02 Java基础-设计模式

命令模式

```
// 客户端，请求者，命令接口，命令实现，接受者，
public class Client {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command commandOne = new ConcreteCommandOne(receiver);
        Command commandTwo = new ConcreteCommandTwo(receiver);
        Invoker invoker = new Invoker(commandOne, commandTwo);
        invoker.actionOne();
        invoker.actionTwo();
    }
}

public class Invoker {
    private Command commandOne;
    private Command commandTwo;

    public Invoker(Command commandOne, Command commandTwo) {
        this.commandOne = commandOne;
        this.commandTwo = commandTwo;

    }

    public void actionOne() {
        commandOne.execute();

    }
```

```
    public void actionTwo() {
        commandTwo.execute();

    }
}

public interface Command {
    void execute();
}

public class ConcreteCommandOne implements Command {
    private Receiver receiver

    public ConcreteCommandOne(Receiver receiver) {
        this.receiver = receiver;

    }

    public void execute() {
        receiver.actionOne();

    }
}

public class ConcreteCommandTwo implements Command {
    private Receiver receiver

    public ConcreteCommandTwo(Receiver receiver) {
        this.receiver = receiver;

    }

    public void execute() {
        receiver.actionTwo();

    }
}

public class Receiver {
    public Receiver() {
        //
```

```
    }

    public void actionOne() {
        System.out.println("ActionOne has been taken.");

    }

    public void actionTwo() {
        System.out.println("ActionTwo has been taken.");

    }
}
```

为什么使用命令模式

```
public class Client {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        receiver.actionOne();
        receiver.actionTwo();
    }
}

public class Receiver {
    public Receiver() {
        //
    }

    public void actionOne() {
        System.out.println("ActionOne has been taken.");
    }

    public void actionTwo() {
        System.out.println("ActionTwo has been taken.");
    }
}
```

(1)我们须要Client和Receiver同时开发,而且在开发过程中分别须要不停重购，改名
(2)如果我们要求Redo ,Undo等功能
(3)我们须要命令不按照调用执行，而是按照执行时的情况排序，执行
(4)在上边的情况下，我们的接受者有很多，不止一个

```
public class Invoker {
    private List cmdList = new ArrayList();

    public Invoker() {
    }

    public add(Command command) {
        cmdList.add(command);
    }

    public remove(Command command) {
        cmdList.remove(command);
    }

    public void action() {
        Command cmd;
        while ((cmd = getCmd()) != null) {
            log("begin" + cmd.getName());
            cmd.execute();
            log("end" + cmd.getName());
        }
    }

    public Command getCmd() {
        //按照自定义优先级，排序取出cmd
    }
}

public class Client {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command commandOne = new ConcreteCommandOne(receiver);
        Command commandTwo = new ConcreteCommandTwo(receiver);
        Invoker invoker = new Invoker();
        invoker.add(commandOne);
        invoker.add(commandTwo);
        iinvoker.action();
    }
}
```

redo undo

```
public class ConcreteCommandOne implements Command {
    private Receiver receiver
    private Receiver lastReceiver;

    public ConcreteCommandOne(Receiver receiver) {
        this.receiver = receiver;
    }

    public void execute() {
        record();
        receiver.actionOne();
    }

    public void undo() {
        //恢复状态
    }

    public void redo() {
        lastReceiver.actionOne();
        //
    }

    public record() {
        //记录状态
    }
}
```

代理模式

静态代理

代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。
代理模式一般涉及到的角色有：
抽象角色：声明真实对象和代理对象的共同接口；
代理角色：代理对象角色内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象提供与真实对象相同的接口以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。
真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。

```java
/**
 * 抽象角色
 */
public abstract class Subject {
    public abstract void request();
```

```java
/**
 * 真实的角色
 */
public class RealSubject extends Subject {

    @Override
    public void request() {
        // TODO Auto-generated method stub

    }

}
```

```java
/**
 * 静态代理，对具体真实对象直接引用
 * 代理角色，代理角色需要有对真实角色的引用，
 * 代理做真实角色想做的事情
 */
public class ProxySubject extends Subject {

    private RealSubject realSubject = null;

    /**
     * 除了代理真实角色做该做的事情，代理角色也可以提供附加操作，
     * 如：preRequest()和postRequest()
     */
    @Override
    public void request() {
        preRequest();  //真实角色操作前的附加操作

        if(realSubject == null){
            realSubject =  new RealSubject();
        }
        realSubject.request();

        postRequest();  //真实角色操作后的附加操作
```

```
    }

    /**
     *  真实角色操作前的附加操作
     */
    private void postRequest() {
        // TODO Auto-generated method stub

    }

    /**
     *  真实角色操作后的附加操作
     */
    private void preRequest() {
        // TODO Auto-generated method stub

    }

}
```

```
/**
 *  客户端调用
 */
public class Main {
    public static void main(String[] args) {
        Subject subject = new ProxySubject();
        subject.request();  //代理者代替真实者做事情
    }
}
```

