# 动态代理设计模式

## 静态代理

那么，到底什么是代理呢？

所谓的代理，其实就是中间人的意思。例如：让朋友代替你去取快递，你的朋友就充当了代理的作用。再比如，让你的朋友帮你去借款，你的朋友实际上也充当了代理的作用，最终这笔钱的受益人还是你自己。

理解了代理的意思，接下来我们一起来看一下，在面向对象编程语言中，到底应该如何体现代理呢。

这里我们就以上面提到的代取快递为例，来写一个简单的代理实现。

我们用Friend类表示你的朋友，用Self表示你自己，上面的例子用代码实现应该是这样：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | public class Friend {  private String name;   public Friend(String name) {  this.name = name;  }   public void collectPack() {  System.out.println(this.name + "去取快递...");  } } |

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | public class Self {  private Friend friend;   public Self(Friend friend) {  this.friend = friend;  }   // 由于实际动作的执行者是你的朋友  // 因此，这里直接调用Friend的collectPack方法  public void collectPack() {  friend.collectPack();  }   public static void main(String[] args) {  Friend friend = new Friend("张三");  Self self = new Self(friend);  // 这里将打印”张三去取快递...“  self.collectPack();  } } |

在上面这段代码中，我们实现了一个简单的代理。这里的代理类是Friend，被代理类是Self，通过这段代码，我们知道了一个基本事实：代理对象是真正去执行动作的对象，被代理对象是被动执行动作的对象（并不真正执行动作）。

上面的实现看似没有问题，实际上却不够友好，由于代理对象能够替被代理对象执行动作。所以，他们应该具有同样的一些方法。换句话说，应该实现他们应该实现同样的接口，这个接口中的方法表示双方都可以执行的一些动作，或者说可能要被代理的一些动作。

因此，上面的代码可以改写成下面这样：

|  |  |
| --- | --- |
| 1 2 3 4 5 | // 这个接口表示代理类与被代理类可以共同执行的动作 // 或者说，具体想要被代理的动作方法集合 public interface Collectable {  void collectPack(); } |

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | public class Friend implements Collectable {  private String name;   public Friend(String name) {  this.name = name;  }   public void collectPack() {  System.out.println(this.name + "去取快递...");  } } |

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | public class Self implements Collectable {  private Friend friend;   public Self(Friend friend) {  this.friend = friend;  }   // 由于实际动作的执行者是你的朋友  // 因此，这里直接调用Friend的collectPack方法  public void collectPack() {  friend.collectPack();  }   public static void main(String[] args) {  Friend friend = new Friend("张三");  Self self = new Self(friend);  self.collectPack();  } } |

恭喜你！通过上面的改造，我们已经完成了一个标准的静态代理实现。之所以称之为静态代理，是因为这里的逻辑是写死的，并不具备动态特性。与之相对的，就是今天这篇文章的主角：动态代理。

## 代理的作用是什么

看到这里，应该有人会问了，说了这么多，代理到底有什么用呢。这个问题并不容易回答，为了回答你的这个问题，我们先来看一个场景。

假设有一个类Driver，类中只有一个方法drive，我们不能改动这个类的源码，如何获取这个方法的执行时间呢？

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | public class Driver {    public void drive() throws InterruptedException {  System.out.println("I'm driving...");  Thread.sleep(1000);  System.out.println("Drive completed...");  } } |

有的同学可能会说，这还不简单，在main方法中，方法执行前记录一个时间，方法执行后记录一个时间，两个时间相减就得到了方法最终的执行时间。

**位置p1**

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | public static void main(String[] args) throws InterruptedException {  Driver driver = new Driver();   long start = System.currentTimeMillis();  driver.drive();  long end = System.currentTimeMillis();   System.out.println(end - start); } |

这个方法到底对不对呢，先说结论：不对！这里的时间会大于方法实际执行的时间，因为这里包含了准备方法的那些时间。

除了这个方法之外，还有一个比较容易想到的方法就是继承。通过继承Driver类，在drive方法前后打印时间，计算时间差，这种方式似乎可行！

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | public class Driver1 extends Driver {   @Override  public void drive() throws InterruptedException {  long start = System.currentTimeMillis();  super.drive();  long end = System.currentTimeMillis();  System.out.println(end - start);  } } |

**位置p2**

|  |  |
| --- | --- |
| 1 2 3 4 | public static void main(String[] args) throws InterruptedException {  Driver1 driver1 = new Driver1();  driver1.drive(); } |

这里有一个疑点，有些同学在问，到底方法的准备时间是什么，为什么会有准备方法的这些时间。这两种方式不是都需要准备方法吗？

上面我们说到，直接打印的方式包含了准备方法的那些时间，准备一个方法通常需要先压栈，调用后自动出栈，这些都需要时间，尤其在一些性能比较低的机器上会体现的特别明显。不信，大家可以执行位置p1处的代码与位置p2处的代码，你会发现，位置p2处的执行时间几乎总是比位置p1处的执行时间少3~5毫秒（在我的Macbook Pro 15.4 2017上执行是这样的结果，其它机型时间可能略有差异）。在继承中不需要这些时间的原因是：我们是方法内部执行的，这个时候方法已经准备好，就不存在这个准备时间了。

