
南京大学

# 设计模式–结构型模式(三)

# Design Pattern-Structural Pattern (3)

# 结构型模式概述

| 模式名称 | 定 义 | 学习难度 | 使用频率 |
| --- | --- | --- | --- |
| 适配器模式<br>(Adapter Pattern) | 将一个类的接口转换成客户希望的另一个接口。适配器模式让那些接口不兼容的类可以一起工作。 | ★★☆☆☆ | ★★★★☆ |
| 桥接模式<br>(Bridge Pattern) | 将抽象部分与它的实现部分解耦，使得两者都能够独立变化。 | ★★★☆☆ | ★★★☆☆ |
| 组合模式<br>(Composite Pattern) | 组合多个对象形成树形结构，以表示具有部分-整体关系的层次结构。组合模式让客户端可以统一对待单个对象和组合对象。 | ★★★☆☆ | ★★★★☆ |
| 装饰模式<br>(Decorator Pattern) | 动态地给一个对象增加一些额外的职责。就扩展功能而言，装饰模式提供了一种比使用子类更加灵活的替代方案。 | ★★★☆☆ | ★★★☆☆ |
| 外观模式<br>(Facade Pattern) | 为子系统中的一组接口提供一个统一的入口。外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。 | ★☆☆☆☆ | ★★★★★ |
| 享元模式<br>(Flyweight Pattern) | 运用共享技术有效地支持大量细粒度对象的复用。 | ★★★★☆ | ★☆☆☆☆ |
| 代理模式<br>(Proxy Pattern) | 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。 | ★★★☆☆ | ★★★★☆ |

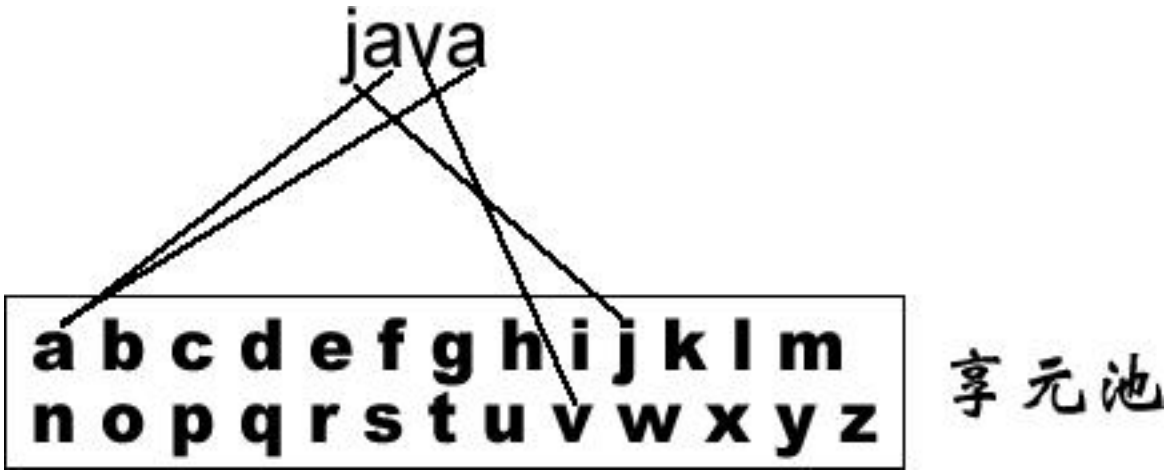
# 享元模式概述

- 动机
  - 如果一个软件系统在运行时所创建的相同或相似对象数量太多，将导致运行代价过高，带来系统资源浪费、性能下降等问题
  - 如何避免系统中出现大量相同或相似的对象，同时又不影响客户端程序通过面向对象的方式对这些对象进行操作呢?

**享元模式**

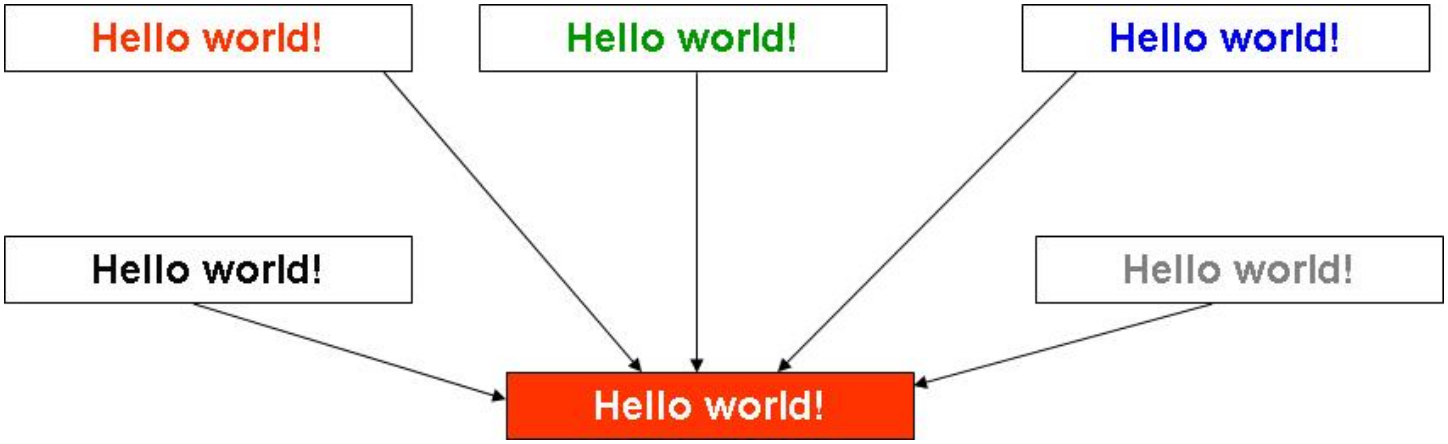
# 享元模式概述

- 字符享元对象示意图



# 享元模式概述

- 分析
  - 享元模式：通过共享技术实现相同或相似对象的重用
  - 享元池(Flyweight Pool)：存储共享实例对象的地方



# 享元模式概述

- 分析
  - 内部状态(Intrinsic State)：存储在享元对象内部并且不会随环境改变而改变的状态，内部状态可以共享（例如：字符的内容）
  - 外部状态(Extrinsic State)：随环境改变而改变的、不可以共享的状态。享元对象的外部状态通常由客户端保存，并在享元对象被创建之后，需要使用的时候再传入到享元对象内部。一个外部状态与另一个外部状态之间是相互独立的（例如：字符的颜色和大小）

# 享元模式概述

- 原理
  - (1) 将具有相同内部状态的对象存储在享元池中，享元池中的对象是可以实现共享的
  - (2) 需要的时候将对象从享元池中取出，即可实现对象的复用
  - (3) 通过向取出的对象注入不同的外部状态，可以得到一系列相似的对象，而这些对象在内存中实际上只存储一份

