JDK动态代理确实已经很强大了，但是某些情况下就会很奇怪，为了创建代理类，我必须实现一个接口，虽说面向接口编程是一种主流，但不代表我写任何类都得基于接口去写，这反倒成为一种枷锁了。CGLIB的出示就是解决这一问题的，想要代理但又不需要接口的情况，就能使用CGLIB动态代理去处理。

我们依旧沿用静态代理中的UserDaoImpl，看看CGLIB的代码如何处理。

```
public class CglibProxy implements MethodInterceptor {

	private Enhancer enhancer;

	public Object getProxy(Class clazz) {
		enhancer = new Enhancer();
		enhancer.setSuperclass(clazz);
		enhancer.setCallback(this);
		return enhancer.create();
	}

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

		System.out.println("准备");
		Object result = proxy.invokeSuper(obj, args);
		System.out.println("结束");

		return result;
	}

	public static void main(String[] args) {
		CglibProxy cglibProxy = new CglibProxy();
		UserDaoImpl userDaoImpl =
(UserDaoImpl)cglibProxy.getProxy(UserDaoImpl.class);
		userDaoImpl.add();
	}

}
```

可以看到，我们全程是没有去涉及UserDao这个接口的，这里面我们自己写的一个getProxy方法以及它对Enhance类的操作就是创建代理类对象的关键。研究源码前先说下实现原理吧，CGLIB是通过字节码技术（一直有人鼓吹CGLIB效率比JDK动态代理高的原因）为类创建一个子

类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。

那就看源码吧。（不知道能写成啥样。。。上面的JDK动态代理的还是乱七八糟的）。

我们先看看我们自己操作Enhancer都干了些啥。

```
enhancer.setSuperclass(clazz);
enhancer.setCallback(this);
return enhancer.create();
```

第一个设值意思设置超类（联系CGLIB创建子类的方式），第二个设值意思设置回调（直接把本处理类放入），总体反正就是把目标类和处理类都作为参数设置好了，最后调用了create方法就创建了代理类对象，我们就顺序进create开始看。

```
public Object create()
{
    this.classOnly = false;
    this.argumentTypes = null;
    return createHelper();
}
```

去看createHelper方法。

```
private Object createHelper()
{
  preValidate();
  Object key = KEY_FACTORY.newInstance(this.superclass != null ? this.superclass.getName() : null,
    ReflectUtils.getNames(this.interfaces),
    this.filter == ALL_ZERO ? null : new WeakCacheKey(this.filter), this.callbackTypes, this.useFactory, this.interceptDuringConstruction, this.serialVersionUID);

  this.currentKey = key;
  Object result = super.create(key);
  return result;
}
```

这里首先调用了preValidate方法，应该是个校验方法，因为里面涉及了对callbacks属性的操作，我们看看如何操作。

```
private void preValidate() {
  if (this.callbackTypes == null) {
    this.callbackTypes = CallbackInfo.determineTypes(this.callbacks, false);
    this.validateCallbackTypes = true;
  }
  if (this.filter == null) {
    if (this.callbackTypes.length > 1) {
      throw new IllegalStateException("Multiple callback types possible but no filter specified");
    }
    this.filter = ALL_ZERO;
  }
}
```

红线处讲callbacks作为参数最后得到了一个callbackTypes属性，这个在createHelper方法中是有用的。总得来说，这一步得到callbackTypes是关键处。

回到createHelper，我们看看接下来干了啥，额，一长串代码得到了一个key，我们慢慢看，首先这个KEY_FACTORY是啥呢。

```
private static final EnhancerKey KEY_FACTORY = (EnhancerKey)KeyFactory.create(EnhancerKey.class, KeyFactory.HASH_ASM_TYPE, null)
  ;
```