1. jdk的动态代理是基于接口实现的，所以一定要有接口，它可以实现两个功能 1）为接口生成一个实例对象 2）为实现接口的实例对象加入切面逻辑；

2. 为接口生成实例对象

假设有个MoveAble接口，意为可移动的，现在我想通过jdk的动态代理生成实现这个接口的实例。

2.1 MoveAble接口

```
public interface MoveAble {
 void move();
}
```

2.2 调用处理程序 MoveAbleHandler，jdk生成的代理实例最终会调用这个类的invoke方法，相当于写在这个处理程序实现了原本应该由MoveAble实现的逻辑。

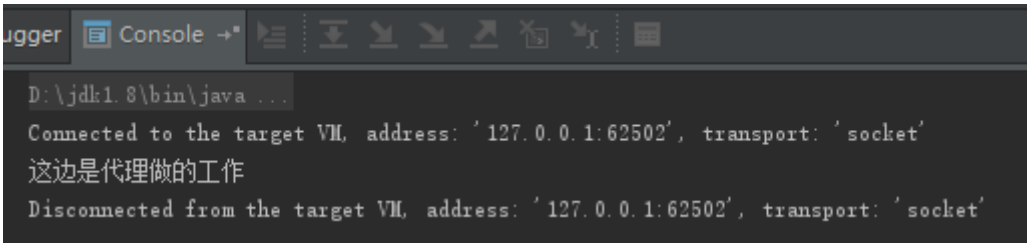
```
public class MoveAbleHandler implements InvocationHandler {

 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
  System.out.println("这边是代理做的工作");
  return null;
  }
}
```

2.3 测试类

```
public class Test {
 public static void main(String[] args) throws Exception {
  // 第一个参数是类加载器，用来加载生成的代理类
  // 第二个参数是代理类实现的接口
  // 第三个参数是调用处理程序
  MoveAble moveAble = (MoveAble) Proxy.newProxyInstance(MoveAble.class.getClassLoader(), new Class[]{MoveAble.class}, new MoveAbleHandler());
  moveAble.move();
  }
}
```

2.4 结果如下



3. 动态代理原理分析

Proxy.newProxyInstance()方法最终会通过native方法生成代理对象

```
private static native Class<?> defineClass0(ClassLoader loader, String name,
 byte[] b, int off, int len);
```

通过jdk的sum.misc.ProxyGenerator类可以查看生成的字节码文件，如下配置

```
import java.io.File;
import java.io.FileOutputStream;
import java.lang.reflect.Proxy;

/**
 * Created by rongyaowen
 * on 2019/5/23.
 */
public class Test {
  public static void main(String[] args) throws Exception {

  // 第一个参数是类加载器，用来加载生成的代理类
  // 第二个参数是代理类实现的接口
  // 第三个参数是调用处理程序
  MoveAble moveAble = (MoveAble) Proxy.newProxyInstance(MoveAble.class.getClassLoader(), new Class[]{MoveAble.class}, new MoveableHandler());
  moveAble.move();
  byte[] bts = ProxyGenerator.generateProxyClass("$MoveAble", Tank.class.getInterfaces());
  FileOutputStream fos = new FileOutputStream(new File("E:/$MoveAble.class"));
  fos.write(bts);
  fos.flush();
  fos.close();
  }
 }
```