# 享元模式概述

- 享元模式的定义

**享元模式：**运用共享技术有效地支持大量细粒度对象的复用。

**Flyweight Pattern:** Use **sharing** to support large numbers of fine-grained objects efficiently.

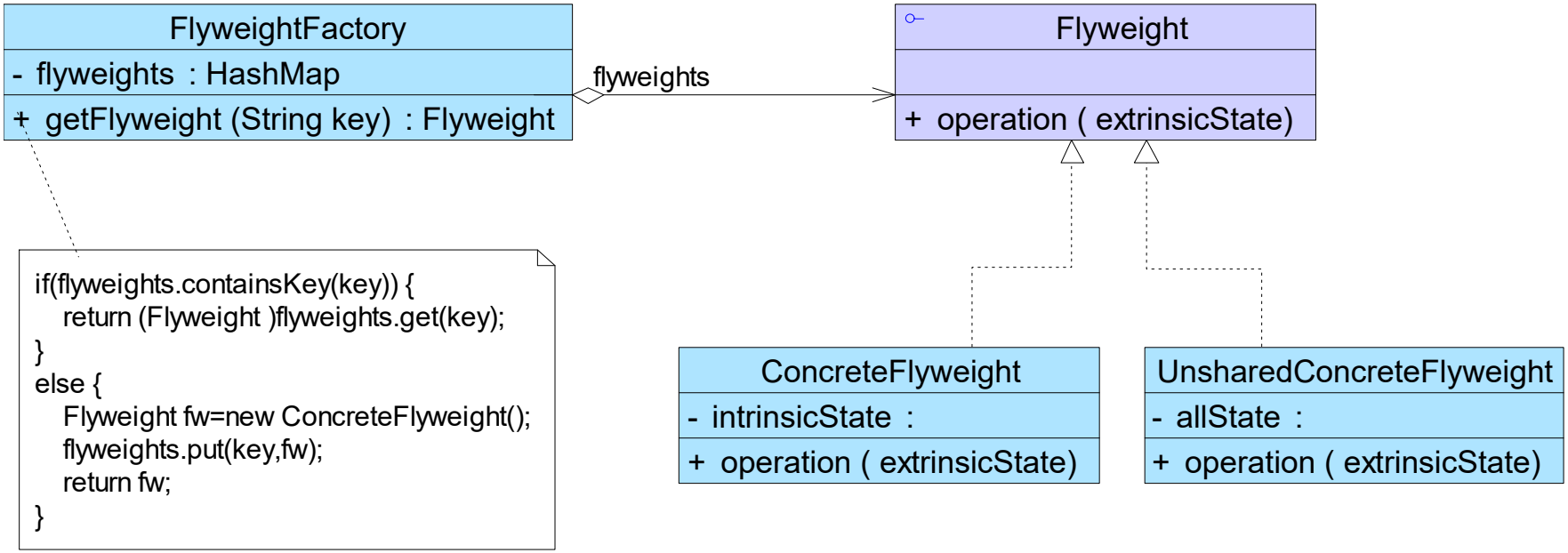
- 对象行为型模式

# 享元模式概述

- 享元模式的定义
  - 又称为轻量级模式
  - 要求能够被共享的对象必须是细粒度对象

# 享元模式的结构与实现

- 享元模式的结构



# 享元模式的结构与实现

- 享元模式的结构
  - 享元模式包含以下4个角色：
    - Flyweight（抽象享元类）
    - ConcreteFlyweight（具体享元类）
    - UnsharedConcreteFlyweight（非共享具体享元类）
    - FlyweightFactory（享元工厂类）

# 享元模式的结构与实现

- 享元模式的实现
  - 典型的抽象享元类代码：

```
public abstract class Flyweight {
    public abstract void operation(String extrinsicState);
}
```

# 享元模式的结构与实现

- 享元模式的实现
  - 典型的具体享元类代码：

```
public class ConcreteFlyweight extends Flyweight {
   //内部状态intrinsicState作为成员变量，同一个享元对象其内部状态是一致的
   private String intrinsicState;
   public ConcreteFlyweight(String intrinsicState) {
      this.intrinsicState = intrinsicState;
   }

   //外部状态extrinsicState在使用时由外部设置，不保存在享元对象中，即使是同一个对象，在每一次调用时可以传入不同的外部状态
   public void operation(String extrinsicState) {
      //实现业务方法
   }
}
```

# 享元模式的结构与实现

- 享元模式的实现
  - 典型的非共享具体享元类代码：

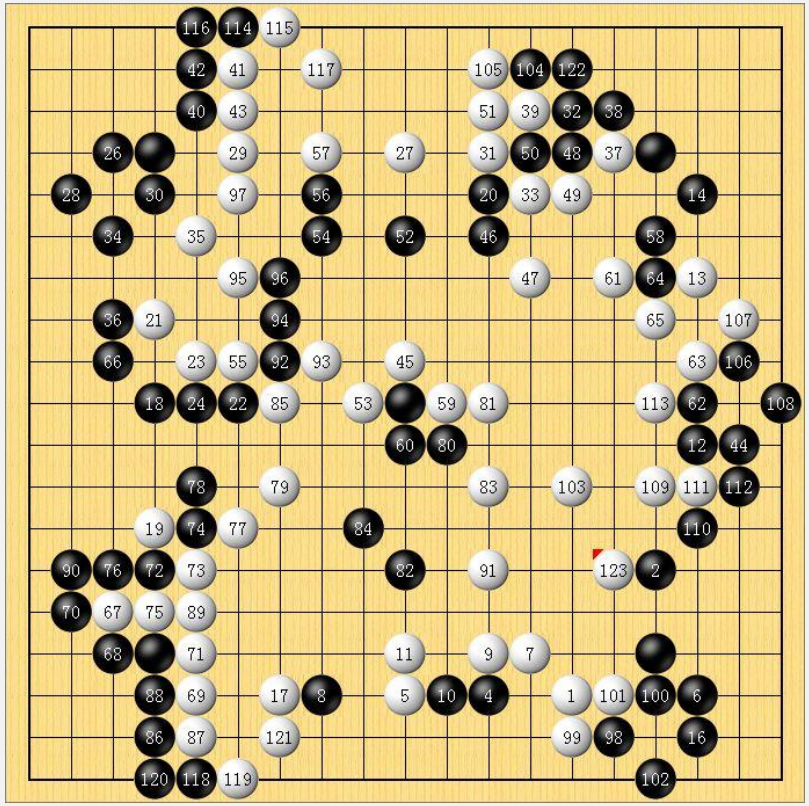
```
public class UnsharedConcreteFlyweight extends Flyweight {
    public void operation(String extrinsicState) {
        //实现业务方法
    }
}
```

# 享元模式的结构与实现

享元模式的实现

```
public class FlyweightFactory {
    //定义一个HashMap用于存储享元对象，实现享元池
    private HashMap flyweights = new HashMap();

    public Flyweight getFlyweight(String key) {
        //如果对象存在，则直接从享元池获取
        if (flyweights.containsKey(key)) {
            return (Flyweight)flyweights.get(key);
        }
        //如果对象不存在，先创建一个新的对象添加到享元池中，然后返回
        else {
            Flyweight fw = new ConcreteFlyweight();
            flyweights.put(key,fw);
            return fw;
        }
    }
}
```

