第14章 外观模式 ★

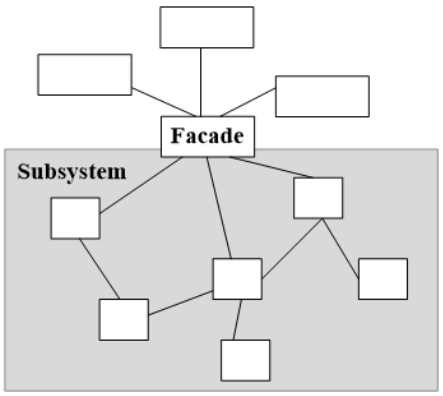
模式动机

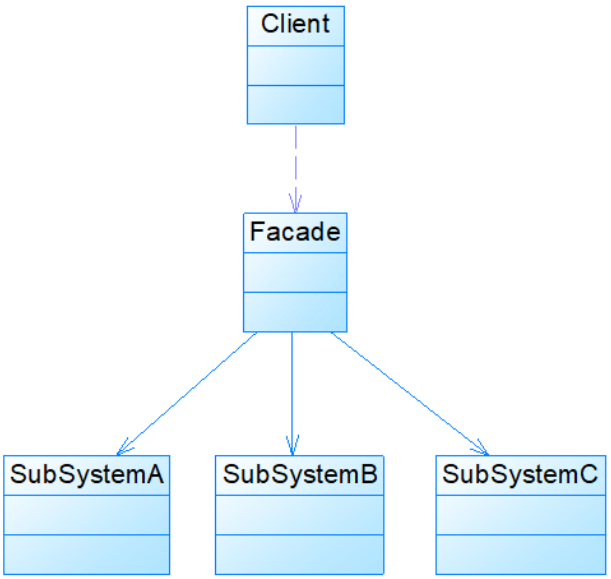
- ✓ 一个客户类需要和多个业务类交互，有时候这些需要交互的业务类会作为一个整体出现
- ✓ 引入一个新的外观类(Facade)来负责和多个业务类【子系统(Subsystem)】进行交互，而客户类只需与外观类交互
- ✓ 为多个业务类的调用提供了一个统一的入口，简化了类与类之间的交互
- ✓ 没有外观类：每个客户类需要和多个子系统之间进行复杂的交互，系统的耦合度将很大
- ✓ 引入外观类：客户类只需要直接与外观类交互，客户类与子系统之间原有的复杂引用关系由外观类来实现，从而降低了系统的耦合度

模式定义

- ✓ 外观模式(Facade Pattern)：外部与子系统的通信通过一个统一的外观对象进行，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，这个接口使得子系统更加容易使用
- ✓ 外观模式又称为门面模式，它是一种对象结构型模式

模式结构





外观模式包含如下角色：

Facade：外观角色

SubSystem:子系统角色

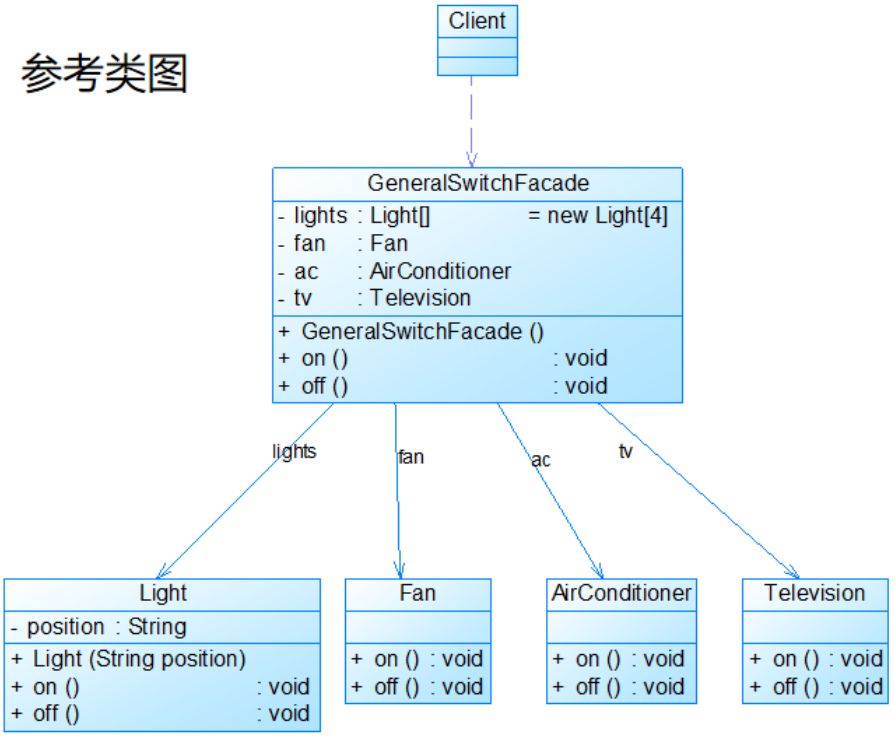
外观类示例代码：

```
public class Facade {
    private SubSystemA obj1 = new SubSystemA();
    private SubSystemB obj2 = new SubSystemB();
    private SubSystemC obj3 = new SubSystemC();

    public void method() {
        obj1.method();
        obj2.method();
        obj3.method();
    }
}
```