优点：可以做到在不修改目标对象的功能前提下,对目标功能扩展.
缺点：每一个代理类都必须实现一遍委托类（也就是realsubject）的接口，如果接口增加方法，则代理类也必须跟着修改。其次，代理类每一个接口对象对应一个委托对象，如果委托对象非常多，则静态代理类就非常臃肿，难以胜任。

jdk动态代理

动态代理解决静态代理中代理类接口过多的问题，通过反射来实现的，借助Java自带的java.lang.reflect.Proxy,通过固定的规则生成。
步骤如下：

1. 编写一个委托类的接口，即静态代理的（Subject接口）
2. 实现一个真正的委托类，即静态代理的（RealSubject类）
3. 创建一个动态代理类，实现InvocationHandler接口，并重写该invoke方法
4. 在测试类中，生成动态代理的对象。

第一二步骤，和静态代理一样,第三步：

```
public class DynamicProxy implements InvocationHandler {
    private Object object;
    public DynamicProxy(Object object) {
        this.object = object;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws T
hrowable {
        Object result = method.invoke(object, args);
        return result;
    }
}
```

创建动态代理的对象

```
Subject realSubject = new RealSubject();
DynamicProxy proxy = new DynamicProxy(realSubject);
ClassLoader classLoader = realSubject.getClass().getClassLoader();
Subject subject = (Subject) Proxy.newProxyInstance(classLoader, new  Class[]{
Subject.class}, proxy);
subject.visit();
```

上述代码的关键是Proxy.newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler handler)方法，该方法会根据指定的参数动态创建代理对象。三个参数的意义如下：

1. loader，指定代理对象的类加载器；
2. interfaces，代理对象需要实现的接口，可以同时指定多个接口；
3. handler，方法调用的实际处理者，代理对象的方法调用都会转发到这里（*注意1）。

原理：

```
public class Proxy implements java.io.Serializable {
    public static Object newProxyInstance(ClassLoader loader,
                                          Class<?>[] interfaces,
                                          InvocationHandler h)
        throws IllegalArgumentException
    {
        Objects.requireNonNull(h);

        // 准备一份所有被实现的业务接口
        final Class<?>[] intfs = interfaces.clone();
        final SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            checkProxyAccess(Reflection.getCallerClass(), loader, intfs);
        }

        /*
         * 代理类生成的核心代码
         */
        Class<?> cl = getProxyClass0(loader, intfs);
```