某软件公司要开发一个围棋软件，其界面效果如下图所示：

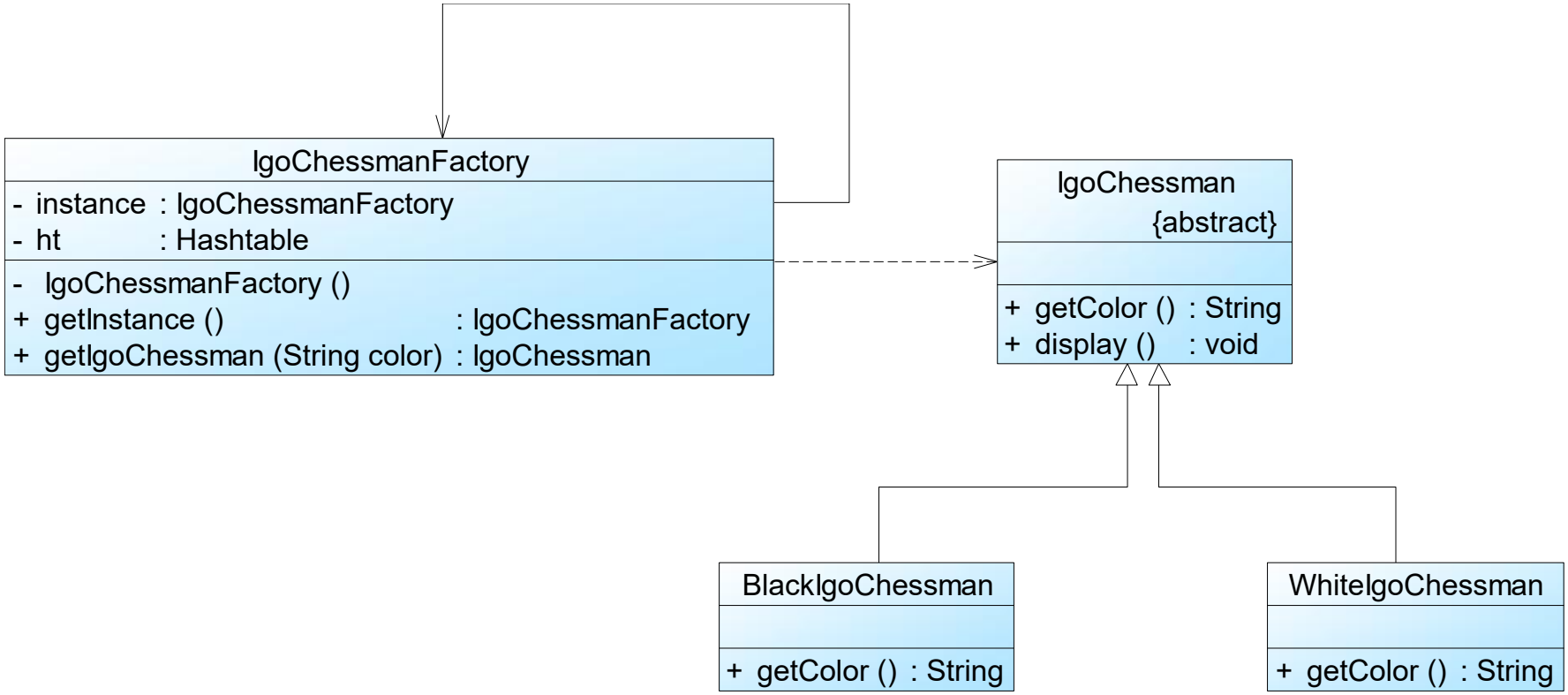


**围棋软件界面效果图**

该软件公司开发人员通过对围棋软件进行分析发现，在图中，围棋棋盘中包含大量的黑子和白子，它们的形状、大小都一模一样，只是出现的位置不同而已。如果将每一个棋子都作为一个独立的对象存储在内存中，将导致该围棋软件在运行时所需内存空间较大，如何降低运行代价、提高系统性能是需要解决的一个问题。为了解决该问题，现使用享元模式来设计该围棋软件的棋子对象。

# 享元模式的应用实例

- 实例类图




围棋棋子结构图

# 享元模式的应用实例

- 实例代码
  - (1) IgoChessman：围棋棋子类，充当抽象享元类
  - (2) BlackIgoChessman：黑色棋子类，充当具体享元类
  - (3) WhiteIgoChessman：白色棋子类，充当具体享元类
  - (4) IgoChessmanFactory：围棋棋子工厂类，充当享元工厂类
  - (5) Client：客户端测试类

演示……

Code (designpatterns.flyweight.simple)

# 享元模式的应用实例

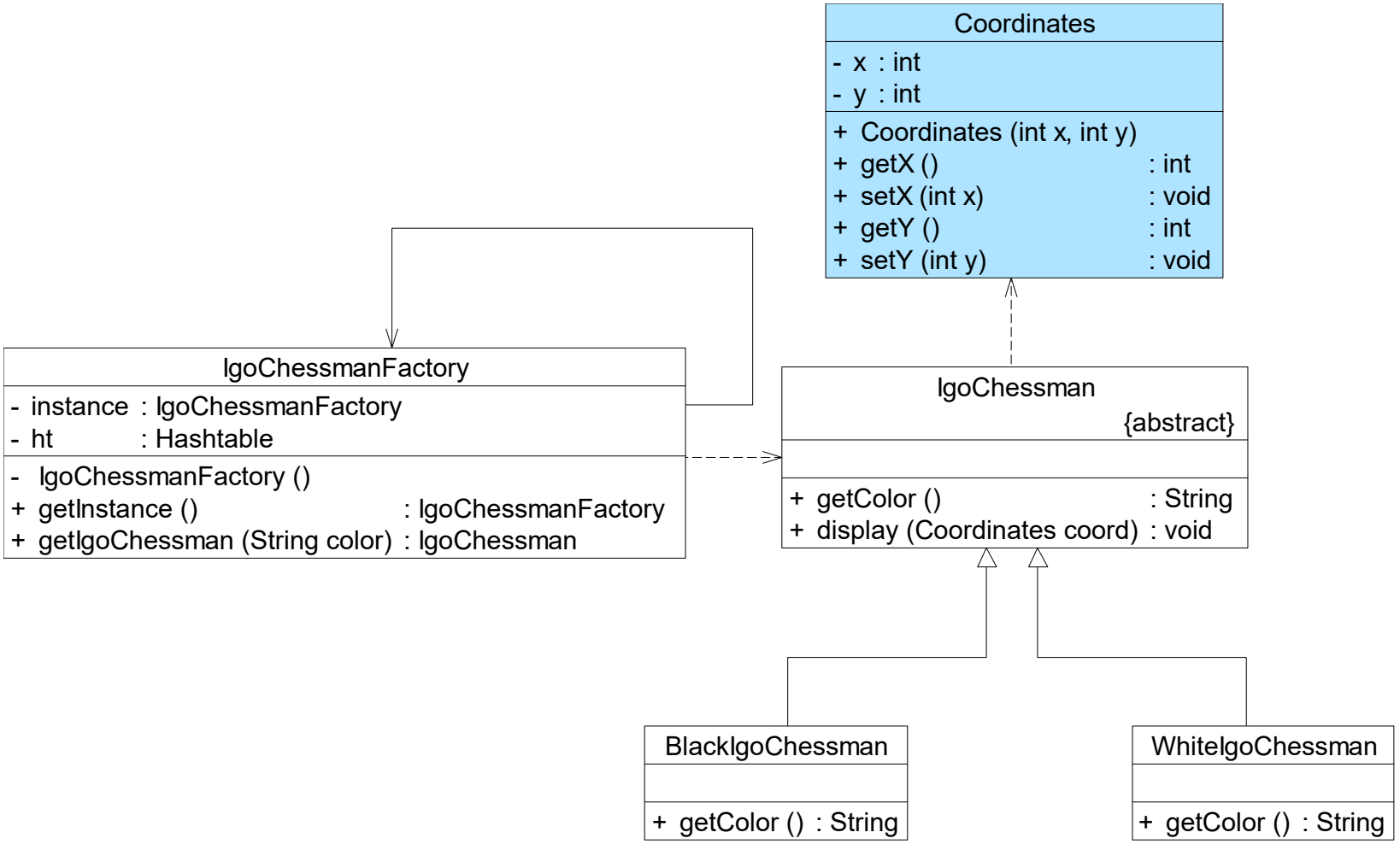
- 结果及分析
  - 在实现享元工厂类时使用了单例模式和简单工厂模式，确保了享元工厂对象的唯一性，并提供了工厂方法向客户端返回享元对象