模式实例

✓ 电源总开关：参考类图



外观模式优点：

1）它对客户端屏蔽了子系统组件，减少了客户端所需处理的对象数目，并使得子系统使用起来更加容易

2）它实现了子系统与客户端之间的松耦合关系，这使得子系统的变化不会影响到调用它的客户端，只需要调整外观类即可

3）一个子系统的修改对其他子系统没有任何影响，而且子系统的内部变化也不会影响到外观对象

外观模式缺点：

1）不能很好地限制客户端直接使用子系统类，如果对客户端访问子系统类做太多的限制则减少了可变性和灵活性

2）如果设计不当，增加新的子系统可能需要修改外观类的源代码，违背了开闭原则

在以下情况下可以使用外观模式：

1）要为访问一系列复杂的子系统提供一个简单入口

2）客户端程序与多个子系统之间存在很大的依赖性

3）在层次化结构中，可以使用外观模式定义系统中每一层的入口，层与层之间不直接产生联系，而是通过外观类建立联系，降低层之间的耦合度

第16章 代理模式

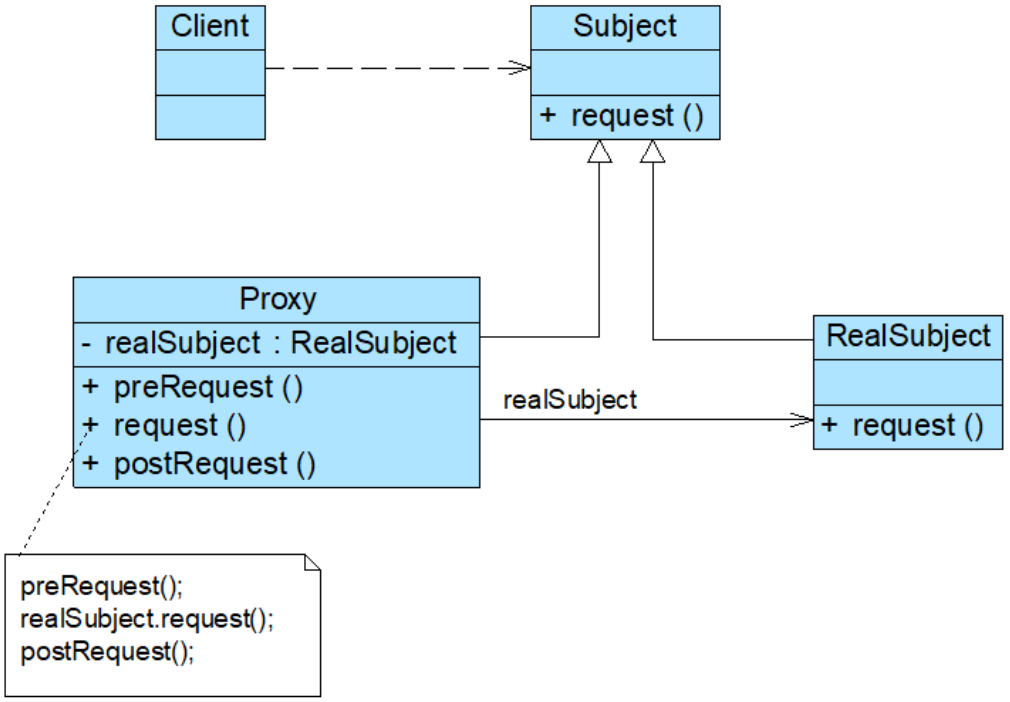
模式动机

- ✓ 通过引入一个新的对象（如小图片和远程代理对象）来实现对真实对象的操作，或者将新的对象作为真实对象的一个替身
- ✓ 引入代理对象来间接访问一个对象 → **代理模式**