将生成的字节码拉入idea中可以看到如下内容

```
import com.honor.proxy.MoveAble;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;

public final class $MoveAble extends Proxy implements MoveAble {
 private static Method m1;
 private static Method m3;
```

```
10  private static Method m2;
11  private static Method m0;
12
13  public $MoveAble(InvocationHandler var1) throws {
14  super(var1);
15  }
16
17  public final boolean equals(Object var1) throws {
18  try {
19  return ((Boolean)super.h.invoke(this, m1, new Object[]{var1})).booleanV
alue();
20  } catch (RuntimeException | Error var3) {
21  throw var3;
22  } catch (Throwable var4) {
23  throw new UndeclaredThrowableException(var4);
24  }
25  }
26
27  public final void move() throws {
28  try {
29  super.h.invoke(this, m3, (Object[])null);
30  } catch (RuntimeException | Error var2) {
31  throw var2;
32  } catch (Throwable var3) {
33  throw new UndeclaredThrowableException(var3);
34  }
35  }
36
37  public final String toString() throws {
38  try {
39  return (String)super.h.invoke(this, m2, (Object[])null);
40  } catch (RuntimeException | Error var2) {
41  throw var2;
42  } catch (Throwable var3) {
43  throw new UndeclaredThrowableException(var3);
44  }
45  }
46
47  public final int hashCode() throws {
48  try {
49  return ((Integer)super.h.invoke(this, m0, (Object[])null)).intValue();
```

```
50  } catch (RuntimeException | Error var2) {
51  throw var2;
52  } catch (Throwable var3) {
53  throw new UndeclaredThrowableException(var3);
54  }
55  }
56
57  static {
58  try {
59  m1 = Class.forName("java.lang.Object").getMethod("equals", new Class[]
{Class.forName("java.lang.Object")});
60  m3 = Class.forName("com.honor.proxy.MoveAble").getMethod("move", new Cl
ass[0]);
61  m2 = Class.forName("java.lang.Object").getMethod("toString", new
Class[0]);
62  m0 = Class.forName("java.lang.Object").getMethod("hashCode", new
Class[0]);
63  } catch (NoSuchMethodException var2) {
64  throw new NoSuchMethodError(var2.getMessage());
65  } catch (ClassNotFoundException var3) {
66  throw new NoClassDefFoundError(var3.getMessage());
67  }
68  }
69 }
```

注：重写了equals、hashCode和toString方法。重点看move方法，move方法通过反射调用了父类的invoke方法

```
1 return ((Boolean)super.h.invoke(this, m1, new Object[]{var1})).booleanVal
ue();
```

super.h是什么呢？跟进去一看是

```
1 protected InvocationHandler h;
```

h是什么时候赋值的呢，当生成的代理实例初始化时，会将InvocationHandler对象赋值给proxy

```
1 public $MoveAble(InvocationHandler var1) throws {
2  super(var1);
3 }
```

那么初始化实例的时候传入的是什么值呢

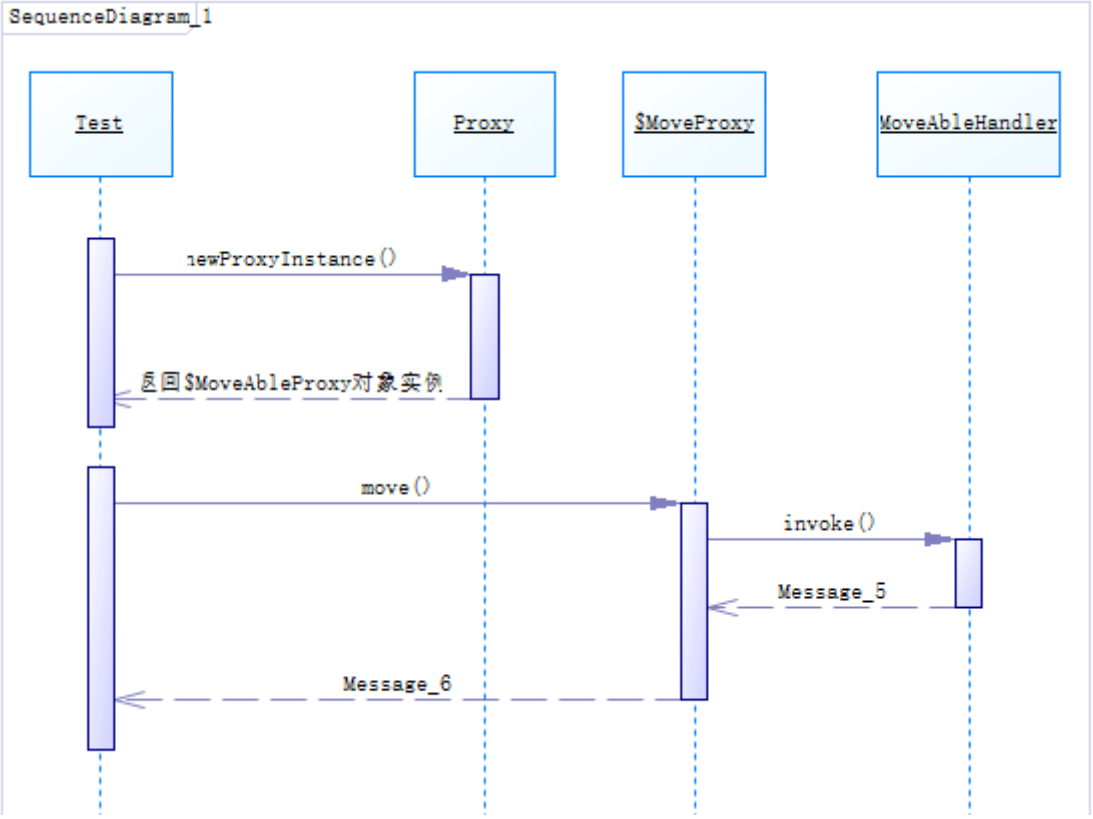
```
1 return cons.newInstance(new Object[]{h});
```