# 有外部状态的享元模式

- 动机
  - 如何让相同的黑子或者白子能够多次重复显示但位于一个棋盘的不同地方？
  - 解决方案：将棋子的位置定义为棋子的一个外部状态，在需要时再进行设置

# 有外部状态的享元模式

- 结构




引入外部状态之后的围棋棋子结构图

# 有外部状态的享元模式

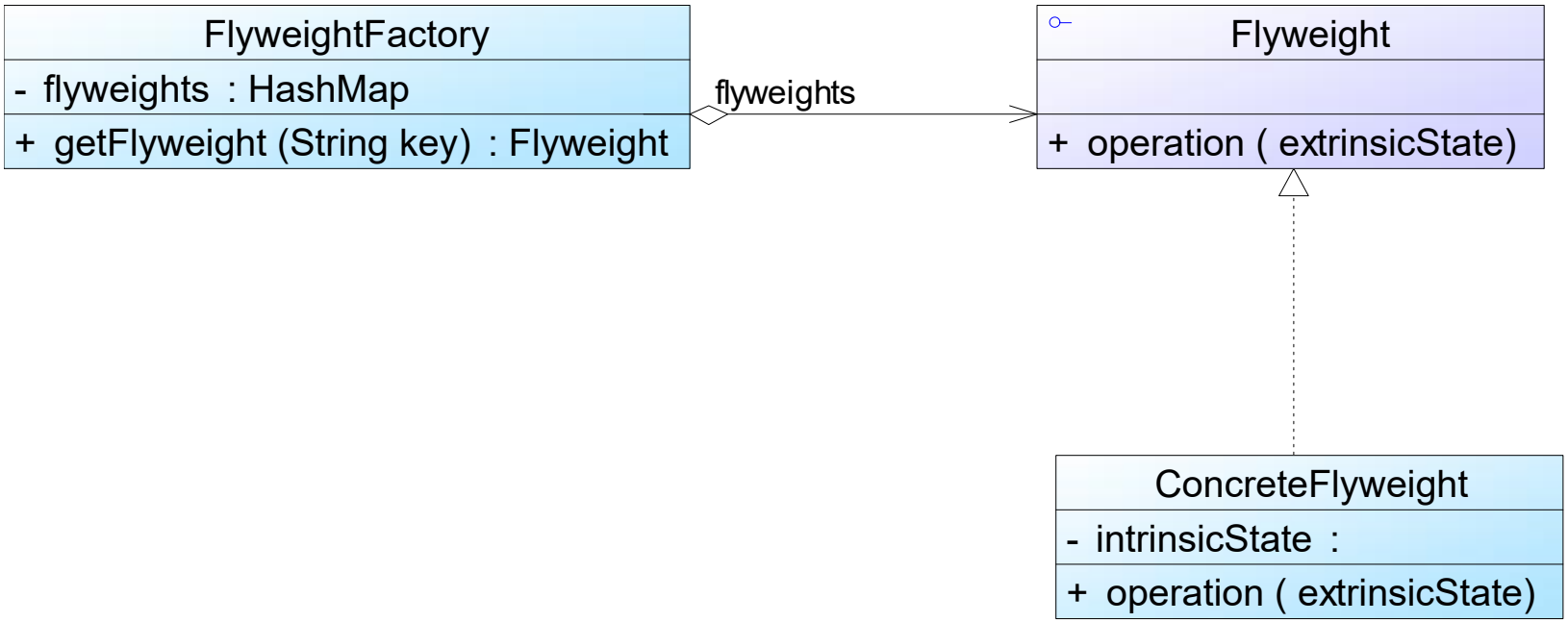
```
package designpatterns.flyweight.extend;
public class Coordinates {
    private int x;
    private int y;
```

```
package designpatterns.flyweight.extend;

//围棋棋子类：抽象享元类
public abstract class IgoChessman {
    public abstract String getColor();

    public void display(Coordinates coord){
        System.out.println("棋子颜色：" + this.getColor() + "，棋子位置：" + coord.getX() + "，" + coord.getY() );
    }
}
```

```
        return this.y;
    }
    public void setY(int y) {
        this.y = y;
    }
}
```

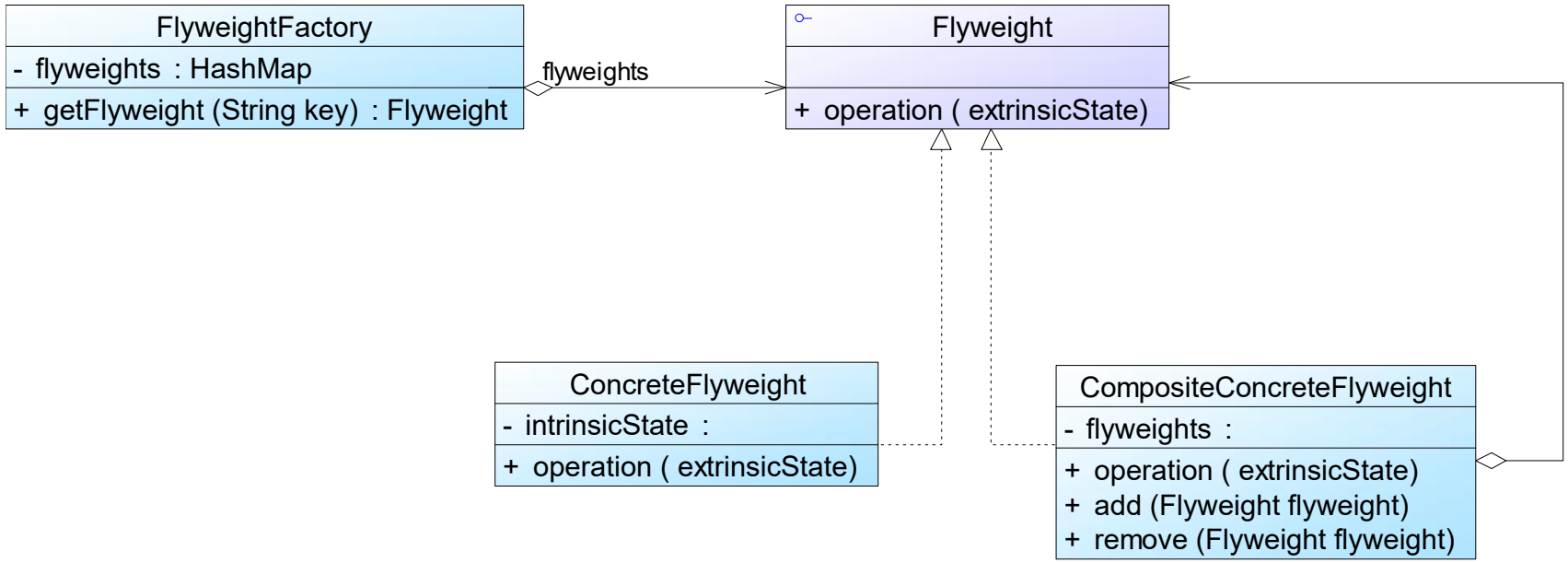
# 单纯享元模式和复合享元模式

- 单纯享元模式
  - 所有的具体享元类都是可以共享的，不存在非共享具体享元类



# 单纯享元模式和复合享元模式

- 复合享元模式
  - 将一些单纯享元对象使用组合模式加以组合
  - 如果希望为多个内部状态不同的享元对象设置相同的外部状态，可以考虑使用复合享元模式



# 享元模式与String类

```
public class Demo {
    public static void main(String args[]) {
        String str1 = "abcd";
        String str2 = "abcd";
        String str3 = "ab" + "cd";
        String str4 = "ab";
        str4 += "cd";

        System.out.println(str1 == str2);
        System.out.println(str1 == str3);
        System.out.println(str1 == str4);

        str2 += "e";
        System.out.println(str1 == str2);
    }
}
```

```
true
true
false
false
```

