# 代理模式简介

## 概述

1. 由于某些原因，客户端不想或不能直接访问一个对象，此时可以通过一个代理称为“代理”的第三者来实现间接访问，该方案对应的设计模式称为代理模式。为了保证客户端的使用的透明性，所访问的真实对象与代理对象需要实现相同的接口。
2. 根据代理模式的使用目的不同，代理模式又分为多种类型，例如保护代理、远程代理、虚拟代理、缓冲代理等。
3. 代理模式（Proxy Pattern）的定义如下：给某一个对象提供一个代理或占位符，并由代理对象来控制对原对象的访问。
4. 代理模式是一种对象结构型模式。在代理模式中引入一个新的代理对象，代理对象在客户端对象和目标对象起到中介的作用。



## 结构

代理模式结构如下：



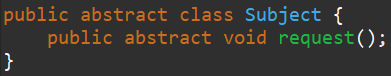
由图可知：

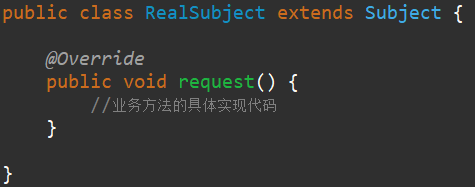
1. Subject（抽象主题角色）：它是真实主题和代理主题的共同接口，这样一来在任何使用真实主题的地方都可以使用代理主题，客户端通常需要针对抽象主题角色进行编程。
2. Proxy（代理主题角色）：它是抽象主题角色的子类，通常是一个具体类；它需要包含一个真实主题对象的引用，从而可以在任何时候操作真实主题对象；在代理主题角色中提供了一个与真实主题角色相同的接口，以便在任何时候都可以替代真实主题；代理主题角色还可以控制对真实主题的使用，负责在需要的时候创建和删除真实主题对象，并对真实主题对象的使用加以约束。通常，在代理主题角色中客户端在调用所引用的真实主题操作之前或之后还需要执行其他操作，而不仅仅是单纯调用真实主题对象中的操作。
3. RealSubject（真实主题角色）：它也是抽象主题角色的子类，定义了代理角色所代表的真实对象，在真实对象角色中实现了真实的业务操作，客户端可以通过代理主题角色间接调用真实主题角色中定义的操作。

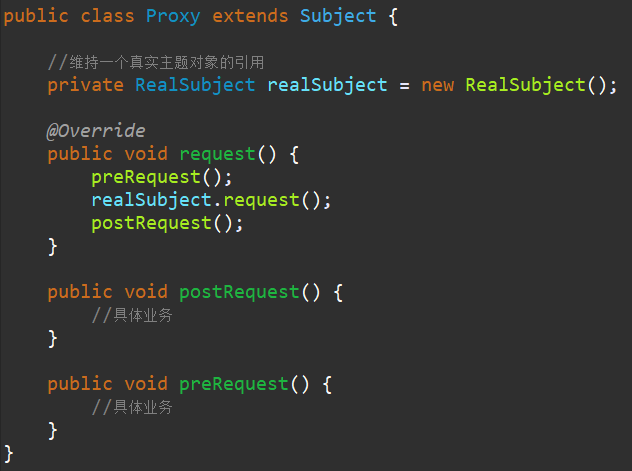
# 实现

## 实现原理

1. 抽象主题类声明了真实主题类和代理类的公共方法，它可以是接口、抽象类或具体类，客户端针对抽象主题类进行编程，一致性地对待真实主题和代理主题。典型的抽象主题类代码如下：



1. 真实主题类继承了抽象主题类，提供了业务方法的具体实现。其典型代码如下：
2. 代理类也是抽象主题类的子类，它维持一个对真实主题对象的引用，调用在真实主题中实现的业务方法，在调用时可以在原有业务方法的基础上附加一些新的方法对功能进行扩充或约束。最简单的代理类实现代码如下：