模式定义

- ✓ 代理模式(Proxy Pattern)：给某一个对象提供一个代理，并由代理对象控制对原对象的引用
- ✓ 对象结构型模式
- ✓ 代理对象可以在客户端和目标对象之间起到中介的作用
- ✓ 通过代理对象去掉客户不能看到的内容和服务或者添加客户需要的额外的新服务

模式结构



代理模式包含如下角色：

Subject：抽象主题角色

Proxy：代理主题角色

RealSubject：真实主题角色

代理类示例代码：

```
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() {
      ……
   }
}
```

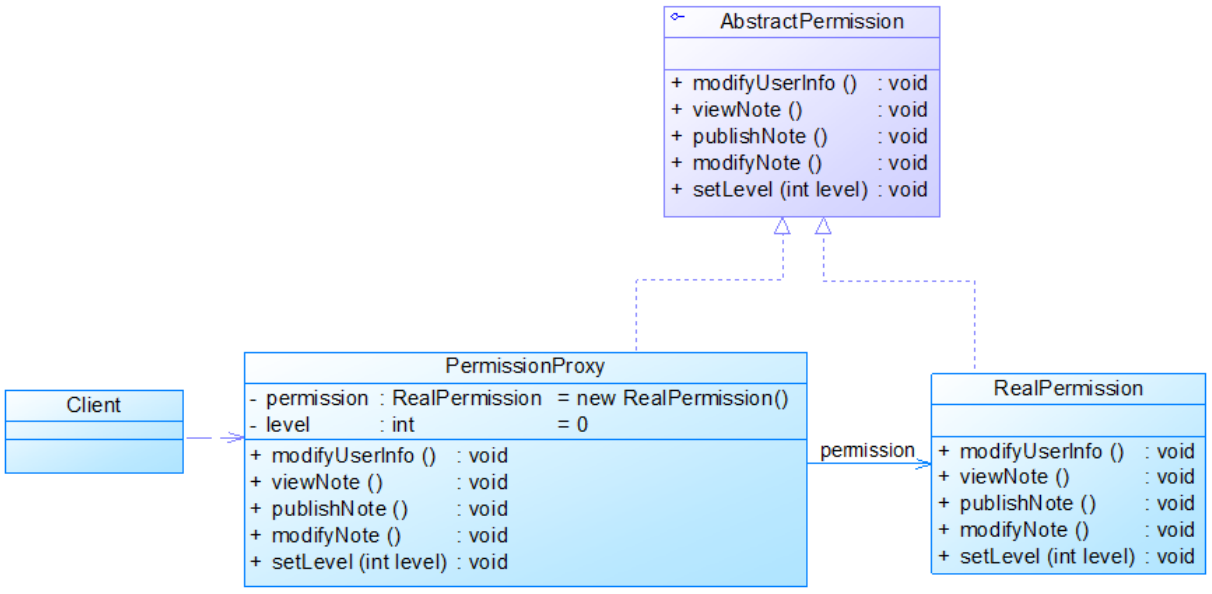
几种常用的代理模式

- ✓ 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
- ✓ 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

论坛权限控制代理：实例说明

- 在一个论坛中已注册用户和游客的权限不同，已注册的用户拥有发帖、修改自己的注册信息、修改自己的帖子等功能；而游客只能看到别人发的帖子，没有其他权限。使用代理模式来设计该权限管理模块。
- 在本实例中我们使用代理模式中的保护代理，该代理用于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。

论坛权限控制代理：参考类图



代理模式优点：

- ✓ 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
- ✓ 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**）

