代理模式

|  |  |
| --- | --- |
|  | 🢂 内容概览 |
|  | Why：此文档用来做什么？它存在的意义是什么？为解决什么问题？   |  | | --- | | **定义：** Provide a surrogate（代理） or placeholder（占位符） for another object to control access（控制访问） or append ability（赋能） to it.  **类型：** 结构型模式  **场景：** 访问控制，赋能  **思路：** 为调用者和被调用者之间留有余地，以应对变化 |   What：当前包含了那些内容？   |  | | --- | |  |   How：此文档应如何参考？   |  | | --- | |  |   Who：此文档适用于那些人员阅读参考？   |  | | --- | |  |   Summary：摘要   |  | | --- | |  |   Reference：参考文献   |  | | --- | | [Spring AOP 实现原理—-AspectJ与CGLIB介绍](http://blog.csdn.net/wenbingoon/article/details/8988553)  [JVM即时编译（JIT）](http://blog.csdn.net/sunxianghuang/article/details/52094859) | |

目录

[1 模式描述 3](#_Toc500353860)

[2 模式类图 3](#_Toc500353861)

[3 模式实现 4](#_Toc500353862)

[3.1 Subject 4](#_Toc500353863)

[3.2 RealSubject 4](#_Toc500353864)

[3.3 Proxy 5](#_Toc500353865)

[3.4 Client 5](#_Toc500353866)

[4 模式扩展 6](#_Toc500353867)

[4.1 通过继承实现代理 6](#_Toc500353868)

[4.2 动态代理 7](#_Toc500353869)

[5 适用场景 7](#_Toc500353870)

[5.1 远程代理 7](#_Toc500353871)

[5.2 虚拟代理 8](#_Toc500353872)

[5.3 保护代理 8](#_Toc500353873)

[5.4 缓存代理 8](#_Toc500353874)

[5.5 同步代理 8](#_Toc500353875)

[5.6 智能引用代理 8](#_Toc500353876)

[5.7 日志代理 8](#_Toc500353877)

[5.8 分布式代理 8](#_Toc500353878)

[6 权衡点 8](#_Toc500353879)

[6.1 优点 8](#_Toc500353880)

[6.2 缺点 9](#_Toc500353881)

[7 应用案例 9](#_Toc500353882)

[8 相关原则 9](#_Toc500353883)

[9 相关模式 10](#_Toc500353884)

[10 问题思考 10](#_Toc500353885)

# 模式描述

在调用者和被调用者之间添加一个代理层，由代理层控制目标对象的引用，作为被调用者的替身，增加了访问的封装性、灵活性、扩展性。

* 封装性：代理封装了被调用者的访问细节，不需要上层调用者了解细节和介入实现；
* 灵活性：代理层对被调用者的时机、方式可以灵活控制；
* 扩展性：代理层可以给被调用者赋予新的能力，或者强化已有能力。

代理模式给调用者和被调用者之间增加了一个缓冲，从而为以后的变化留有余地，从一定程度上降低了耦合。

|  |
| --- |
| D:\share\01_project\mynote\201706_设计模式\20171129 代理模式\模式描述1.png |

# 模式类图

|  |
| --- |
| D:\share\01_project\mynote\201706_设计模式\20171129 代理模式\代理模式类图1--基本类图.png |

代理模式主要包含如下4个角色：

* **Subject（被调用者的抽象）：**定义被调用者想要/能够提供的服务。request()方法是要提供的服务；

注：此处对于抽象类的理解既可以是一个抽象类（abstract class），也可以是一个或者多个接口（interface），根据被访问对象实际需要暴露和提供的能力而定，不需要局限于类图。

* **RealSubject（被调用者的具体实现）：**服务的具体实现；
* **Client（调用者） ：** 服务的调用者，不局限于单个对象，可以是任何上层应用或者服务；
* **Proxy（访问代理）：**作为”代理”存在于调用者和被调用者之间，代替调用者访问目标服务；代理可以自行决定何时以及何种方式调用request()方法，同时提供额外的before()、after()等额外方法用于扩展目标服务的能力；