# 享元模式的优缺点与适用环境

- 模式优点
  - 可以减少内存中对象的数量，使得相同或者相似的对象在内存中只保存一份，从而可以节约系统资源，提高系统性能
  - 外部状态相对独立，而且不会影响其内部状态，从而使得享元对象可以在不同的环境中被共享

# 享元模式的优缺点与适用环境

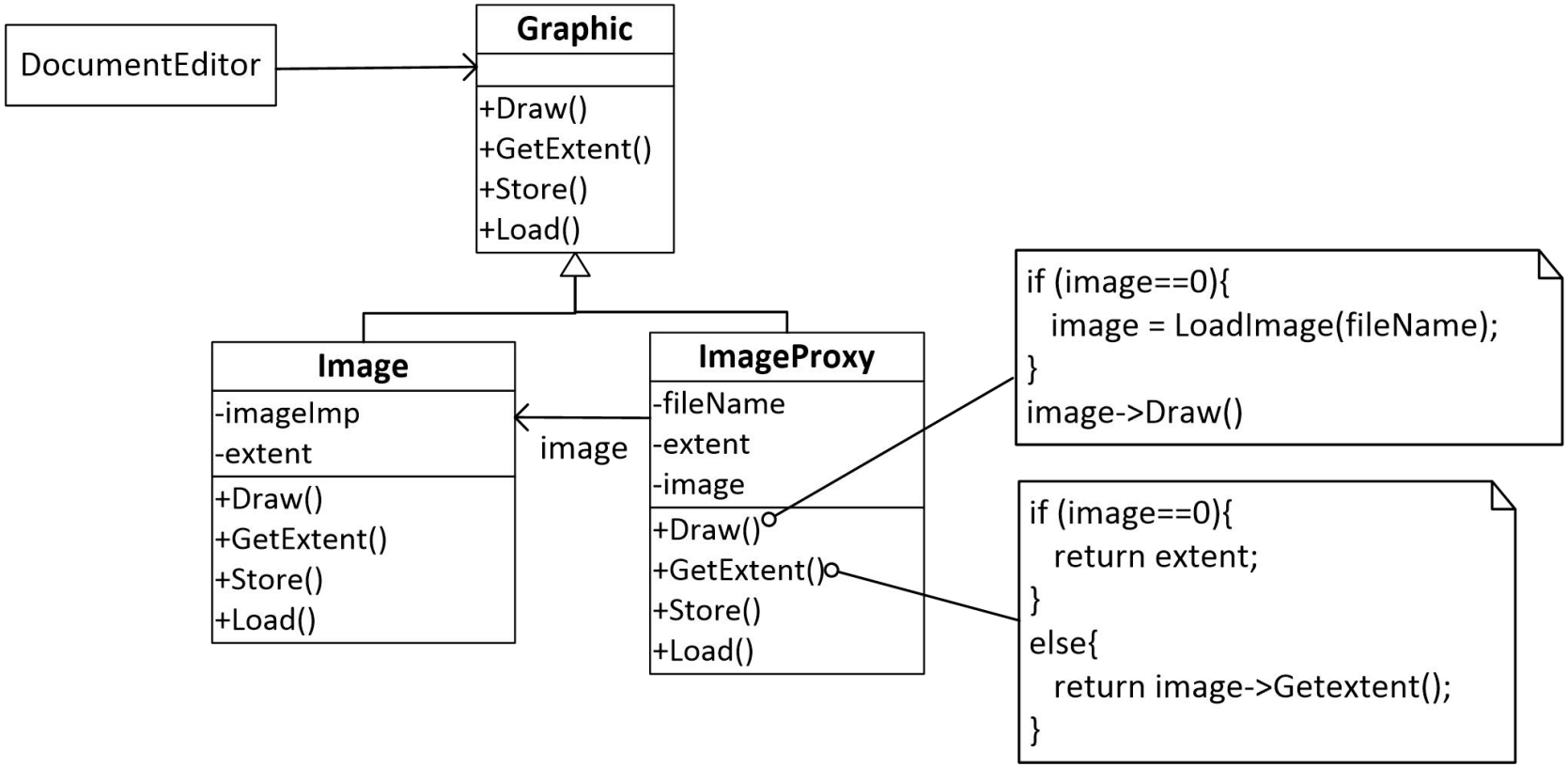
- 模式缺点
  - 使得系统变得复杂，需要分离出内部状态和外部状态，这使得程序的逻辑复杂化
  - 为了使对象可以共享，享元模式需要将享元对象的部分状态外部化，而读取外部状态将使得运行时间变长

# 享元模式的优缺点与适用环境

- 模式适用环境
  - 一个系统有大量相同或者相似的对象，造成内存的大量耗费
  - 对象的大部分状态都可以外部化，可以将这些外部状态传入对象中
  - 在使用享元模式时需要维护一个存储享元对象的享元池，而这需要耗费一定的系统资源，因此，在需要多次重复使用享元对象时才值得使用享元模式