在以下情况下可以使用代理模式：

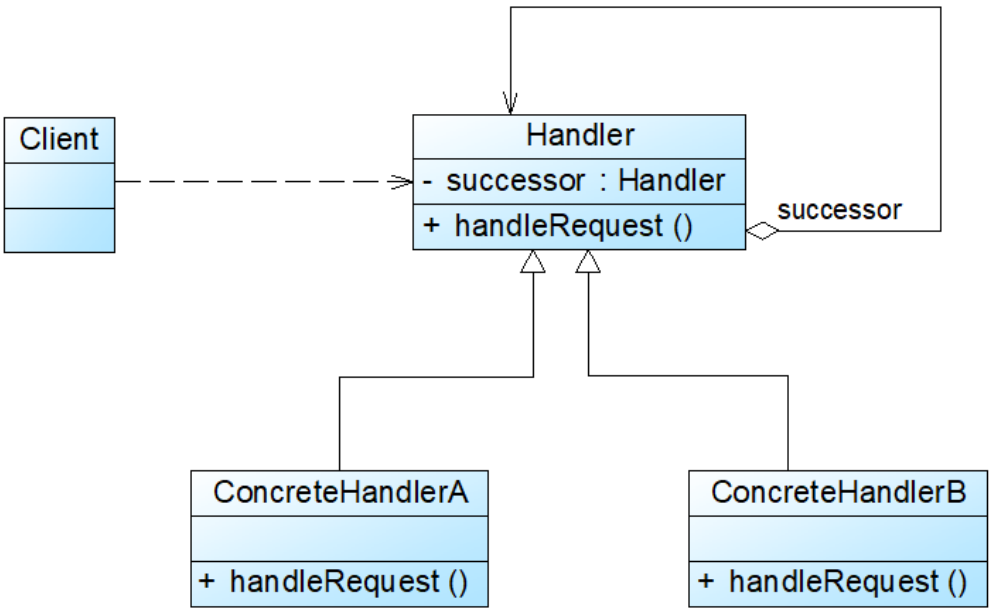
- ✓ 当客户端对象需要访问远程主机中的对象时可以使用远程代理
- ✓ 当需要用一个消耗资源较少的对象来代表一个消耗资源较多的对象，从而降低系统开销、缩短运行时间时可以使用虚拟代理
- ✓ 当需要为某一个被频繁访问的操作结果提供一个临时存储空间，以供多个客户端共享访问这些结果时可以使用缓冲代理
- ✓ 当需要控制对一个对象的访问，为不同用户提供不同级别的访问权限时可以使用保护代理
- ✓ 当需要为一个对象的访问（引用）提供一些额外的操作时可以使用智能引用代理

第17章 责任链模式

模式定义

- ✓ 职责链模式(Chain of Responsibility Pattern)：避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求。将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。
- ✓ 对象行为型模式

模式结构



模式结构

- ✓ 职责链模式包含如下角色：
  - Handler: 抽象处理者
  - ConcreteHandler: 具体处理者

模式分析

- ✓ 将请求的处理者组织成一条链，并让请求沿着链传递，由链上的处理者对请求进行相应的处理
- ✓ 客户端无须关心请求的处理细节以及请求的传递，只需将请求发送到链上，将请求的发送者和请求的处理者解耦

职责链模式优点：

- ✓ 使得一个对象无须知道是其他哪一个对象处理其请求，降低了系统的耦合度
- ✓ 可简化对象之间的相互连接
- ✓ 给对象职责的分配带来更多的灵活性
- ✓ 增加一个新的具体请求处理者时无须修改原有系统的代码，只需要在客户端重新建链即可

职责链模式缺点：

- ✓ 不能保证请求一定会被处理
- ✓ 对于比较长的职责链，系统性能将受到一定影响，在进行代码调试时不太方便
- ✓ 如果建链不当，可能会造成循环调用，将导致系统陷入死循环

在以下情况下可以使用职责链模式：

- ✓ 有多个对象可以处理同一个请求，具体哪个对象处理该请求待运行时刻再确定
- ✓ 在不明确指定接收者的情况下，向多个对象中的一个提交一个请求
- ✓ 可动态指定一组对象处理请求

第18章 命令模式

模式动机

- ✓ 现实生活
  - 相同的开关可以通过不同的电线来控制不同的电器
  - 开关 ← → 请求发送者
  - 电灯 ← → 请求的最终接收者和处理者
  - 开关和电灯之间并不存在直接耦合关系，它们通过电线连接在一起，使用不同的电线可以连接不同的请求接收者

