如其名，这是JDK自身提供的一种代理方式，为啥说是动态呢，其实他就是静态代理演化过来的，静态代理拥有繁琐杂多的代理类显然不符合我们的需求，恰恰很多代理类又干着相似的活，这在我们看来，肯定是有办法合并的嘛，恰恰有项技术就能完成这样的通泛性工作--反射。JDK动态代理正是运用了反射的技术让代理类具备了更大的通用性。

接口和实现的类依旧沿用静态代理的，我们看看动态如何去实现中介。

```
public class DynamicProxyHandler implements InvocationHandler {

	private Object target;

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

	@Override
	public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

		System.out.println("准备");

		Object object = method.invoke(target, args);

		System.out.println("结束");

		return object;
	}

	public static void main(String[] args) {
		UserDao userDao = new UserDaoImpl();

		UserDao proxyUserDao = (UserDao)
Proxy.newProxyInstance(userDao.getClass().getClassLoader(),
				userDao.getClass().getInterfaces(), new
DynamicProxyHandler(userDao));

		proxyUserDao.add();
	}

}
```

我们可以看到这个代理类和UserDao并没有半毛钱的关系，在构建代理对象时才指定了UserDao，是不是就实现了解耦，当然这个不能算是完全的动态代理类， 这只是我们需要去写的处理的地方，真正最终生

成代理类的是Proxy这个类，包括Proxy类和我们处理类实现的接口InvocationHandler都是位于反射包下的，所以要研究JDK动态代理就会涉及到反射技术的运用。为啥最后的代理对象执行了add方法就能调用到我们处理类中的invoke，我们开始深入到源码中去解析。首先我们去看看newProxyInstance这个方法。

```
@CallerSensitive
public static Object newProxyInstance(ClassLoader loader,
                                      Class<?>[] interfaces,
                                      InvocationHandler h)
    throws IllegalArgumentException
{
    Objects.requireNonNull(h);

    final Class<?>[] intfs = interfaces.clone();
    final SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
        checkProxyAccess(Reflection.getCallerClass(), loader, intfs);
    }

    /*
     * Look up or generate the designated proxy class.
     */
    Class<?> cl = getProxyClass0(loader, intfs);

    /*
     * Invoke its constructor with the designated invocation handler.
     */
    try {
        if (sm != null) {
            checkNewProxyPermission(Reflection.getCallerClass(), cl);
        }

        final Constructor<?> cons = cl.getConstructor(constructorParams);
        final InvocationHandler ih = h;
        if (!Modifier.isPublic(cl.getModifiers())) {
            AccessController.doPrivileged(new PrivilegedAction<Void>() {
                public Void run() {
                    cons.setAccessible(true);
                    return null;
                }
            });
        }
        return cons.newInstance(new Object[]{h});
    } catch (IllegalAccessException|InstantiationException e) {
        throw new InternalError(e.toString(), e);
    } catch (InvocationTargetException e) {
```

```
        Throwable t = e.getCause();
        if (t instanceof RuntimeException) {
            throw (RuntimeException) t;
        } else {
            throw new InternalError(t.toString(), t);
        }
    } catch (NoSuchMethodException e) {
        throw new InternalError(e.toString(), e);
    }
}
```

看似一长段代码，下面的catch不用看，中间穿插的校验也不看，其实重要的就是画红框的几处，首先第一处看代码注释也明白了，这个方法就是动态生成代理类的，它两个参数一个是目标类的类加载器，一个是目标类的接口。我们看看这个方法实现了啥。

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

    // If the proxy class defined by the given loader implementing
    // the given interfaces exists, this will simply return the cached copy;
    // otherwise, it will create the proxy class via the ProxyClassFactory
    return proxyClassCache.get(loader, interfaces);
}
```

这个proxyClassCache.get方法有意思啊，看样子这边还用到缓存（cache）相关的东西的样子，我们进去看看，到达另一个类WeakCache（弱缓存？），我们定位到get方法。

```
public V get(K key, P parameter) {
    Objects.requireNonNull(parameter);

    expungeStaleEntries();

    Object cacheKey = CacheKey.valueOf(key, refQueue);

    // lazily install the 2nd level valuesMap for the particular cacheKey
    ConcurrentMap<Object, Supplier<V>> valuesMap = map.get(cacheKey);
    if (valuesMap == null) {
        ConcurrentMap<Object, Supplier<V>> oldValuesMap
            = map.putIfAbsent(cacheKey,
                              valuesMap = new ConcurrentHashMap<>());
        if (oldValuesMap != null) {
            valuesMap = oldValuesMap;
        }
```

