## JDK 原生动态代理是怎么实现的 + 面试题

### 反射

反射机制是 Java 语言提供的一种基础功能，赋予程序在运行时自省（introspect）的能力。简单来说就是通过反射，可以在运行期间获取、检测和调用对象的属性和方法。

#### 反射的使用场景

在现实中反射的使用场景有很多，比如以下几个。

**使用场景一** ：编程工具 IDEA 或 Eclipse 等，在写代码时会有代码（属性或方法名）提示，就是因为使用了反射。

**使用场景二** ：很多知名的框架，为了让程序更优雅更简洁，也会使用到反射。

例如， **Spring** 可以通过配置来加载不同的类，调用不同的方法，代码如下所示：

<bean id="person" class="com.spring.beans.Person" init-method="initPerson">  
</bean>

例如， **MyBatis** 在 Mapper 使用外部类的 SQL 构建查询时，代码如下所示：

@SelectProvider(type = PersonSql.class, method = "getListSql")  
List<Person> getList();  
class PersonSql {  
 public String getListSql() {  
 String sql = new SQL() {{  
 SELECT("\*");  
 FROM("person");  
 }}.toString();  
 return sql;  
 }  
}

**使用场景三** ：数据库连接池，也会使用反射调用不同类型的数据库驱动，代码如下所示：

String url = "jdbc:mysql://127.0.0.1:3306/mydb";  
String username = "root";  
String password = "root";  
Class.forName("com.mysql.jdbc.Driver");  
Connection connection = DriverManager.getConnection(url, username, password);

当然反射还有其他很多类似的使用场景，这里就不一一列举，读者可以举一反三，想想在平常的开发中，还有哪些使用了反射功能的场景。

#### 反射的基本使用

下来我们通过反射调用类中的某个方法，来学习反射的基本使用。

使用反射调用类中的方法，分为三种情况：

* 调用静态方法
* 调用公共方法
* 调用私有方法

假设有一个实体类 MyReflect 包含了以上三种方法，代码如下：

package com.interview.chapter4;  
class MyReflect {  
 // 静态方法  
 public static void staticMd() {  
 System.out.println("Static Method");  
 }  
 // 公共方法  
 public void publicMd() {  
 System.out.println("Public Method");  
 }  
 // 私有方法  
 private void privateMd() {  
 System.out.println("Private Method");  
 }  
}

下面分别来看，使用反射如何调用以上三种类型的方法。

##### ① **反射调用静态方法**

Class myClass = Class.forName("com.interview.chapter4.MyReflect");  
Method method = myClass.getMethod("staticMd");  
method.invoke(myClass);

##### ② **反射调用公共方法**

Class myClass = Class.forName("com.interview.chapter4.MyReflect");  
// 创建实例对象（相当于 new ）  
Object instance = myClass.newInstance();  
Method method2 = myClass.getMethod("publicMd");  
method2.invoke(instance);

##### ③ **反射调用私有方法**

Class myClass = Class.forName("com.interview.chapter4.MyReflect");  
// 创建实例对象（相当于 new ）  
Object object = myClass.newInstance();  
Method method3 = myClass.getDeclaredMethod("privateMd");  
method3.setAccessible(true);  
method3.invoke(object);

#### 反射使用总结

反射获取调用类可以通过 Class.forName()，反射获取类实例要通过 newInstance()，相当于 new 一个新对象，反射获取方法要通过 getMethod()，获取到类方法之后使用 invoke() 对类方法进行调用。如果是类方法为私有方法的话，则需要通过 setAccessible(true) 来修改方法的访问限制，以上的这些操作就是反射的基本使用。

### 动态代理

动态代理可以理解为，本来应该自己做的事情，却交给别人代为处理，这个过程就叫做动态代理。

#### 动态代理的使用场景

动态代理被广为人知的使用场景是 Spring 中的面向切面编程（AOP）。例如，依赖注入 @Autowired 和事务注解 @Transactional 等，都是利用动态代理实现的。

动态代理还可以封装一些 RPC 调用，也可以通过代理实现一个全局拦截器等。

#### 动态代理和反射的关系

JDK 原生提供的动态代理就是通过反射实现的，但动态代理的实现方式还可以是 ASM（一个短小精悍的字节码操作框架）、cglib（基于 ASM）等，并不局限于反射。

下面我们分别来看：JDK 原生动态代理和 cglib 的实现。

#### 1）JDK 原生动态代理

interface Animal {  
 void eat();  
}  
class Dog implements Animal {  
 @Override  
 public void eat() {  
 System.out.println("The dog is eating");  
 }  
}  
class Cat implements Animal {  
 @Override  
 public void eat() {  
 System.out.println("The cat is eating");  
 }  
}  
  
// JDK 代理类  
class AnimalProxy implements InvocationHandler {  
 private Object target; // 代理对象  
 public Object getInstance(Object target) {  
 this.target = target;  
 // 取得代理对象  
 return Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(), this);  
 }  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 System.out.println("调用前");  
 Object result = method.invoke(target, args); // 方法调用  
 System.out.println("调用后");  
 return result;  
 }  
}  
  
public static void main(String[] args) {  
 // JDK 动态代理调用  
 AnimalProxy proxy = new AnimalProxy();  
 Animal dogProxy = (Animal) proxy.getInstance(new Dog());  
 dogProxy.eat();  
}