**注：**设计模式提供的是一种设计思路，所以代理模式的实现并不局限于类图的实现，其中的角色依据看待问题的视角可大可小，即可以是一个具体的类或者对象，也可以是一个组件、一个服务、一种资源…

代理模式中的Proxy和RealSubject就好比经纪人和明星的关系。经纪人负责明星的公共事务，比如：签约、谈片酬、财务管理，危机公关等，这些事情都可以划在Subject范畴，即经纪人的本职工作。同时经纪人本身又可以做一些自己的事情，比如：transfer the actor’s property and visit his wife sometime.

# 模式实现

## Subject

|  |
| --- |
| /\*\*  \* 被调用者的抽象  \*/  public interface Subject {  /\*\*  \* 服务抽象  \*/  void request();  } |

## RealSubject

|  |
| --- |
| /\*\*  \* 被调用者的具体实现  \*/  public class RealSubject implements Subject {  @Override  public void request() {  // 服务的具体实现...  System.out.println("real service is called.");  }  } |

## Proxy

|  |
| --- |
| /\*\*  \* 代理类  \*/  public class Proxy implements Subject {  // 存放被代理的对象  private Subject subject;  // 注入方式可扩展，亦可以是setter注入、通过元数据生成...  public Proxy(Subject subject) {  // 被代理的对象由上层决定，传入什么对象，就代理什么对象；  this.subject = subject;  }  @Override  public void request() {  // 代理操作的实现，可以根据实际扩展，充满想象...  // 调用方式：同步/异步...  // 调用时机：立即/延迟...  // 调用目的：扩展功能、权限控制、过滤...  before();  subject.request();  after();  }  /\*\*  \* 对目标对象服务的具体定制，可以是一群方法、类、服务...  \*/  public void before() {  System.out.println("called before request().");  }  public void after() {  System.out.println("called after request().");  }  } |

## Client

|  |
| --- |
| /\*\*  \* 调用者  \*/  public class Client {  public static void main(String[] args) {  // 目标对象和代理的创建时间和地点可以不同，此处仅示意  RealSubject subject = new RealSubject();  Proxy proxy = new Proxy(subject);  // 执行代理方法  proxy.request();  }  } |

运行结果

|  |
| --- |
| called before request().  real service is called.  called after request(). |

# 模式扩展

## 通过继承实现代理

代理模式还有一种简化模式，即代理类直接继承被调用者，而无须继承抽象接口。根据继承的特性，子类可以通过super关键字调用父类的公有方法，从而实现代理操作。类图如下：

|  |
| --- |
| D:\share\01_project\mynote\201706_设计模式\20171129 代理模式\代理模式类图2--通过继承实现代理.png |

通过继承实现代理的虽然方式简单，但是很显然违背了合成复用原则。然而我们不是为了模式而模式，原则只是为了更好的指导软件开发，而并非强制的规则，有时为了满足业务需求，违背原则也无妨。简化模式的具体实现如下，RealSubject无变化，此处仅列出代理类：

|  |
| --- |
| /\*\*  \* 代理类  \*/  public class Proxy extends RealSubject {  @Override  public void request() {  // 代理操作的实现，可以根据实际扩展，充满想象...  // 调用方式：同步/异步...  // 调用时机：立即/延迟...  // 调用目的：扩展功能、权限控制、过滤...  before();  super.request();  after();  }  /\*\*  \* 对目标对象服务的具体定制，可以是一群方法、类、服务...  \*/  public void before() {  System.out.println("called before request().");  }  public void after() {  System.out.println("called after request().");  }  } |

我们常用的cglib在实现动态代理时候，用的即是这种方式，将动态生成的代理类作为被调用类的子类，以获得访问权限。

## 动态代理

# 适用场景

## 远程代理

要访问的资源在远端，通过代理封装远程资源的访问，屏蔽操作的复杂性。调用者调用代理对象如同调用本地对象一样，对调用者透明。远程代理可以作为分布式RPC的实现思路，比如：分布式服务框架Dubbo通过创建本地Service来代理远程服务，实现透明化的远程方法调用，即是一种远程代理的实现。