这个h就是newProxyInstance传入得第三个参数，是调用处理程序

```
1 public static Object newProxyInstance(ClassLoader loader,
2  Class<?>[] interfaces,
```

```
InvocationHandler h)
```

总结：通过newProxyInstance方法会生成一个实现了接口（第二个参数）的实例对象，这个对象实现了接口的方法，调用的是调用处理程序（第三个参数）的invoke方法。



4. 动态代理还有另一个功能就是为实现了接口的实例增加切面逻辑

4.1 上述Move接口不变

4.2 增加接口实现类 Tank

```
public class Tank implements MoveAble {
 public void move() {
 System.out.println("坦克移动。。。");
 }
}
```

4.3 修改代理处理程序 TankHandler

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

/**
 * Created by rongyaowen
 * on 2019/5/23.
 */
public class TankHandler implements InvocationHandler {
 // 传入实现了接口的实例，如这边的Tank实例
  private MoveAble moveAble;
```

```java

 public TankHandler(MoveAble moveAble) {
 this.moveAble = moveAble;
 }

 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
 System.out.println("代理开始");
 Object object = method.invoke(moveAble, args);
 System.out.println("代理结束");
 return object;
 }
}
```

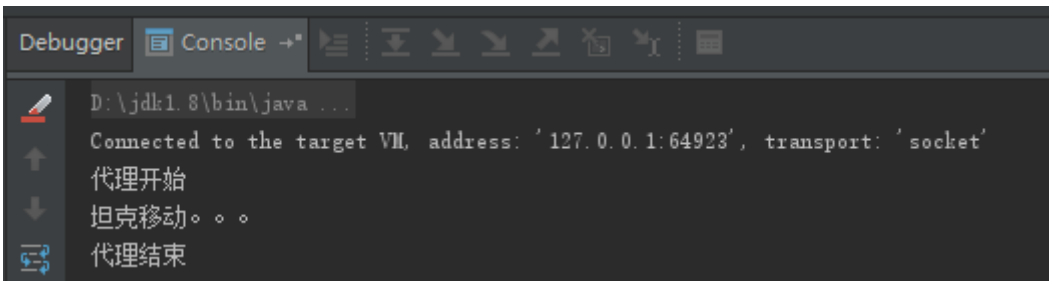
4.4 测试

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

/**
 * Created by rongyaowen
 * on 2019/5/23.
 */
public class Test {
 public static void main(String[] args) throws Exception {
 Tank tank = new Tank();
 InvocationHandler invocationHandler = new TankHandler(tank);
 MoveAble moveAble = (MoveAble) Proxy.newProxyInstance(MoveAble.class.getClassLoader(), new Class[]{MoveAble.class},
 invocationHandler);
 moveAble.move();

 }
}
```

4.5 结果



4.6 总结

这边在代理执行程序中还执行了代理类的方法

```
Object object = method.invoke(moveAble, args);
```

4.7 调用过程就在上面多加一个环节