以上代码，我们实现了通过动态代理，在所有请求前、后都打印了一个简单的信息。

**注意：** JDK Proxy 只能代理实现接口的类（即使是 extends 继承类也是不可以代理的）。

#### 2）cglib 动态代理

要是用 cglib 实现要添加对 cglib 的引用，如果是 maven 项目的话，直接添加以下代码：

<dependency>  
 <groupId>cglib</groupId>  
 <artifactId>cglib</artifactId>  
 <version>3.2.12</version>  
</dependency>

cglib 的具体实现，请参考以下代码：

class Panda {  
 public void eat() {  
 System.out.println("The panda is eating");  
 }  
}  
class CglibProxy implements MethodInterceptor {  
 private Object target; // 代理对象  
 public Object getInstance(Object target) {  
 this.target = target;  
 Enhancer enhancer = new Enhancer();  
 // 设置父类为实例类  
 enhancer.setSuperclass(this.target.getClass());  
 // 回调方法  
 enhancer.setCallback(this);  
 // 创建代理对象  
 return enhancer.create();  
 }  
 public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  
 System.out.println("调用前");  
 Object result = methodProxy.invokeSuper(o, objects); // 执行方法调用  
 System.out.println("调用后");  
 return result;  
 }  
}  
public static void main(String[] args) {  
 // cglib 动态代理调用  
 CglibProxy proxy = new CglibProxy();  
 Panda panda = (Panda)proxy.getInstance(new Panda());  
 panda.eat();  
}

以上程序执行的结果：

调用前

The panda is eating

调用后

由以上代码可以知道，cglib 的调用通过实现 MethodInterceptor 接口的 intercept 方法，调用 invokeSuper 进行动态代理的。它可以直接对普通类进行动态代理，并不需要像 JDK 代理那样，需要通过接口来完成，值得一提的是 Spring 的动态代理也是通过 cglib 实现的。

**注意** ：cglib 底层是通过子类继承被代理对象的方式实现动态代理的，因此代理类不能是最终类（final），否则就会报错 java.lang.IllegalArgumentException: Cannot subclass final class xxx。

### 相关面试题

#### 1.动态代理解决了什么问题？

答：首先它是一个代理机制，如果熟悉设计模式中的代理模式，我们会知道，代理可以看作是对调用目标的一个包装，这样我们对目标代码的调用不是直接发生的，而是通过代理完成，通过代理可以让调用者与实现者之间解耦。比如进行 RPC 调用，通过代理，可以提供更加友善的界面；还可以通过代理，做一个全局的拦截器。

#### 2.动态代理和反射的关系是什么？

答：反射可以用来实现动态代理，但动态代理还有其他的实现方式，比如 ASM（一个短小精悍的字节码操作框架）、cglib 等。

#### 3.以下描述错误的是？

A：cglib 的性能更高  
B：Spring 中有使用 cglib 来实现动态代理  
C：Spring 中有使用 JDK 原生的动态代理  
D：JDK 原生动态代理性能更高

答：D

题目解析：Spring 动态代理的实现方式有两种：cglib 和 JDK 原生动态代理。

#### 4.请补全以下代码？

class MyReflect {  
 // 私有方法  
 private void privateMd() {  
 System.out.println("Private Method");  
 }  
}  
class ReflectTest {  
 public static void main(String[] args) throws ClassNotFoundException, NoSuchMethodException, InvocationTargetException, IllegalAccessException, InstantiationException {  
 Class myClass = Class.forName("MyReflect");  
 Object object = myClass.newInstance();  
 // 补充此行代码  
 method.setAccessible(true);  
 method.invoke(object);  
 }  
}

答：Method method = myClass.getDeclaredMethod(“privateMd”);

题目解析：此题主要考的是私有方法的获取，私有方法的获取并不是通过 getMethod() 方式，而是通过 getDeclaredMethod() 获取的。

#### 5.cglib 可以代理任何类这句话对吗？为什么？

答：这句话不完全对，因为 cglib 只能代理可以有子类的普通类，对于像最终类（final），cglib 是不能实现动态代理的，因为 cglib 的底层是通过继承代理类的子类来实现动态代理的，所以不能被继承类无法使用 cglib。

#### 6.JDK 原生动态代理和 cglib 有什么区别？

答：JDK 原生动态代理和 cglib 区别如下：

* JDK 原生动态代理是基于接口实现的，不需要添加任何依赖，可以平滑的支持 JDK 版本的升级；
* cglib 不需要实现接口，可以直接代理普通类，需要添加依赖包，性能更高。

#### 7.为什么 JDK 原生的动态代理必须要通过接口来完成？

答：这是由于 JDK 原生设计的原因，来看动态代理的实现方法 newProxyInstance() 的源码：

/\*\*  
 \* ......  
 \* @param loader the class loader to define the proxy class  
 \* @param interfaces the list of interfaces for the proxy class to implement  
 \* ......  
 \*/   
@CallerSensitive  
public static Object newProxyInstance(ClassLoader loader,  
 Class<?>[] interfaces,  
 InvocationHandler h)  
 throws IllegalArgumentException  
{  
// 省略其他代码

来看前两个参数的声明：

* loader：为类加载器，也就是 target.getClass().getClassLoader()
* interfaces：接口代理类的接口实现列表

看了上面的参数说明，我们就明白了，要使用 JDK 原生的动态只能通过实现接口来完成。

### 总结

通过本文可以知道 JDK 原生动态代理是使用反射实现的，但动态代理的实现方式不止有反射，还可以是 ASM（一个短小精悍的字节码操作框架）、cglib（基于 ASM）等。其中 JDK 原生的动态代理是通过接口实现的，而 cglib 是通过子类实现的，因此 cglib 不能代理最终类（final）。而反射不但可以反射调用静态方法，还可以反射调用普通方法和私有方法，其中调用私有方法时要设置 setAccessible 为 true。

[点击此处下载本文源码](https://github.com/vipstone/java-%20interview/tree/master/interview-code/src/main/java/com/interview)

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持