## 虚拟代理

“以小见大”、”懒加载”，对于创建或者加载消耗性能较大的资源，可以先创建一个消耗较小的代理，等到真正需要调用资源的时候，再通过代理进行资源的创建和加载，实现”按需加载”。代理可以是资源的”缩略图”，也可以是资源的一部分。

## 保护代理

实现资源访问权限控制，通过在代理层增加权限判断，保护资源的安全性。可以作为权限控制的实现思路。

## 缓存代理

代理对被调用这的结果进行缓存，从而降低反复调用带来的性能消耗。比如：通过spring-cache将查询结果缓存到内存中，对于相同条件的调用，直接访问缓存，提高查询性能，便是缓存代理的实现。

## 同步代理

通过代理对象实现对被调用者的调用同步，防止冲突，保证并发安全；

## 智能引用代理

通过代理实现对象的引用计数。

## 日志代理

通过代理记录对象访问日志；

## 分布式代理

对于分布式系统，提供统一的代理实现。使得访问分布式系统和访问单点无差别。比如：twemproxy作为redis集群的代理，屏蔽了redis集群底层分片部署的复杂性，在客户端看来和直接访问redis单点无差别。

# 权衡点

## 优点

* 一定程度上降低了耦合；
* 降低调用服务的复杂性；
* 降低消耗，提高性能；（虚拟代理、缓存代理）
* 提升安全性；（保护代理）

## 缺点

* 相比于直接调用而言，增加了一层，增加请求耗时；
* 实现代理本身较为复杂；

# 应用案例

**1）AOP（Aspect Oriented Programming，面向切面的编程）**，通过预编译技术或者运行时动态生成字节码技术实现对原有类/对象能力的增强。通常分为”静态代理”和”动态代理”两类，”静态”和”动态”是根据生成AOP代理类的时机进行的划分，对应到”编译时”和”运行时”两个阶段。

静态代理，就是在.java文件编译成.class文件时，通过更改.class的字节码的方式为原有类生成AOP代理，在jvm加载类文件后，字节码已经固定，不会再有改变，因而又称为编译时增强。代表就是AspectJ框架。

动态代理，是通过动态字节码生成技术，在运行时”临时”生成AOP代理对象，代理对象对应的类文件是在运行时动态编译生成的，因而又称为运行时增强。常用工具有JDK原生的动态代理，以及Cglib动态搭理。

# 相关原则

|  |  |  |  |
| --- | --- | --- | --- |
| 原则名称 | 符合 | 违反 | 描述 |
| 单一职责原则 | √ |  | Client、RealSubject仅负责自己的业务实现，不用关注职责以外的事情； |
| 里氏替换原则 | √ |  | Proxy关联Suject而非RealSubject，能够代理具体子类的操作； |
| 依赖倒转原则 | √ |  | 同上，Proxy本身针对Subject抽象编程，没有针对具体的RealSubject实现编程； |
| 接口隔离原则 | √ |  | Subject如果划分合理，可以做到接口隔离，比如屏蔽不需要代理的方法，由RealSubject在子类中实现； |
| 迪米特法则 | √ |  | 同上，Subject如果划分合理，仅暴露需要上层知道的服务，则符合； |
| 开闭原则 | √ |  | 修改具体的RealSubject，对上层不可见；如需要新增服务，可以通过实现新的RealSubject来扩展，故符合； |
| 合成复用原则 | √ |  | Proxy调用RealSubject，使用IOC方式，属于关联关系，故符合； |

# 相关模式

装饰模式，状态模式

# 问题思考

1）代理模式和装饰模式、状态模式有何区别？

2）静态代理和动态代理有何异同？

3）JDK原生动态代理与cglib动态代理有何异同？

4）为什么被final修饰的方法不能够被重写？