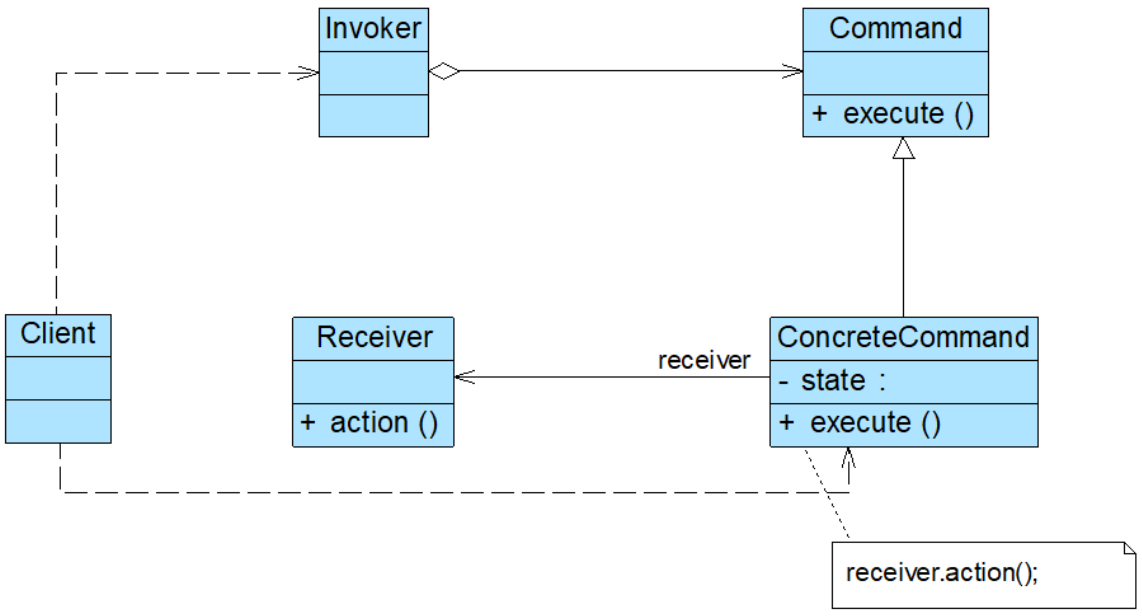
模式动机

- ✓ 软件开发
  - 按钮 ← → 请求发送者
  - 事件处理类 ← → 请求的最终接收者和处理者
  - 发送者与接收者之间引入了新的命令对象（类似电线），将发送者的请求封装在命令对象中，再通过命令对象来调用接收者的方法
  - 相同的按钮可以对应不同的事件处理类

模式定义

- ✓ 命令模式(Command Pattern)：将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。
- ✓ 命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式

模式结构



命令模式包含如下角色：

Command：抽象命令类

ConcreteCommand：具体命令类

Invoker：调用者

Receiver：接收者

调用者（请求发送者）示例代码：

```
public class Invoker {
    private Command command;

    //构造注入
    public Invoker(Command command) {
        this.command = command;
    }

    //设值注入
    public void setCommand(Command command) {
        this.command = command;
    }

    //业务方法，用于调用命令类的execute()方法
    public void call() {
        command.execute();
    }
}
```

具体命令类示例代码：

```
public class ConcreteCommand extends Command {
    private Receiver receiver; //维持一个对请求接收者对象的引用

    public void execute() {
        receiver.action(); //调用请求接收者的业务处理方法action()
    }
}
```

