# 代理

[Java动态代理与CGLIB](http://www.importnew.com/22015.html)

我们常说的代理分为静态代理和动态代理。

* 静态代理：代码中显式指定代理
* 动态代理：类比静态代理，可以发现，代理类不需要实现原接口了，而是实现InvocationHandler。

### 静态代理

因为需要对一些函数进行二次处理，或是某些函数不让外界知道时，可以使用代理模式，通过访问第三方，间接访问原函数的方式，达到以上目的。

#### 弊端

如果要想为多个类进行代理，则需要建立多个代理类，维护难度加大。

仔细想想，为什么静态代理会有这些问题，是因为代理在编译期就已经决定，如果代理发生在运行期，这些问题解决起来就比较简单，所以动态代理的存在就很有必要了。

### 动态代理

当动态生成的代理类调用方法时，会触发 invoke 方法，在 invoke 方法中可以对被代理类的方法进行增强。

// 1. 首先实现一个InvocationHandler，方法调用会被转发到该类的invoke()方法。  
class LogInvocationHandler implements InvocationHandler{  
 ...  
 private Hello hello;  
 public LogInvocationHandler(Hello hello) {  
 this.hello = hello;  
 }  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 if("sayHello".equals(method.getName())) {  
 logger.info("You said: " + Arrays.toString(args));  
 }  
 return method.invoke(hello, args);  
 }  
}  
// 2. 然后在需要使用Hello的时候，通过JDK动态代理获取Hello的代理对象。  
Hello hello = (Hello)Proxy.newProxyInstance(  
 getClass().getClassLoader(), // 1. 类加载器  
 new Class<?>[] {Hello.class}, // 2. 代理需要实现的接口，可以有多个  
 new LogInvocationHandler(new HelloImp()));// 3. 方法调用的实际处理者  
System.out.println(hello.sayHello("I love you!"));

通过动态代理可以很明显的看到它的好处，在使用静态代理时，如果不同接口的某些类想使用代理模式来实现相同的功能，将要实现多个代理类，但在动态代理中，只需要一个代理类就好了。

除了省去了编写代理类的工作量，动态代理实现了可以在原始类和接口还未知的时候，就确定代理类的代理行为，当代理类与原始类脱离直接联系后，就可以很灵活地重用于不同的应用场景中。

* 继承了Proxy类，实现了代理的接口，由于java不能多继承，这里已经继承了Proxy类了，不能再继承其他的类，所以JDK的动态代理不支持对实现类的代理，只支持接口的代理。
* 提供了一个使用InvocationHandler作为参数的构造方法。
* 生成静态代码块来初始化接口中方法的Method对象，以及Object类的equals、hashCode、toString方法

#### 弊端

代理类和委托类需要都实现同一个接口。也就是说只有实现了某个接口的类可以使用Java动态代理机制。但是，事实上使用中并不是遇到的所有类都会给你实现一个接口。因此，对于没有实现接口的类，就不能使用该机制。

### 动态代理与静态代理的区别

* Proxy类的代码被固定下来，不会因为业务的逐渐庞大而庞大；
* 代理对象是在程序运行时产生的，而不是编译期；
* 可以实现AOP编程，这是静态代理无法实现的；
* 解耦，如果用在web业务下，可以实现数据层和业务层的分离。
* 动态代理的优势就是实现无侵入式的代码扩展。
* 静态代理这个模式本身有个大问题，如果类方法数量越来越多的时候，代理类的代码量是十分庞大的。所以引入动态代理来解决此类问题

## [CGLib](http://blog.csdn.net/danchu/article/details/70238002)

cglib 是针对类来实现代理的，他的 **原理是对指定的目标类生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对 final 修饰的类进行代理**。同样的，**final 方法是不能重载的**，所以也不能通过CGLIB代理，遇到这种情况不会抛异常，而是会跳过 final 方法只代理其他方法。

CGLIB 代理主要通过对字节码的操作，为对象引入间接级别，以控制对象的访问。 CGLIB 底层使用了ASM（一个短小精悍的字节码操作框架）来操作字节码生成新的类。

## CGLIB和Java动态代理的区别

* Java 动态代理只能够对接口进行代理，不能对普通的类进行代理（因为所有生成的代理类的父类为 Proxy，Java 类继承机制不允许多重继承）；CGLIB能够代理普通类；
* Java 动态代理使用 Java 原生的反射 API 进行操作，在生成类上比较高效；CGLIB 使用 ASM 框架直接对字节码进行操作，在类的执行过程中比较高效