# 代理模式概述

- 图形图像绘制软件



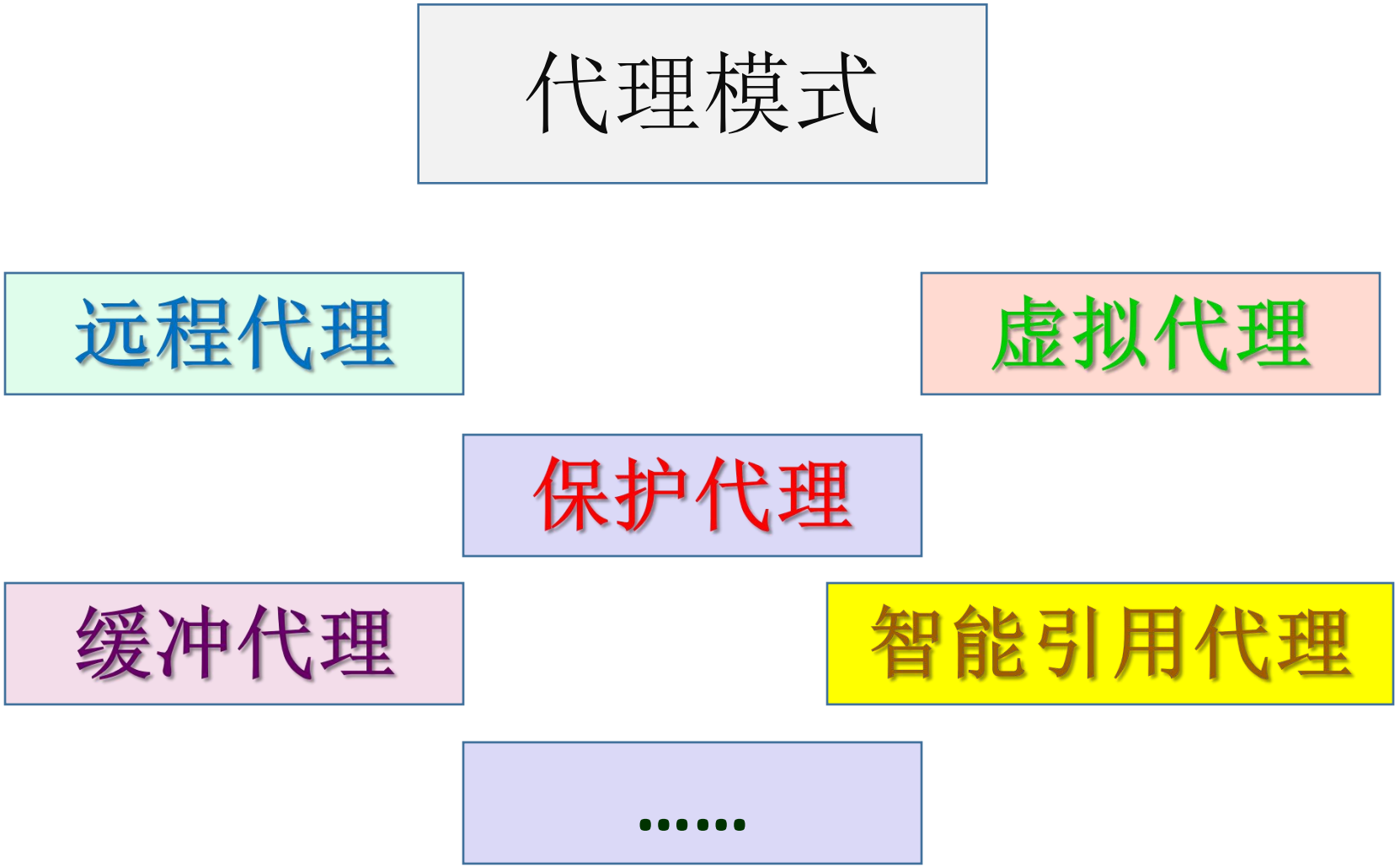

图像对象创建的内存消耗很大；
图形图像绘制软件中，只有当文档编辑器激活图像Draw功能的时候才需要加载图像

# 代理模式概述

- 分析
  - 软件开发：客户端 → 代理对象 → 真实对象

# 代理模式概述

- 类型



# 代理模式概述

- 代理模式的定义

**代理模式：**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。

**Proxy Pattern:** Provide **a surrogate or placeholder** for another object to control access to it.

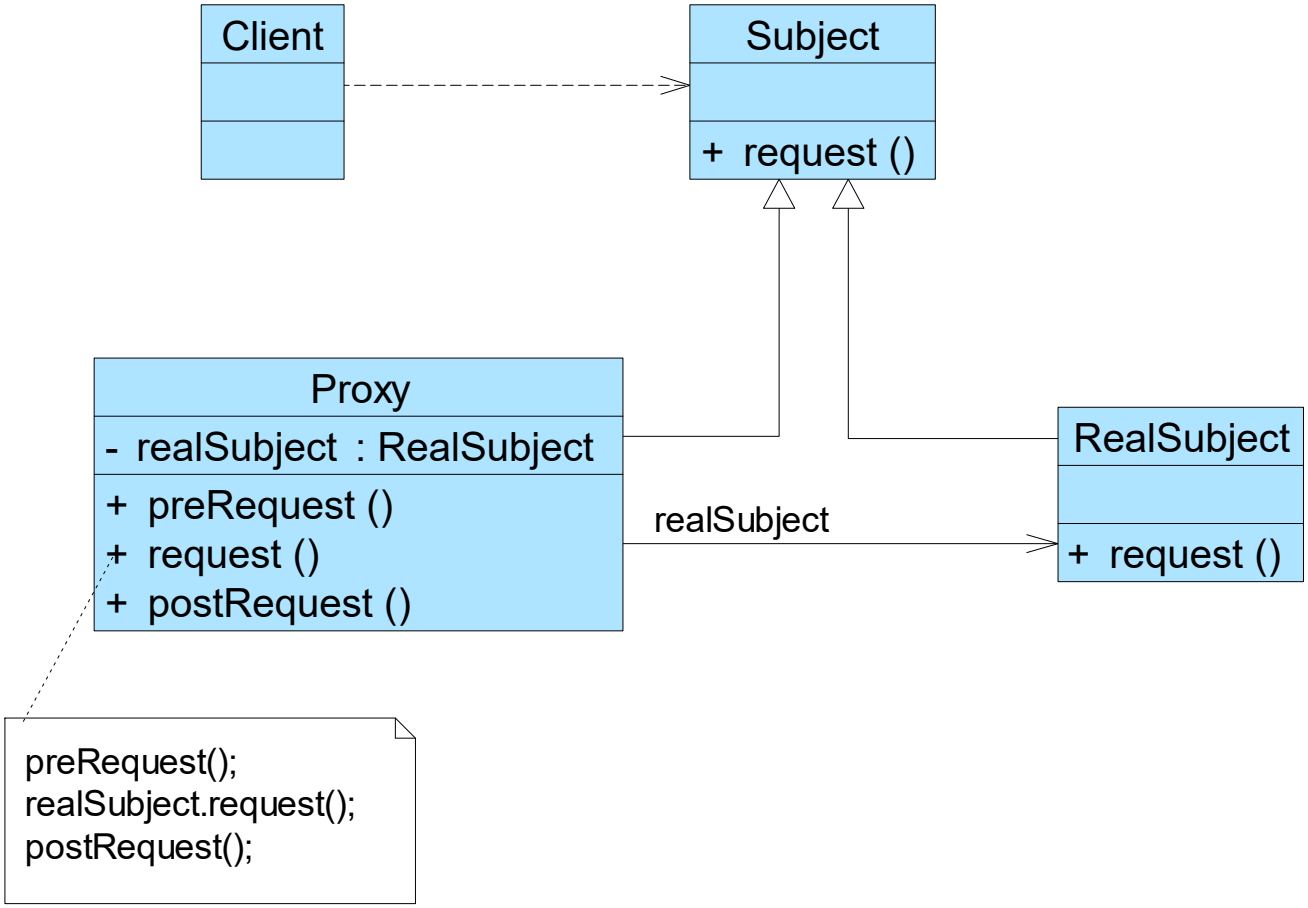
- 对象结构型模式

# 代理模式概述

- 代理模式的定义
  - 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之间起到中介的作用
  - 去掉客户不能看到的内容和服务或者增添客户需要的额外的新服务

# 代理模式的结构与实现

- 代理模式的结构



# 代理模式的结构与实现

- 代理模式的结构
  - 代理模式包含以下3个角色：
    - Subject（抽象主题角色）
    - Proxy（代理主题角色）
    - RealSubject（真实主题角色）

# 代理模式的结构与实现

- 代理模式的实现
  - 抽象主题类典型代码：

```
public abstract class Subject {
    public abstract void request();
}
```

# 代理模式的结构与实现

- 代理模式的实现
  - 真实主题类典型代码：

```
public class RealSubject extends Subject{
    public void request() {
        //业务方法具体实现代码
    }
}
```

# 代理模式的结构与实现

- 代理模式的实现

```
public class Proxy extends Subject {
    private RealSubject realSubject = new RealSubject(); //维持一个对真实主题对象的引用
    public void preRequest() {
        ……
    }

    public void request() {
        preRequest();
        realSubject.request(); //调用真实主题对象的方法
        postRequest();
    }

    public void postRequest() {
        ……
    }
}
```

# 代理模式的结构与实现

- 几种常见的代理模式
  - 远程代理(Remote Proxy)：为一个位于不同的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)
  - 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，真实对象只在需要时才会被真正创建

# 代理模式的结构与实现

- 几种常见的代理模式
  - 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
  - 缓冲代理(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
  - 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

