名词解释

静态代理：编译期就已确定代理对象。即编码出代理类。

动态代理：运行时动态生成代理对象。可对被代理类做出统一的处理，如日志打印，统计调用次数等。

JDK动态代理：即JDK中自带的动态代理生成方式。JDK动态代理的实现依赖于被代理类必须实现自接口。

cglib动态代理：cglib工具包实现的动态代理生成方式，通过字节码来实现动态代理，不需要被代理类必须实现接口。

动态代理核心源码实现

```
public static Object newProxyInstance(ClassLoader loader,
                                       Class<?>[] interfaces,
                                       InvocationHandler h)
    throws IllegalArgumentException
  {
    //...
                        //克隆接口的字节码
    final Class<?>[] intfs = interfaces.clone();
    //...
                        //从缓存中获取或生成指定的代理类
    Class<?> cl = getProxyClass0(loader, intfs);
    try {
      //获取构造函数
      final Constructor<?> cons = cl.getConstructor(constructorParams);
      final InvocationHandler ih = h;
      //根据Proxy的有参构造函数构造出代理类
      return cons.newInstance(new Object[]{h});
    }
    //...
  }

private static Class<?> getProxyClass0(ClassLoader loader,
                                       Class<?>... interfaces) {
                    //...接口的数量不能超过65535
    if (interfaces.length > 65535) {
      throw new IllegalArgumentException("interface limit exceeded");
    }

                    // WeakCache<ClassLoader, Class<?>[], Class<?>> proxyClassCache=new
WeakCache<>(new KeyFactory(), new ProxyClassFactory());
                    //如果指定的类加载器已经生成代理实现类，那么直接从缓存获取副本，否则生成新的代理实现类。
    return proxyClassCache.get(loader, interfaces);
```

```
    }

//proxyClassCache的get方法
public V get(K key, P parameter) {

                //...key为classloader，parameter为接口的Class数组
                //删除过时的entry
        expungeStaleEntries();
                        //构造CacheKey key为null时，cacheKey为object对象，否则为虚引用对象
        Object cacheKey = CacheKey.valueOf(key, refQueue);

        //根据cacheKey加载二级缓存
        ConcurrentMap<Object, Supplier<V>> valuesMap = map.get(cacheKey);
        if (valuesMap == null) {
          //如果不存在，构造二级缓存
            ConcurrentMap<Object, Supplier<V>> oldValuesMap
                = map.putIfAbsent(cacheKey,
                                  valuesMap = new ConcurrentHashMap<>());
            if (oldValuesMap != null) {
              //如果出于并发情况，返回了缓存map，将原缓存map赋值给valuesMap
                valuesMap = oldValuesMap;
            }
        }

        //构造二级缓存key，subKey
        Object subKey = Objects.requireNonNull(subKeyFactory.apply(key, parameter));
                //获取生成代理类的代理类工厂
        Supplier<V> supplier = valuesMap.get(subKey);
        Factory factory = null;

        while (true) {
          //循环获取生成代理类的代理类工厂
            if (supplier != null) {
                // 如果代理类工厂不为空，通过get方法获取代理类。该supplier为WeakCache的内部类
Factory
                V value = supplier.get();
                if (value != null) {
                    return value;
                }
            }

            if (factory == null) {
              //代理工厂类为null，创建代理工厂类
                factory = new Factory(key, parameter, subKey, valuesMap);
            }

            if (supplier == null) {
                supplier = valuesMap.putIfAbsent(subKey, factory);
                if (supplier == null) {
                    // successfully installed Factory
                    supplier = factory;
                }
```

```
                // else retry with winning supplier
            } else {
                if (valuesMap.replace(subKey, supplier, factory)) {
                    // successfully replaced
                    // cleared CacheEntry / unsuccessful Factory
                    // with our Factory
                    supplier = factory;
                } else {
                    // retry with current supplier
                    supplier = valuesMap.get(subKey);
                }
            }
        }
    }

//Factory的get方法
public synchronized V get() { // serialize access
            // re-check
            Supplier<V> supplier = valuesMap.get(subKey);
            if (supplier != this) {
             //如果在并发等待的时候有变化，返回null，继续执行外层的循环。
                return null;
            }
            //创建新的代理类
            V value = null;
            try {
             //通过ProxyClassFactory的apply方法生成代理类
                value = Objects.requireNonNull(valueFactory.apply(key, parameter));
            } finally {
                if (value == null) { // remove us on failure
                    valuesMap.remove(subKey, this);
                }
            }
            //用CacheValue包装value值(代理类)
            CacheValue<V> cacheValue = new CacheValue<>(value);

            //将cacheValue放入reverseMap
            reverseMap.put(cacheValue, Boolean.TRUE);
            return value;
        }

//ProxyClassFactory类的apply方法
public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {

            Map<Class<?>, Boolean> interfaceSet = new IdentityHashMap<>(interfaces.length);
            //校验class是否正确，校验class是否是interface，校验class是否重复
                                //...
                                        //代理类的包名
            String proxyPkg = null;     // package to define proxy class in
                                //代理类的访问修饰符
            int accessFlags = Modifier.PUBLIC | Modifier.FINAL;
                                        //记录非public修饰的被代理类接口，用来作为代理类的包名，同时
```