```
        }
    }

    // create subKey and retrieve the possible Supplier<V> stored by that
    // subKey from valuesMap
    Object subKey = Objects.requireNonNull(subKeyFactory.apply(key, parameter));
    Supplier<V> supplier = valuesMap.get(subKey);
    Factory factory = null;

    while (true) {
        if (supplier != null) {
            // supplier might be a Factory or a CacheValue<V> instance
            V value = supplier.get();
            if (value != null) {
                return value;
            }
        }
        // else no supplier in cache
        // or a supplier that returned null (could be a cleared CacheValue
        // or a Factory that wasn't successful in installing the CacheValue)

        // lazily construct a Factory
        if (factory == null) {
            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);
            }
        }
```

看样子这就是一个完全的泛型方法，看两个参数名，key和parameter，联系类名弱缓存，估计是用了关于map的技术涉及到缓存相关的内容。那显然这边传进来的key是类加载器，parameter是接口组。

我们通读下代码，发现最后的返回值（也就是我们的动态生成的代理类）是这样获取的。

```
V value = supplier.get();
```

这个supplier又是哪来的呢，往代码上方找。

```
Supplier<V> supplier = valuesMap.get(subKey);
```

原来从一个ConcurrentMap中获取的，其实，我们深入到supplier.get会发现有点意思的。

```
private final class Factory implements Supplier<V> {
```

其实用的是这个WeakCache的内部类Factory，通读上面WeakCache的get方法就会发现这样的代码。

```
if (factory == null) {
        factory = new Factory(key, parameter, subKey, valuesMap);
}
```

我们继续看Factory中的get方法。

```
// create new value
V value = null;
try {
    value = Objects.requireNonNull(valueFactory.apply(key, parameter));
} finally {
    if (value == null) { // remove us on failure
        valuesMap.remove(subKey, this);
    }
}
```

显然最终的value是这样取的。

```
private static final class ProxyClassFactory
    implements BiFunction<ClassLoader, Class<?>[], Class<?>>
{
    // prefix for all proxy class names
    private static final String proxyClassNamePrefix = "$Proxy";

    // next number to use for generation of unique proxy class names
    private static final AtomicLong nextUniqueNumber = new AtomicLong();

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

这个类终于看的亲切了，代理类工厂，显然就是它生成代理类了。我通读了下apply方法，最终定位到这里。

```
 */
byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
    proxyName, interfaces, accessFlags);
```

该处似乎就是生成代理类字节码文件的关键处。

```
public static byte[] generateProxyClass(String paramString, Class<?>[] paramArrayOfClass, int paramInt)
{
  ProxyGenerator localProxyGenerator = new ProxyGenerator(paramString, paramArrayOfClass, paramInt);
  byte[] arrayOfByte = localProxyGenerator.generateClassFile();

  if (saveGeneratedFiles) {
    AccessController.doPrivileged(new PrivilegedAction(paramString, arrayOfByte)
    {
      public Void run() {
        try {
          int i = this.val$name.lastIndexOf('.');
          Path localPath1;
          if (i > 0) {
            Path localPath2 = Paths.get(this.val$name.substring(0, i).replace('.', File.separatorChar), new String[0]);
            Files.createDirectories(localPath2, new FileAttribute[0]);
            localPath1 = localPath2.resolve(this.val$name.substring(i + 1, this.val$name.length()) + ".class");
          } else {
            localPath1 = Paths.get(this.val$name + ".class", new String[0]);
          }
          Files.write(localPath1, this.val$classFile, new OpenOption[0]);
          return null; } catch (IOException localIOException) {
        }
        throw new InternalError("I/O exception saving generated file: " + localIOException);
      }

    });
  }

  return arrayOfByte;
}
```

代码实现是这样，额，总得来说就是生成了代理类了（脑壳疼）。看到这，基本稍微了解了获得代理类的来源了。但这个代理类目前似乎跟我们的处理类并无关联，回到我们Proxy的newProxyInstance方法，下面我画的两个红框正是做了这项处理，将InvactionHandler的对象作为参数让由反射得到的代理类构造器创建真正的代理类实例，最后代理类的每个方法都会执行处理类中的invoke（为啥会这样，暂时我也没搞清楚，头疼，以后再看）。