## 实例

实例说明：某软件公司某信息咨询公司的收费商务信息查询系统的开发任务，该系统的基本需求如下：

1. 在进行商务信息查询之前用户需要通过身份验证，只有合法用户才能使用该查询系统
2. 在进行商务信息查询时系统需要记录查询日志，以便根据查询次数收取查询费用

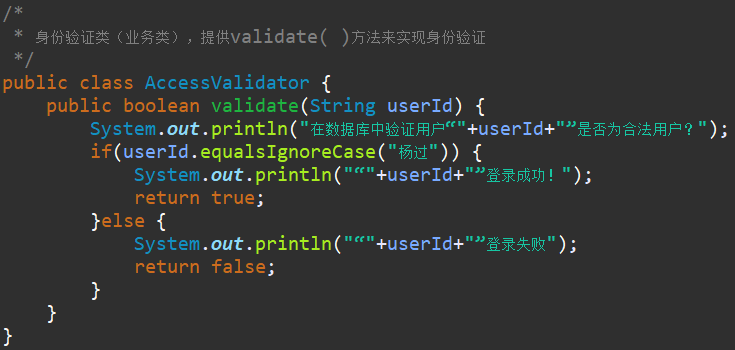
该软件公司的开发人员已完成了商务信息查询模块的开发任务，现希望能够以一种松耦合的方式向原有系统增加身份验证和日志记录功能，客户端代码可以无区别地对待原始的商务查询模块和增加新功能之后的商务信息查询模块，而且可能在将来还要在该信息查询模块中增加一些新的功能。使用代理模式设计并实现该收费商务信息查询系统。



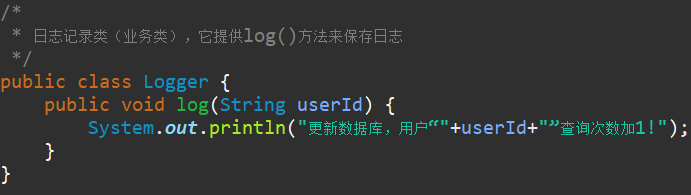


在上述结构图中，业务类AccessValidator用于验证身份，业务类Logger用于记录用户查询日志，Searcher充当抽象主题角色，RealSearcher充当真实主题角色，ProxySearcher充当代理主题角色。

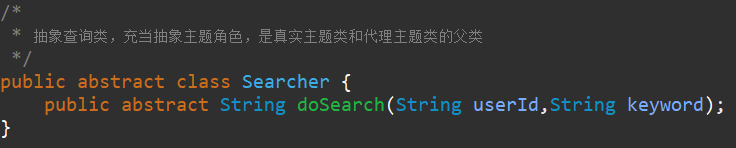
1. AccessValidator：身份验证类（业务类），它提供validate ( ) 方法来实现身份验证。



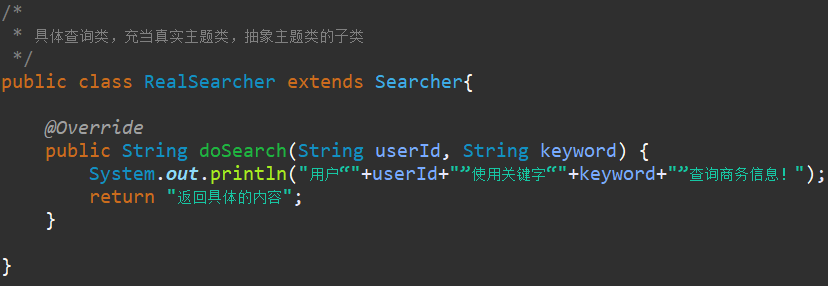
1. Logger：日志记录类（业务类），它提供log ( ) 方法来保存日志。



1. Searcher：抽象查询类，充当抽象主题角色，它声明了doSearch ( ) 方法。



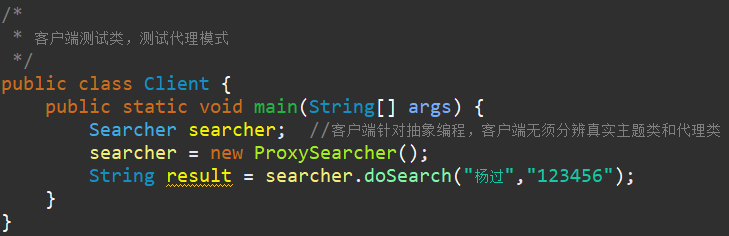
1. RealSearcher：具体查询类，充当真实主题角色，它实现真正的查询功能，提供doSearch ( ) 方法来查询信息.