命令模式优点：

1）降低系统的耦合度

2）新的命令可以很容易地加入到系统中，符合开闭原则

3）可以比较容易地设计一个命令队列或宏命令（组合命令）

4）为请求的撤销(Undo)和恢复(Redo)操作提供了一种设计和实现方案

命令模式缺点：

1）使用命令模式可能会导致某些系统有过多的具体命令类（针对每一个对请求接收者的调用操作都需要设计一个具体命令类）

在以下情况下可以使用命令模式：

1）需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互

2）需要在不同的时间指定请求、将请求排队和执行请求

3）需要支持命令的撤销(Undo)操作和恢复(Redo)操作

4）需要将一组操作组合在一起形成宏命令

第20章 迭代器模式 ★

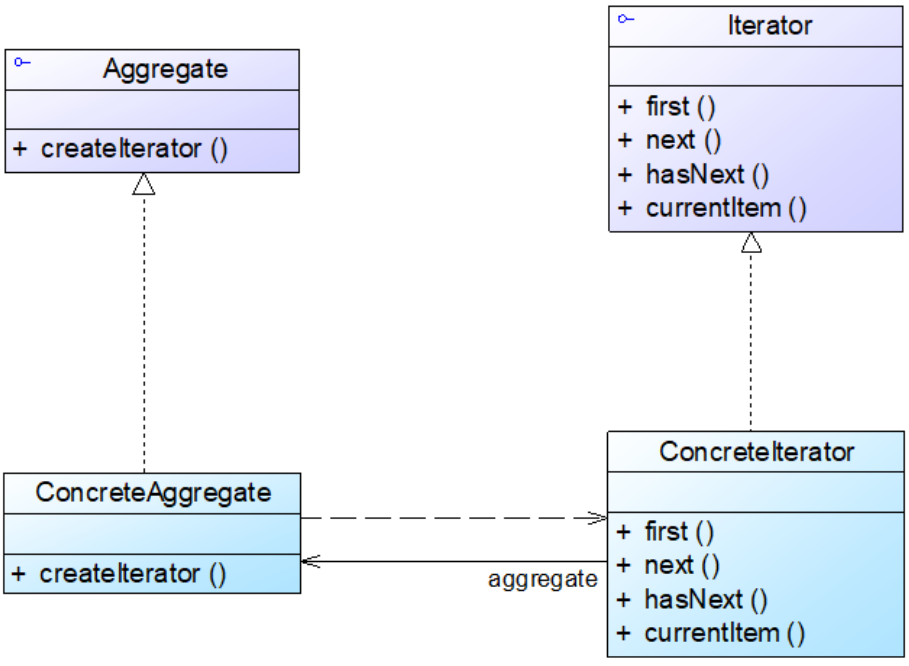
模式动机

- ✓ 电视机 ← → 存储电视频道的集合 ← → 聚合类(Aggregate Classes)
- ✓ 电视机遥控器 ← → 操作电视频道 ← → 迭代器(Iterator)
- ✓ 访问一个聚合对象中的元素但又不需要暴露它的内部结构 ← → 迭代器模式

模式定义

- ✓ 迭代器模式(Iterator Pattern)：提供一种方法来访问聚合对象，而不用暴露这个对象的内部表示。
- ✓ 其别名为游标(Cursor)
- ✓ 迭代器模式是一种对象行为型模式

模式结构



迭代器模式包含如下角色：

Iterator：抽象迭代器

ConcreteIterator：具体迭代器

Aggregate：抽象聚合类

ConcreteAggregate：具体聚合类

模式分析

- ✓ 聚合对象的两个职责：
  - 存储数据，聚合对象的基本职责
  - 遍历数据，既是可变化的，又是可分离的
- ✓ 将遍历数据的行为从聚合对象中分离出来，封装在迭代器对象中
- ✓ 由迭代器来提供遍历聚合对象内部数据的行为，简化聚合对象的设计，更符合单一职责原则

抽象迭代器示例代码：

```
public interface Iterator {
   public void first();              //将游标指向第一个元素
   public void next();               //将游标指向下一个元素
   public boolean hasNext();         //判断是否存在下一个元素
   public Object currentItem();      //获取游标指向的当前元素
}
```

具体迭代器示例代码：

```
public class ConcreteIterator implements Iterator {
   private ConcreteAggregate objects; //维持一个对具体聚合对象的引用，以便于访问存储在聚合对象中的数据
   private int cursor; //定义一个游标，用于记录当前访问位置
   public ConcreteIterator(ConcreteAggregate objects) {
      this.objects=objects;
   }

   public void first() { ..... }

   public void next() { ..... }

   public boolean hasNext() { ..... }

   public Object currentItem() { ..... }
}
```

抽象聚合类示例代码：

```
public interface Aggregate {
    Iterator createIterator();
}
```

具体聚合类示例代码：

```
public class ConcreteAggregate implements Aggregate {
    ......
    public Iterator createIterator() {
        return new ConcreteIterator(this);
    }
    ......
}
```

模式分析

✓ JDK内置迭代器

- java.util.Collection

```
package java.util;

public interface Collection<E> extends
Iterable<E> {
    ......
    boolean add(Object c);
    boolean addAll(Collection c);
    boolean remove(Object o);
    boolean removeAll(Collection c);
    boolean remainAll(Collection c);
    Iterator iterator();
    ......
}
```

- java.util.Iterator

```
package java.util;

public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