classLoader 的作用是将字节码文件加载进虚拟机并生成相应的 class
interfaces 就是被实现的那些接口
h 就是 InvocationHandler

```
public class Proxy implements java.io.Serializable {
/**
     * a cache of proxy classes
     */
    private static final WeakCache<ClassLoader, Class<?>[], Class<?>>
        proxyClassCache = new WeakCache<>(new KeyFactory(), new ProxyClassFac
tory());

    private static Class<?> getProxyClass0(ClassLoader loader,Class<?>... int
erfaces) {

         //代理接口数限制
        if (interfaces.length > 65535) {
            throw new IllegalArgumentException("interface limit exceeded");
        }

        //如果由给定的加载器实现给定的接口定义的代理类存在，返回缓存;否则，它将通过ProxyC
lassFactory创建代理类
```

```
        return proxyClassCache.get(loader, interfaces);
    }
```

```
//通过ClassLoader和接口列表，生成和定义一个proxy class
    private static final class ProxyClassFactory
        implements BiFunction<ClassLoader, Class<?>[], Class<?>>
    {
        // 所有代理类的命名前缀
        private static final String proxyClassNamePrefix = "$Proxy";

        // 一个唯一的number作为proxy class的名称标识
        private static final AtomicLong nextUniqueNumber = new AtomicLong();
        //proxy class生成方法
        @Override
        public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {

            Map<Class<?>, Boolean> interfaceSet = new IdentityHashMap<>(inter
faces.length);
            for (Class<?> intf : interfaces) {
                 //确认类加载器解析了这个名字的接口到相同的Class对象。
                Class<?> interfaceClass = null;
                try {
                    interfaceClass = Class.forName(intf.getName(), false, loa
der);
                } catch (ClassNotFoundException e) {
                }
                if (interfaceClass != intf) {
                    throw new IllegalArgumentException(
                        intf + " is not visible from class loader");
                }
                //确认代理的Class是接口，从这可看出JDK动态代理的劣势
                if (!interfaceClass.isInterface()) {
                    throw new IllegalArgumentException(
                        interfaceClass.getName() + " is not an interface");
                }
                //接口重复校验
                if (interfaceSet.put(interfaceClass, Boolean.TRUE) != null) {
                    throw new IllegalArgumentException(
                        "repeated interface: " + interfaceClass.getName());
                }
            }

            String proxyPkg = null;     // 定义proxy class 所在的包
```

```
            int accessFlags = Modifier.PUBLIC | Modifier.FINAL;

            //记录所有non-public的 proxy interfaces都在同一个package中
            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non-public的interfaces，默认包为com.sun.proxy
                proxyPkg = ReflectUtil.PROXY_PACKAGE + ".";
            }

            /*
             * Choose a name for the proxy class to generate.
             */
            long num = nextUniqueNumber.getAndIncrement();
            //最终大概名字为：com.sun.proxy.$Proxy1
            String proxyName = proxyPkg + proxyClassNamePrefix + num;

            /*
             * 生成特殊的proxy class.
             */
            byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
                proxyName, interfaces, accessFlags);
            try {
                return defineClass0(loader, proxyName,
                                    proxyClassFile, 0, proxyClassFile.length)
;
            } catch (ClassFormatError e) {
                /*
                 * A ClassFormatError here means that (barring bugs in the
```

```
                 * proxy class generation code) there was some other
                 * invalid aspect of the arguments supplied to the proxy
                 * class creation (such as virtual machine limitations
                 * exceeded).
                 */
                throw new IllegalArgumentException(e.toString());
            }
        }
    }
```

```
public static byte[] generateProxyClass(String arg, Class<?>[] arg0, int arg1
) {
      ProxyGenerator arg2 = new ProxyGenerator(arg, arg0, arg1);
      byte[] arg3 = arg2.generateClassFile();
      if(saveGeneratedFiles) {
         AccessController.doPrivileged(new 1(arg, arg3));
      }

      return arg3;
   }
   //私有构造器
   private ProxyGenerator(String arg0, Class<?>[] arg1, int arg2) {
        this.className = arg0;
        this.interfaces = arg1;
        this.accessFlags = arg2;
    }
```

```
private byte[] generateClassFile() {
          //添加从 Object基类中继承的方法
        this.addProxyMethod(hashCodeMethod, Object.class);
        this.addProxyMethod(equalsMethod, Object.class);
        this.addProxyMethod(toStringMethod, Object.class);
        //添加接口中的方法实现
        Class[] arg0 = this.interfaces;
        int arg1 = arg0.length;

        int arg2;
        Class arg3;
        for (arg2 = 0; arg2 < arg1; ++arg2) {
            arg3 = arg0[arg2];
            Method[] arg4 = arg3.getMethods();
            int arg5 = arg4.length;
```

```
            for (int arg6 = 0; arg6 < arg5; ++arg6) {
                Method arg7 = arg4[arg6];
                this.addProxyMethod(arg7, arg3);
            }
        }

        Iterator arg10 = this.proxyMethods.values().iterator();

        List arg11;
        while (arg10.hasNext()) {
            arg11 = (List) arg10.next();
            checkReturnTypes(arg11);
        }

        Iterator arg14;
        try {
                //构造方法
            this.methods.add(this.generateConstructor());
            arg10 = this.proxyMethods.values().iterator();

            while (arg10.hasNext()) {
                arg11 = (List) arg10.next();
                arg14 = arg11.iterator();

                while (arg14.hasNext()) {
                    ProxyMethod arg15 = (ProxyMethod) arg14.next();
                    this.fields.add(new FieldInfo(this, arg15.methodFieldName
, "Ljava/lang/reflect/Method;", 10));
                    this.methods.add(ProxyMethod.access$100(arg15));
                }
            }

            this.methods.add(this.generateStaticInitializer());
        } catch (IOException arg9) {
            throw new InternalError("unexpected I/O Exception", arg9);
        }

        if (this.methods.size() > '□') {
            throw new IllegalArgumentException("method limit exceeded");
        } else if (this.fields.size() > '□') {
            throw new IllegalArgumentException("field limit exceeded");
        } else {
```

```
            this.cp.getClass(dotToSlash(this.className));
            this.cp.getClass("java/lang/reflect/Proxy");
            arg0 = this.interfaces;
            arg1 = arg0.length;

            for (arg2 = 0; arg2 < arg1; ++arg2) {
                arg3 = arg0[arg2];
                this.cp.getClass(dotToSlash(arg3.getName()));
            }

            this.cp.setReadOnly();
            //生成文件
            ByteArrayOutputStream arg12 = new ByteArrayOutputStream();
            DataOutputStream arg13 = new DataOutputStream(arg12);

            try {
                arg13.writeInt(-889275714);
                arg13.writeShort(0);
                arg13.writeShort(49);
                this.cp.write(arg13);
                arg13.writeShort(this.accessFlags);
                arg13.writeShort(this.cp.getClass(dotToSlash(this.className))
);
                arg13.writeShort(this.cp.getClass("java/lang/reflect/Proxy"))
;
                arg13.writeShort(this.interfaces.length);
                Class[] arg16 = this.interfaces;
                int arg17 = arg16.length;

                for (int arg18 = 0; arg18 < arg17; ++arg18) {
                    Class arg21 = arg16[arg18];
                    arg13.writeShort(this.cp.getClass(dotToSlash(arg21.getNam
e())));
                }

                arg13.writeShort(this.fields.size());
                arg14 = this.fields.iterator();

                while (arg14.hasNext()) {
                    FieldInfo arg19 = (FieldInfo) arg14.next();
                    arg19.write(arg13);
                }
```

```
                arg13.writeShort(this.methods.size());
                arg14 = this.methods.iterator();

                while (arg14.hasNext()) {
                    MethodInfo arg20 = (MethodInfo) arg14.next();
                    arg20.write(arg13);
                }

                arg13.writeShort(0);
                //返回字节流
                return arg12.toByteArray();
            } catch (IOException arg8) {
                throw new InternalError("unexpected I/O Exception", arg8);
            }
        }
    }
```