校验所有非public修饰的被代理类接口必须处于同一包名下

```
        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public的接口类，包名使用com.sun.proxy package
            proxyPkg = ReflectUtil.PROXY_PACKAGE + ".";
        }

        /*
         * Choose a name for the proxy class to generate.
         */
        long num = nextUniqueNumber.getAndIncrement();
                                //构造代理类名称，使用包名+代理类前缀+自增值作为代理类名称
        String proxyName = proxyPkg + proxyClassNamePrefix + num;

        //生成代理类的字节码文件
        byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
            proxyName, interfaces, accessFlags);
        try {
            //通过native的方法生成代理类
            return defineClass0(loader, proxyName,
                                proxyClassFile, 0, proxyClassFile.length);
        }
                                //...
    }
```

总结

Proxy.newProxyInstance方法获取代理类执行过程：

Proxy.getProxyClass0()方法获取代理类class。

WeakCache.get()方法

CacheKey.valueOf(key, refQueue)获取一级缓存key，cacheKey。

ConcurrentMap.get()方法获取二级缓存ConcurrentMap。

KeyFactory生成二级缓存key，subKey。

ConcurrentMap.get()方法获取二级缓存value，Supplier实现类Factory。

Factory不存在，则通过new Factory生成新的Factory。
通过Factory的get方法获取二级缓存值CacheValue实例。
通过Factory内部缓存ConcurrentMap.get()方法获取Supplier实例。
如果Supplier实例不存在，通过ProxyClassFactory.apply()方法生成代理类class。
使用cacheValue包装代理类class。
Class.getConstructor(InvocationHandler.class)获取有参（InvocationHandler）构造函数。
Constructor.newInstance(InvocationHandler)获取代理类。
代理类的包名：由被代理类实现的接口的限定修饰符确定，如果有非public修饰符，则包名为非public接口所在包路径。如果多个非public修饰符的接口，这些接口必须处于同一包中。如果全为public接口，那么包名为com.sun.proxy。
代理类的全路径类名：包名+代理类名前缀（$Proxy）+自增数字。
Proxy内部采用了多级缓存缓存生成的代理类class，避免重复生成相同的代理类，从而提高性能。
缓存使用的类是WeakCache。

```
//初始化
private static final WeakCache<ClassLoader, Class<?>[], Class<?>>
    proxyClassCache = new WeakCache<>(new KeyFactory(), new ProxyClassFactory());
```

一级缓存的key是CacheKey，CacheKey由classloader和refQueue（引用队列）构成。
一级缓存的value是ConcurrentMap<Object, Supplier> 。
二级缓存的key，subKey，由subKeyFactory（KeyFactory）工厂类根据被代理类实现的接口数量生成。
二级缓存的value是Supplier的实现类，Factory。

代理类class由二级缓存的get（）方法获得，最终生成代理类class的是ProxyClassFactory的apply方法，apply方法生成字节码文件后，通过调用native方法defineClass0最终生成Class。

```
package com.sun.proxy;

import com.xt.design.pattern.proxy.dynamic.jdk.HelloService;
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 HelloService {
    private static Method m1;
    private static Method m3;
    private static Method m2;
    private static Method m0;
```

```
    public $Proxy0(InvocationHandler var1) throws  {
        super(var1);
    }

    public final boolean equals(Object var1) throws  {
        try {
            return (Boolean)super.h.invoke(this, m1, new Object[]{var1});
        } catch (RuntimeException | Error var3) {
            throw var3;
        } catch (Throwable var4) {
            throw new UndeclaredThrowableException(var4);
        }
    }

    public final void sayHello() throws  {
        try {
            super.h.invoke(this, m3, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    public final String toString() throws  {
        try {
            return (String)super.h.invoke(this, m2, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    public final int hashCode() throws  {
        try {
            return (Integer)super.h.invoke(this, m0, (Object[])null);
        } catch (RuntimeException | Error var2) {
            throw var2;
        } catch (Throwable var3) {
            throw new UndeclaredThrowableException(var3);
        }
    }

    static {
        try {
            m1 = Class.forName("java.lang.Object").getMethod("equals",
Class.forName("java.lang.Object"));
            m3 =
Class.forName("com.xt.design.pattern.proxy.dynamic.jdk.HelloService").getMethod("sayHello");
```

```
            m2 = Class.forName("java.lang.Object").getMethod("toString");
            m0 = Class.forName("java.lang.Object").getMethod("hashCode");
        } catch (NoSuchMethodException var2) {
            throw new NoSuchMethodError(var2.getMessage());
        } catch (ClassNotFoundException var3) {
            throw new NoClassDefFoundError(var3.getMessage());
        }
    }
}
```