# 代理模式的应用实例

- 实例说明

某软件公司承接了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，只有合法用户才能够使用该查询系统；

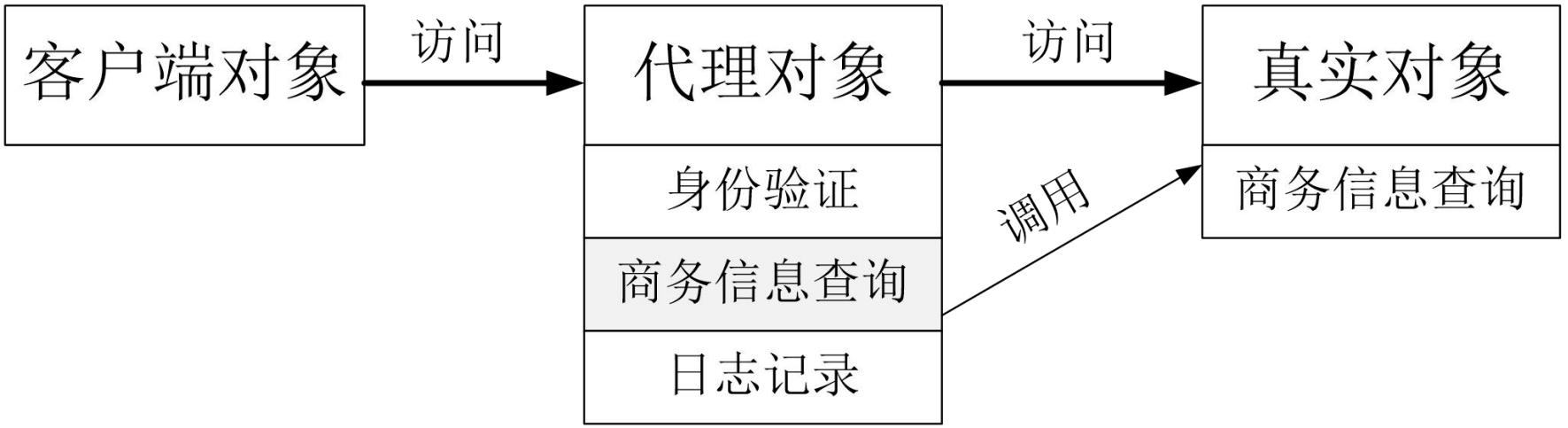
(2) 在进行商务信息查询时系统需要记录查询日志，以便根据查询次数收取查询费用。

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信息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。

现使用代理模式设计并实现该收费商务信息查询系统。

# 代理模式的应用实例

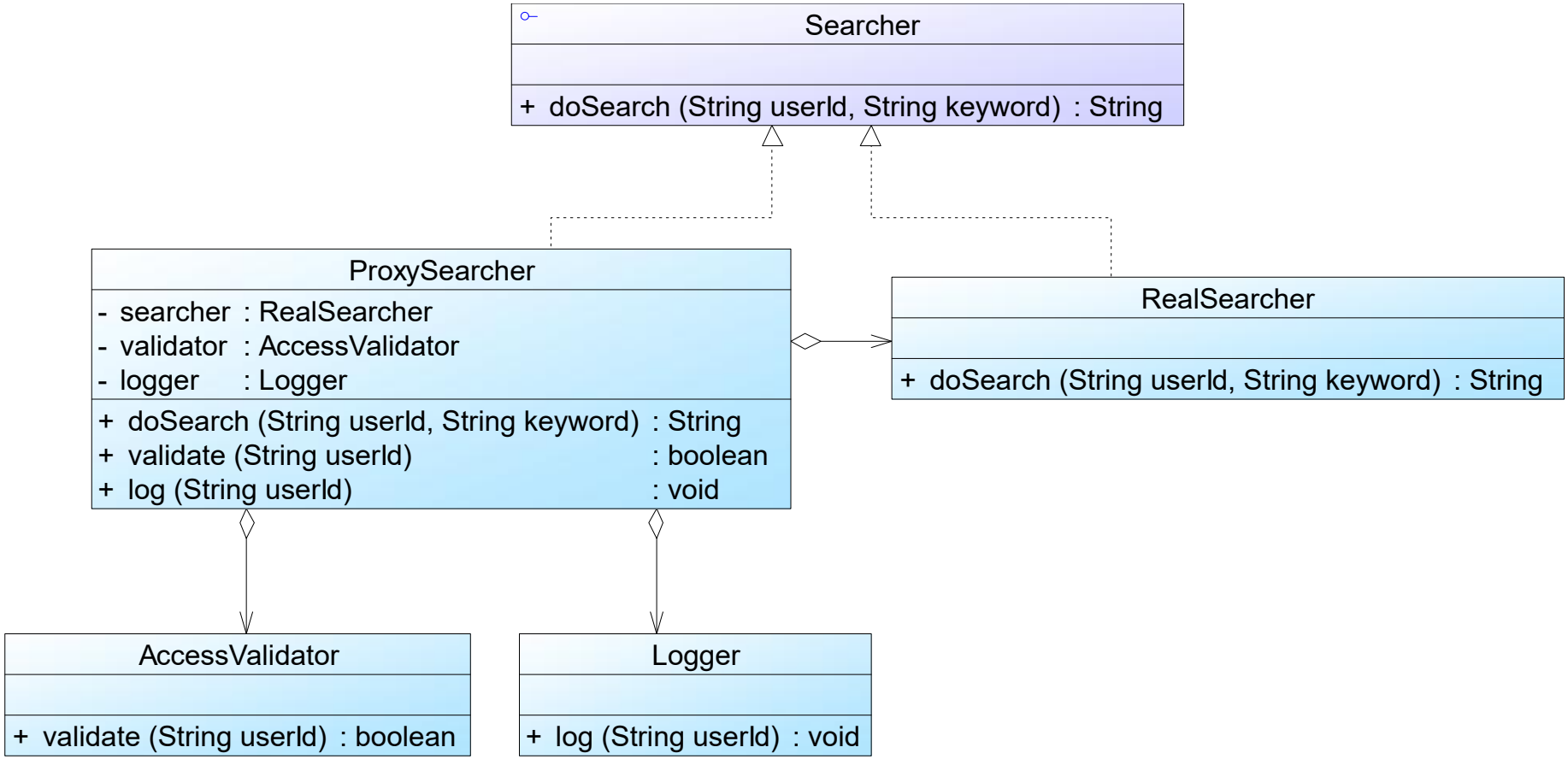
- 实例分析及类图




商务信息查询系统设计方案示意图

# 代理模式的应用实例

- 实例分析及类图




商务信息查询系统结构图

# 代理模式的应用实例

- 实例代码
  - (1) AccessValidator：身份验证类，业务类
  - (2) Logger：日志记录类，业务类
  - (3) Searcher：抽象查询类，充当抽象主题角色
  - (4) RealSearcher：具体查询类，充当真实主题角色
  - (5) ProxySearcher：代理查询类，充当代理主题角色
  - (6) Client：客户端测试类

演示……

Code (designpatterns.proxy)

