6;
    private static Method m3;
    private static Method m5;
    private static Method m4;
    private static Method m0;

    public $Proxy0(InvocationHandler arg0) throws  {
        super(arg0);
    }

    public final boolean equals(Object arg0) throws  {
        try {
            return ((Boolean)super.h.invoke(this, m1, new Object[]
{arg0})).booleanValue();
        } catch (RuntimeException | Error arg2) {
            throw arg2;
        } catch (Throwable arg3) {
            throw new UndeclaredThrowableException(arg3);
        }
    }

    public final String toString() throws  {
        try {
            return (String)super.h.invoke(this, m2, (Object[])null);
        } catch (RuntimeException | Error arg1) {
            throw arg1;
        } catch (Throwable arg2) {
            throw new UndeclaredThrowableException(arg2);
        }
    }

    public final int mul(int arg0, int arg1) throws  {
        try {
            return ((Integer)super.h.invoke(this, m6, new Object[]
{Integer.valueOf(arg0), Integer.valueOf(arg1)})).intValue();
        } catch (RuntimeException | Error arg3) {
            throw arg3;
        } catch (Throwable arg4) {
            throw new UndeclaredThrowableException(arg4);
        }
    }

    public final int add(int arg0, int arg1) throws  {
        try {
            return ((Integer)super.h.invoke(this, m3, new Object[]
{Integer.valueOf(arg0), Integer.valueOf(arg1)})).intValue();
        } catch (RuntimeException | Error arg3) {
            throw arg3;
        } catch (Throwable arg4) {
            throw new UndeclaredThrowableException(arg4);
        }
    }

    public final int sub(int arg0, int arg1) throws  {
        try {
            return ((Integer)super.h.invoke(this, m5, new Object[]
{Integer.valueOf(arg0), Integer.valueOf(arg1)})).intValue();
```

```
        } catch (RuntimeException | Error arg3) {
            throw arg3;
        } catch (Throwable arg4) {
            throw new UndeclaredThrowableException(arg4);
        }
    }

    public final int div(int arg0, int arg1) throws  {
        try {
            return ((Integer)super.h.invoke(this, m4, new Object[]
{Integer.valueOf(arg0), Integer.valueOf(arg1)})).intValue();
        } catch (RuntimeException | Error arg3) {
            throw arg3;
        } catch (Throwable arg4) {
            throw new UndeclaredThrowableException(arg4);
        }
    }

    public final int hashCode() throws  {
        try {
            return ((Integer)super.h.invoke(this, m0,
(Object[])null)).intValue();
        } catch (RuntimeException | Error arg1) {
            throw arg1;
        } catch (Throwable arg2) {
            throw new UndeclaredThrowableException(arg2);
        }
    }

    static {
        try {
            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]);
            m6 =
Class.forName("com.atguigu.spring.aop.proxy.ArithmeticCalculator").getMethod("mu
l",
                    new Class[]{Integer.TYPE, Integer.TYPE});
            m3 =
Class.forName("com.atguigu.spring.aop.proxy.ArithmeticCalculator").getMethod("ad
d",
                    new Class[]{Integer.TYPE, Integer.TYPE});
            m5 =
Class.forName("com.atguigu.spring.aop.proxy.ArithmeticCalculator").getMethod("su
b",
                    new Class[]{Integer.TYPE, Integer.TYPE});
            m4 =
Class.forName("com.atguigu.spring.aop.proxy.ArithmeticCalculator").getMethod("di
v",
                    new Class[]{Integer.TYPE, Integer.TYPE});
            m0 = Class.forName("java.lang.Object").getMethod("hashCode", new
Class[0]);
        } catch (NoSuchMethodException arg1) {
            throw new NoSuchMethodError(arg1.getMessage());
        } catch (ClassNotFoundException arg2) {
            throw new NoClassDefFoundError(arg2.getMessage());
        }
```

```
    }
}
```

通过CDLIB代理：

基于继承