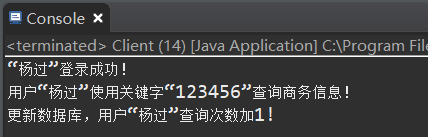
1. ProxySearcher：代理查询类，充当代理主题角色，它是查询代理，维持了对RealSeacher对象，AccessValidator对象和Logger对象的引用。



1. Client：客户端测试类，测试代理模式。



运行结果：



本实例是保护代理和只能引用代理的应用实例。

# 分类

代理模式根据其目的和实现方式不同可分为很多种类，对其中常用的几种代理模式简要说明如下：

1. 远程代理（Remote Proxy）：为一个位于不同地址空间的对象提供一个本地的代理对象，这个不同的地址空间可以在同一台主机中，也可以在另一台主机中，远程代理又称为大使（Ambassador）。
2. 虚拟代理（Virtual Proxy）：如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，用它来代理真实对象，而真实对象只在需要时才会被真正创建。
3. 保护代理（Protect Proxy）：控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限。
4. 缓存代理（Cache Proxy）：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。
5. 智能引用代理（Smart Reference Proxy）：当一个对象被引用时提供一些额外的操作，例如将对象被调用的次数记录下来等，或者记录日志、功能增强等。

## 远程代理

1. 远程代理（Remote Proxy）是一种常用的代理模式，它使得客户端程序可以访问在远程主机上的对象，远程主机可能具有很好的计算性能与处理速度，可以快速响应并处理客户端的请求。远程代理可以将网络的细节隐藏起来，使得客户端不必考虑网络的存在。客户端完全可以认为被代理的远程业务对象是在本地而不是在远程，而远程代理对象承担了大部分的网络通信工作，并负责对远程业务方法的调用。
2. 远程代理示意图如下，客户端对象不能直接访问远程主机中的业务对象，只能采取间接访问的方式。远程业务对象在本地主机中有一个代理对象，该代理对象负责对远程业务对象的访问和网络通信，它对于客户端对象而言是透明的。客户端无须关心实现具体业务的是谁，只需要按照服务接口所定义的方式直接与本地主机中的代理对象交互即可。
3. 在Java语言中可以通过一种RMI（Remote Method Invocation，远程方法调用）的机制来实现远程代理，它能够实现一个Java虚拟机中的对象调用另一个Java虚拟机中对象的方法。在RMI中。客户端对象可以通过一个桩（Stub）对象与远程主机上的业务对象进行通信，由于桩对象和远程业务对象接口一致，因此对于客户端而言操作远程对象和本地桩对象没有任何区别，桩对象就是远程业务对象在本地主机的代理对象。

1. 在RMI实现的过程中，远程主机端有一个Skeleton（骨架）对象来负责与Stub对象通信，RMI的基本实现步骤如下：
2. 客户端发起请求，将请求转交给RMI客户端的Stub类。
3. Stub类将请求的接口、方法、参数等信息进行序列化。
4. 将序列化后的流使用Socket传输至服务器端。
5. 服务器端接收到流后将其转发给相应的Skeleton类。
6. Skeleton类将请求信息反序列化后调用实际的业务处理类。
7. 业务处理类处理完毕后将结果返回给Skeleton类。
8. Skeleton类将结果序列化，再次通过Socket将流传给客户端的Stub。
9. Stub在接收到流后进行反序列化，将反序列化后得到的Java Object返回给客户端调用者。

至此，一次完整的远程方法调用得以完成。除了RMI之外，Java语言还可以通过很多其他方式来实现远程通信和远程方法调用，例如XML-RPC、Binary-RPC、JBoss-Remoting、Spring-Remoting、Hessian等。

## 虚拟代理

虚拟代理（Virtual Proxy）也是一种常用代理模式，对于一些占用系统资源较多或者加载时间较长的对象，可以给这些对象提供一个虚拟代理。在真实对象创建成功之前虚拟代理扮演真实对象的替身，而当真实对象创建之后虚拟代理将用户的请求转发给真实对象。通常在以下两种情况下可以考虑使用虚拟代理：

1. 由于对象本身的复杂性或者网络等原因导致一个对象需要较长的加载时间，此时可以用一个加载时间相对较短的代理对象来代表真实对象。通常在实现时可以结合多线程技术，一个线程用于显示代理对象，其他线程用于加载真实对象。这种虚拟代理模式可以应用在程序启动的时候，由于创建代理对象在时间和处理复杂度上要少于创建真实对象，因此在程序启动时可以用代理对象代理真实对象初始化，大大加速了系统的启动时间。当需要使用真实对象时再通过代理对象引用，而此时真实对象可以加载完毕，可以缩短用户的等待时间。
2. 当一个对象的加载十分耗费系统资源的时候也非常适合使用虚拟代理。虚拟代理可以让那些占用大量或处理起来非常复杂的对象推迟到使用它们的时候才创建，而在此之前用一个相对来说占用资源较少的代理对象来代表真实对象，再通过代理对象来引用真实对象。为了节省内存，在第一个引用真实对象时再创建对象，并且该对象可以被多次重复，在以后每次访问时需要检测所需对象是否已经被创建，因此在访问该对象时需要记性存在性检测，这需要消耗一定的系统时间，但是可以节省内存空间，这是一种用时间换取空间的做法。