# 代理模式的应用实例

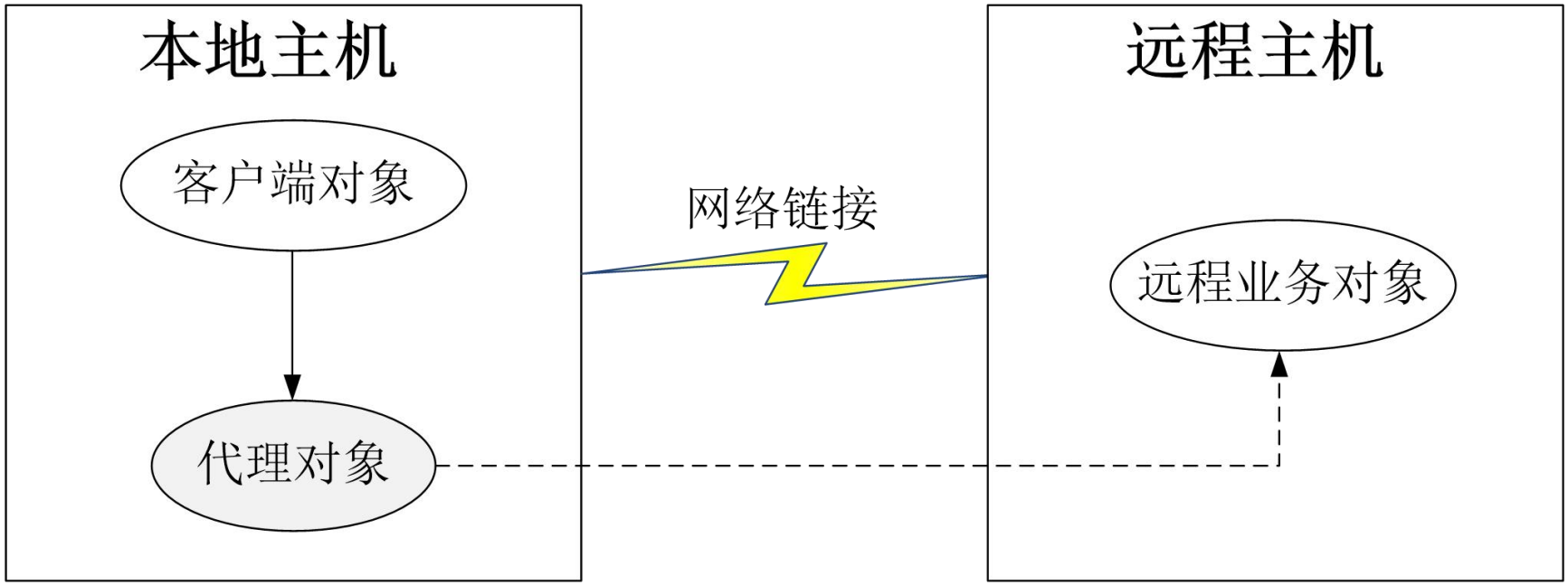
- 结果及分析
  - 保护代理和智能引用代理
  - 在代理类ProxySearcher中实现对真实主题类的权限控制和引用计数

# 远程代理

- 动机
  - 客户端程序可以访问在远程主机上的对象，远程主机可能具有更好的计算性能与处理速度，可以快速地响应并处理客户端的请求
  - 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

# 远程代理

- 结构




**Java RMI (Remote Method Invocation)**

# 虚拟代理

- 动机
  - 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，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
  - 使用一个“虚假”的代理对象来代表真实对象，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性能

# 虚拟代理

- 应用
  - 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（结合多线程技术）
  - 一个对象的加载十分耗费系统资源，让那些占用大量内存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（用时间换取空间）

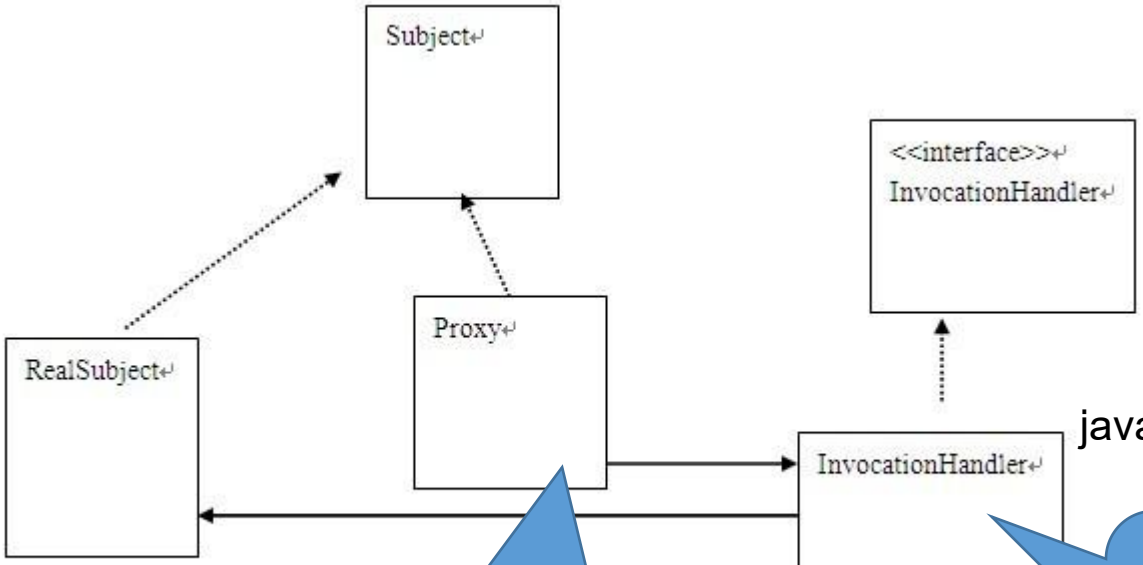
# Java动态代理

- 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
- Java语言提供了对动态代理的支持，Java语言实现动态代理时需要用到位于java.lang.reflect包中的一些类

# Java动态代理



- Java Dynamic Proxy




java.lang.reflect.InvocationHandler

**Java** 动态代理机制的主类，提供了一组静态方法来为一组接口动态地生成代理类及其对象

调用处理器接口，它自定义了一个 **invoke** 方法，用于集中处理在动态代理类对象上的方法调用，通常在该方法中实现对委托类的代理访问

java.lang.reflect.Proxy

# Java动态代理

- Proxy类
  - public static Class<?> getProxyClass(ClassLoader loader, Class<?>... interfaces)：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（与真实主题类的接口列表一致）
  - public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)：该方法用于返回一个动态创建的代理类的实例，方法中第一个参数loader表示代理类的类加载器，第二个参数interfaces表示代理类所实现的接口列表（与真实主题类的接口列表一致），第三个参数h表示所指派的调用处理程序类

# Java动态代理

- 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提供一个相关的具体调用处理者（InvocationHandler接口的子类）
  - 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。invoke()方法包含三个参数，其中第一个参数proxy表示代理类的实例，第二个参数method表示需要代理的方法，第三个参数args表示代理方法的参数数组

# Java动态代理



- 简化的动态代理对象创建过程：

```
// InvocationHandlerImpl 实现了 InvocationHandler 接口，并能实现方法调用从
//代理类到委托类的分派转发
InvocationHandler handler = new InvocationHandlerImpl(..);

// 通过 Proxy 直接创建动态代理类实例
Interface proxy = (Interface)Proxy.newProxyInstance( classLoader,
    new Class[] { Interface.class }, handler );
```

# Java动态代理

- 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。

# Java动态代理

- 动态代理实例

某软件公司欲为公司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。

# Java动态代理

- 实例代码
  - (1) AbstractUserDAO：抽象用户DAO类，抽象主题角色
  - (2) AbstractDocumentDAO：抽象文档DAO类，抽象主题角色
  - (3) UserDAO：用户DAO类，具体主题角色
  - (4) DocumentDAO：文档DAO类，具体主题角色
  - (5) DAOLogHandler：自定义请求处理程序类
  - (6) Client：客户端测试类

演示……

Code (designpatterns.proxy.dynamic)

# 代理模式的优缺点与适用环境

- 模式优点
  - 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
  - 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

# 代理模式的优缺点与适用环境

- 模式优点——逐个分析
  - 远程代理：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
  - 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
  - 缓冲代理：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
  - 保护代理：可以控制对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

# 代理模式的优缺点与适用环境

- 模式缺点
  - 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）
  - 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）

# 代理模式的优缺点与适用环境

- 模式适用环境
  - 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
  - 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
  - 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使用缓冲代理
  - 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
  - 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

# 相关模式



- Adapter模式中适配器为它所适配的对象提供了一个不同的接口，而Proxy则提供了与它的实体相同的接口，或其接口的子集(Protection Proxy可能会拒绝执行实体的操作)
- Decorator的实现和Proxy类似，但目的不同：Decorator对象添加一个或多个功能，而Proxy则控制对对象的访问