生成的proxy.class

```
import com.proxy.api.PeopleService;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;
/**
 * 生成的类继承了Proxy,实现了要代理的接口PeopleService
 *
 */
public final class $Proxy11
  extends Proxy
  implements PeopleService
{
  private static Method m1;
  private static Method m3;
  private static Method m2;
  private static Method m4;
  private static Method m0;

  public $proxy11(InvocationHandler paramInvocationHandler)
  {
    //调用基类Proxy的构造器
    super(paramInvocationHandler);
  }
```

```
  public final boolean equals(Object paramObject)
  {
    try
    {
      return ((Boolean)this.h.invoke(this, m1, new Object[] { paramObject }))
.booleanValue();
    }
    catch (Error|RuntimeException localError)
    {
      throw localError;
    }
    catch (Throwable localThrowable)
    {
      throw new UndeclaredThrowableException(localThrowable);
    }
  }

  public final void mainJiu()
  {
    try
    {
      this.h.invoke(this, m3, null);
      return;
    }
    catch (Error|RuntimeException localError)
    {
      throw localError;
    }
    catch (Throwable localThrowable)
    {
      throw new UndeclaredThrowableException(localThrowable);
    }
  }

  public final String toString()
  {
    try
    {
      return (String)this.h.invoke(this, m2, null);
    }
    catch (Error|RuntimeException localError)
    {
```

```
      throw localError;
    }
    catch (Throwable localThrowable)
    {
      throw new UndeclaredThrowableException(localThrowable);
    }
  }

  public final void printName(String paramString)
  {
    try
    {
      this.h.invoke(this, m4, new Object[] { paramString });
      return;
    }
    catch (Error|RuntimeException localError)
    {
      throw localError;
    }
    catch (Throwable localThrowable)
    {
      throw new UndeclaredThrowableException(localThrowable);
    }
  }

  public final int hashCode()
  {
    try
    {
      return ((Integer)this.h.invoke(this, m0, null)).intValue();
    }
    catch (Error|RuntimeException localError)
    {
      throw localError;
    }
    catch (Throwable localThrowable)
    {
      throw new UndeclaredThrowableException(localThrowable);
    }
  }

  static
  {
```

```
    try
    {
    //利用反射生成5个方法，包括Object中的equals、toString、hashCode以及PeopleServic
e中的mainJiu和printName
      m1 = Class.forName("java.lang.Object").getMethod("equals", new Class[]
{ Class.forName("java.lang.Object") });
      m3 = Class.forName("com.proxy.api.PeopleService").getMethod("mainJiu",
new Class[0]);
      m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[
0]);
      m4 = Class.forName("com.proxy.api.PeopleService").getMethod("printName"
, new Class[] { Class.forName("java.lang.String") });
      m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[
0]);
      return;
    }
    catch (NoSuchMethodException localNoSuchMethodException)
    {
      throw new NoSuchMethodError(localNoSuchMethodException.getMessage());
    }
    catch (ClassNotFoundException localClassNotFoundException)
    {
      throw new NoClassDefFoundError(localClassNotFoundException.getMessage()
);
    }
  }
}
```