无论是以上哪种情况，虚拟代理都是用一个“虚假”的代理对象来代理真实对象，通过代理对象来间接引用真实对象，可以在一定程度上提高系统的性能。

# Java动态代理

## 概述

1. 在传统的代理模式中客户端通过Proxy类调用RealSubject类的request ( ) 方法，同时还可以在代理类中封装其他方法（即在调用真实主题类中的业务方法前后进行增强）。如果按照这种方法使用代理模式，那么代理类和真实主题类都应该是事先已经存在的，代理的接口和所代理的方法都以明确指定。每一个代理类在编译之后都会生成一个class文件，代理类所实现的接口和代理的方法都被固定，这种代理被称为静态代理（Static Proxy）。如果需要为不同的真实主题类提供一个代理类，需要增加新的代理类，这将导致系统中类的个数急剧增加，因此需要想办法减少系统中类的个数。
2. 动态代理（Dynamic Proxy）可以让系统在运行时根据实际需要来动态创建代理类，让同一个代理类能够代理多个不同的真实主题类而且可以代理不同的方法。动态代理是一种较为高级的代理模式，它在事务管理、AOP（Aspect – Orented Programming，，面向切面编程）等领域都发挥了了重要的作用。

3）Java语言实现动态代理需要用到java.lang.reflect包中的一些类，如下：

1. **Proxy类**

Proxy类提供了用于创建动态代理类和实例对象的方法，它是所创建的动态代理类的父类，它最常用的方法如下：

1. public static Class<?> getProxyClass( ClassLOader loader , Class<?>… interface )：该方法用于返回一个Class类型的代理类，在参数中需要提供类加载器并需要指定代理的接口数组（与真实主题类的实现的接口列表一致）。
2. public static Object newProxyInstance ( ClassLoader loader , Class<?> [ ] interface , invocationHandler h )：该方法用于返回一个动态创建的代理类的实例，方法中的第一个参数loader表示代理类的类加载器，第二个参数interface表示代理类所实现的接口列表（与真实主题类的实现的接口列表一致），第三个参数h表示所指派的调用处理程序类。
3. **InvocationHandler接口**

InvocationHandler接口是代理处理程序类的实现接口，该接口作为代理实例的调用处理者的公共父类，每一个代理类的实例都可以额提供一个相关的具体调用处理者（InvocationHandler接口的子类）。在该接口中声明了如下方法：

public Object invoke ( Object proxy , Method method , Object[ ] args )

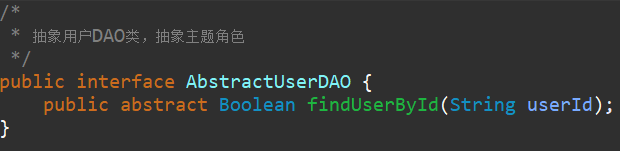
该方法用于处理对代理类实例的方法调用并返回相应的结果，当一个代理实例中的业务方法被调用时将自动调用该方法。其中，第一个方法的参数proxy表示代理类的实例，第二个参数method表示需要代理的方法，第3个参数args表示代理方法的参数数组。

动态代理类需要在运行时指定所代理真实主题类的接口，客户端在调用动态代理对象的方法时调用请求会将请求自动转发给InvocationHandler对象的invoke ( ) 方法，由invoke ( )方法来实现对请求的统一处理。