Ok，说完了上面这个问题，我们继续回到上面的代码。在上面的代码中，我们通过继承的方式获得了方法的执行时间。接下来，新需求来了，我要你在drive方法前后各打印一条日志。

你会怎么做呢，当然毫无疑问，我们继续沿用上面的解决方案，继承Driver类创建新类Driver2。然后，在super调用前后各打印一条日志。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | public class Driver2 extends Driver {   @Override  public void drive() throws InterruptedException {  System.out.println("Drive start...");  super.drive();  System.out.println("Drive complete...");  } } |

问题似乎很简单，可是，新的需求又来了。我要你先打印日志再获取方法的执行时间。其实，这也很简单。我们只要继承上面的Driver2类，重写drive方法获取方法执行的时间即可：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | public class Driver3 extends Driver2 {   @Override  public void drive() throws InterruptedException {  long start = System.currentTimeMillis();  super.drive();  long end = System.currentTimeMillis();  System.out.println(end - start);  } } |

接下来，麻烦来了。需求又变了，我要你先获取方法的执行时间再打印日志，怎么办。

有人说，这也没毛病啊。我们只要继承Driver1创建新类Driver4，然后在drive方法中打印日志即可：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 | public class Driver4 extends Driver1 {   @Override  public void drive() throws InterruptedException {  System.out.println("Drive start...");  super.drive();  System.out.println("Drive complete...");  } } |

也许你已经发现了，这个解决方案存在着明显的问题，如果一个类有100个方法，实现上述这些逻辑大概需要创建400个类，这显然不是一个可取的方法。那么，是否有更好的解决方案呢？

在上面的解决方案中，我们通过继承的方式获取到了父类方法的执行时间。但是，如果Driver类被final修饰呢，大家知道final类是无法被继承的，继承这条路显然走不通了。

但这似乎恰好为我们打开了一扇窗，我们尝试使用文章开头静态代理的方式传入不同功能的Driver类实例，看看能否发生一些特殊的化学反应。

同样，我们以获取方法的执行时间为例，如果要通过静态代理的方式获取方法的执行时间，我们应该这样做：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | public class Driver5 {  private Driver driver;   public Driver5(Driver driver) {  this.driver = driver;  }   public void drive() throws InterruptedException {  long start = System.currentTimeMillis();  driver.drive();  long end = System.currentTimeMillis();  System.out.println(end - start);  } } |

前面说到，我们应该将统一的动作抽象化。因此，这里我们新增统一的接口Drivable，我们让所有的Driver类都实现Drivable接口：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | public class Driver5 implements Drivable {  private Driver driver;   public Driver5(Driver driver) {  this.driver = driver;  }   public void drive() throws InterruptedException {  long start = System.currentTimeMillis();  driver.drive();  long end = System.currentTimeMillis();  System.out.println(end - start);  } } |

同样地，为了在方法的执行前后打印日志，我们创建Driver6通过代理的方式实现日志的打印：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | public class Driver6 implements Drivable {  private Driver driver;   public Driver6(Driver driver) {  this.driver = driver;  }   public void drive() throws InterruptedException {  System.out.println("Drive start...");  driver.drive();  System.out.println("Drive complete...");  } } |

为了与前面的方式区分开来，我们将Driver5重命名为DriverTimeProxy，Driver6重命名为DriverLogProxy，接下来我们尝试先打印日志再获取方法的执行时间。

咋一看，似乎毫无头绪。别急，我们先尝试将聚合对象Driver抽象化。参数类型修改为Drivable（由于Driver类也实现了Drivable接口），修改完成后代码如下：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | public class DriverLogProxy implements Drivable {  private Drivable drivable;   public DriverLogProxy(Drivable drivable) {  this.drivable = drivable;  }   public void drive() throws InterruptedException {  System.out.println("Drive start...");  drivable.drive();  System.out.println("Drive complete...");  } } |

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | public class DriverTimeProxy implements Drivable {  private Drivable drivable;   public DriverTimeProxy(Drivable drivable) {  this.drivable = drivable;  }   public void drive() throws InterruptedException {  long start = System.currentTimeMillis();  drivable.drive();  long end = System.currentTimeMillis();  System.out.println(end - start);  } } |

至此，神奇的化学反应出现了。由于DriverLogProxy与DriveTimeProxy都实现了Drivable接口，我们可以将这两个对象相互聚合到对方的类中。

例如，如果我们要先打印方法的执行时间再打印日志，可以这样做：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | public static void main(String[] args) throws InterruptedException {  Driver driver = new Driver();    DriverTimeProxy driverTimeProxy = new DriverTimeProxy(driver);  DriverLogProxy driverLogProxy = new DriverLogProxy(driverTimeProxy);    driverLogProxy.drive(); } |

而如果我们要先打印日志再打印方法的执行时间，可以这样做：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | public static void main(String[] args) throws InterruptedException {  Driver driver = new Driver();   DriverLogProxy driverLogProxy = new DriverLogProxy(driver);  DriverTimeProxy driverTimeProxy = new DriverTimeProxy(driverLogProxy);   driverTimeProxy.drive(); } |

在前面的例子中，为了实现相同的功能，我们至少需要创建四个类，而这里似乎只需要两个类就搞定了。显然，这种通过代理处理的方式更优。

## 动态代理