现在再看生成的$Proxy.class反编译的java代码中，调用mainJiu时会this.h.invoke(this, m3, null);调用Proxy类中InvocationHandler的invoke方法

总结
newProxyInstance()通过反射生成含有接口方法的proxy class（继承了Proxy类，实现了需要代理的接口）
因此对该对象的所有方法调用都会转发InvocationHandler.invoke()方法

cglib动态代理

假设我们有一个没有实现任何接口的类HelloConcrete

```
public class HelloConcrete {
```

```
    public String sayHello(String str) {
        return "HelloConcrete: " + str;
    }
}
```

```
// CGLIB动态代理
// 1. 首先实现一个MethodInterceptor，方法调用会被转发到该类的intercept()方法。
class MyMethodInterceptor implements MethodInterceptor{
  ...
    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodP
roxy proxy) throws Throwable {
        logger.info("You said: " + Arrays.toString(args));
        return proxy.invokeSuper(obj, args);
    }
}
// 2. 然后在需要使用HelloConcrete的时候，通过CGLIB动态代理获取代理对象。
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(HelloConcrete.class);
enhancer.setCallback(new MyMethodInterceptor());

HelloConcrete hello = (HelloConcrete)enhancer.create();
System.out.println(hello.sayHello("I love you!"));
```

通过CGLIB的Enhancer来指定要代理的目标对象、实际处理代理逻辑的对象，最终通过调用create()方法得到代理对象，对这个对象所有非final方法的调用都会转发给MethodInterceptor.intercept()方法，在intercept()方法里我们可以加入任何逻辑，比如修改方法参数，加入日志功能、安全检查功能等；通过调用MethodProxy.invokeSuper()方法，我们将调用转发给原始对象，具体到本例，就是HelloConcrete的具体方法。
对于从Object中继承的方法，CGLIB代理也会进行代理，如hashCode()、equals()、toString()等，但是getClass()、wait()等方法不会，因为它是final方法，CGLIB无法代理。

原理：
CGLIB是一个强大的高性能的代码生成包，底层是通过使用一个小而快的字节码处理框架ASM，它可以在运行期扩展Java类与实现Java接口
Enhancer是CGLIB的字节码增强器，可以很方便的对类进行拓展
创建代理对象的几个步骤:
1、生成代理类的二进制字节码文件
2、加载二进制字节码，生成Class对象(例如使用Class.forName()方法)
3、通过反射机制获得实例构造，并创建代理类对象

关系

总结

1. jdk动态代理：利用拦截器(拦截器必须实现InvocationHanlder)加上反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。只能对实现了接口的类生成代理只能对实现了接口的类生成代理
2. cglib：利用ASM开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。主要是对指定的类生成一个子类，覆盖其中的方法，并覆盖其中方法实现增强，但是因为采用的是继承，对于final类或方法，是无法继承的。
3. 选择
    i. 如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP。
    ii. 如果目标对象实现了接口，可以强制使用CGLIB实现AOP。
    iii. 如果目标对象没有实现了接口，必须采用CGLIB库，Spring会自动在JDK动态代理和CGLIB之间转换。