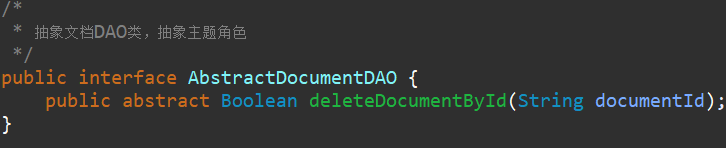
## 实例

实例说明：某软件公司要为公司OA系统数据访问层DAO增加方法调用日志，记录每一个方法被调用的时间和调用结果，现使用动态代理进行设计和实现。

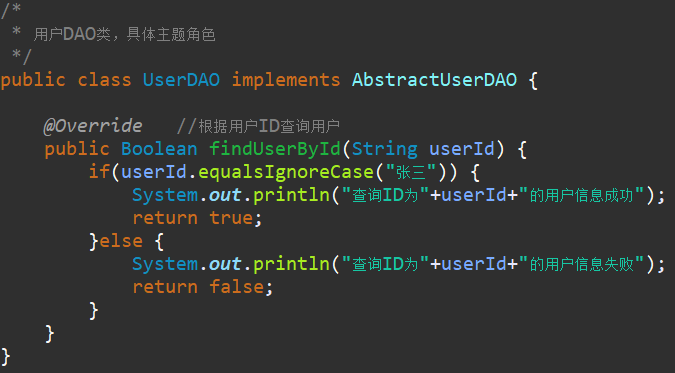
1. AbstractUserDAO：抽象用户DAO类，抽象主题角色。



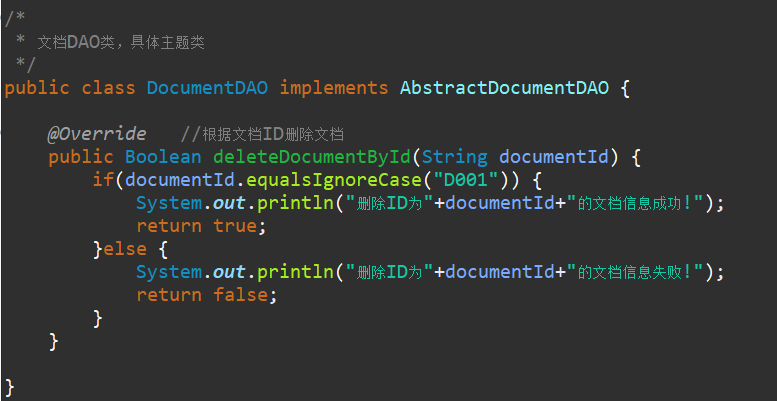
1. AbstractDocumentDAO：抽象文档DAO类，抽象主题角色。



1. UserDAO：用户DAO类，具体主题角色。



1. DocumentDAO：文档DAO类，具体主题角色。



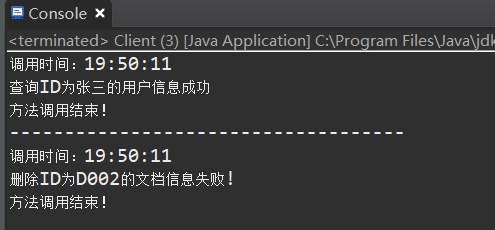
1. DAOLogHandler：自定义请求处理程序类。（实现动态代理的关键类）



1. Client：客户端测试类。



运行结果：



# 优缺点及适用环境

代理模式是常用的结构型设计模式之一，它为对象的间接访问提供了一个解决方案，可以对对象的访问进行控制。

## 优点

代理模式的共同优点（因为代理模式可以分为很多种类）如下：

1. 能够协调调用者和被调用者，在一定程度上降低了系统的耦合度。
2. 客户端可以针对抽象主题角色进行编程，增加和更换代理类无须修改源代码，符合开闭原则，系统具有较好的灵活性和可扩展性。

此外，不同类型的代理模式具有独特的优点，例如：

1. 远程代理为位于两个不同地址空间的对象的访问提供了一种实现机制，可以将一些消耗资源较多的对象和操作移至性能更好的计算机上，提高了系统的整体运行效率。
2. 虚拟代理通过一个消耗资源较少的对象来代表一个消耗资源较多的对象，可以在一定程度上节省系统的运行开销。
3. 缓存代理为某一个操作的结果提供临时的缓存存储空间，以便在后续使用中能够共享这些结果，优化系统性能，缩短执行时间。
4. 保护代理可以控制一个对象的访问权限，为不同用户提供不同级别的使用权限。

## 缺点

代理模式的缺点主要如下：

1. 由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢，例如保护代理。
2. 实现代理模式需要额外的工作，而且有些代理模式的实现过程较为复杂，例如远程代理。

## 适用环境

代理模式的类型较多，不同类型的代理模式有不同的优缺点，它们应用于不同的场合。

1. 当客户端对象需要访问远程主机中的对象时可以使用远程代理。
2. 当需要用一个消耗资源较少的对象来代表一个消耗资源较多的对象，从而降低系统开销、缩短运行时间时可以使用虚拟代理，例如一个对象需要很多时间才能完全加载。
3. 当需要为某一个被频繁访问的操作结果提供一个临时存储空间，以供多个客户端共享访问这些结果时可以缓存代理。通过使用缓冲代理，系统无须在客户端每一个次访问时都重新执行操作，只需直接从临时缓存区获取操作结果即可。
4. 当需要控制对一个对象的访问为不同用户提供不同级别的访问权限时可以使用保护代理。
5. 当需要为一个对象的访问（引用）提供一些额外的操作时可以使用智能引用代理。