```
//测试代码
.....
public static void process(Collection c) {
    Iterator i = c.iterator(); //创建迭代器对象

    //通过迭代器遍历聚合对象
    while(i.hasNext()) {
        System.out.println(i.next().toString());
    }
}
.....
```

迭代器模式优点：

1）支持以不同的方式遍历一个聚合对象，在同一个聚合对象上可以定义多种遍历方式

2）简化了聚合类

3）由于引入了抽象层，增加新的聚合类和迭代器类都很方便，无须修改原有代码，符合开闭原则

迭代器模式缺点：

1）在增加新的聚合类时需要对应地增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性

2）抽象迭代器的设计难度较大，需要充分考虑到系统将来的扩展。在自定义迭代器时，创建一个考虑全面的抽象迭代器并不是一件很容易的事情

在以下情况下可以使用迭代器模式：

1）访问一个聚合对象的内容而无须暴露它的内部表示

2）需要为一个聚合对象提供多种遍历方式

3）为遍历不同的聚合结构提供一个统一的接口，在该接口的实现类中为不同的聚合结构提供不同的遍历方式，而客户端可以一致性地操作该接口

第23章 观察者模式 ★

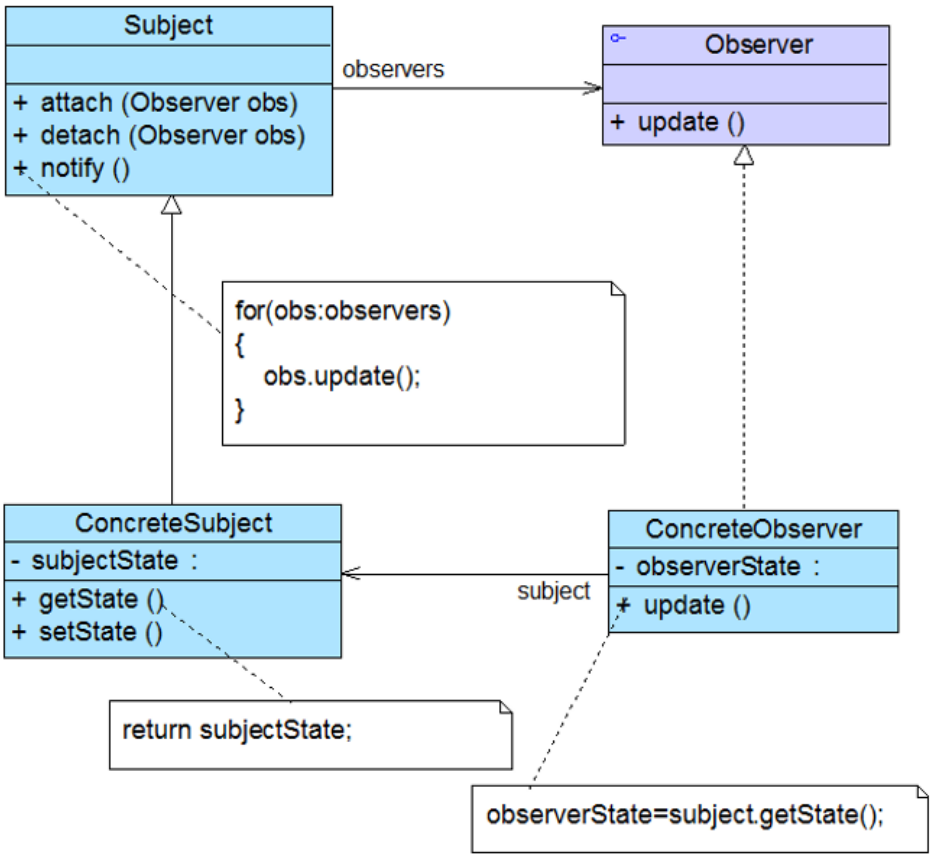
模式动机

- ✓ 软件系统：一个对象的状态或行为的变化将导致其他对象的状态或行为也发生改变，它们之间将产生联动
- ✓ 观察者模式：
  - 定义了对象之间一种一对多的依赖关系，让一个对象的改变能够影响其他对象
  - 发生改变的对象称为观察目标，被通知的对象称为观察者
  - 一个观察目标可以对应多个观察者

模式定义

- ✓ 观察者模式(Observer Pattern)：定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。
- ✓ 观察者模式又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式
- ✓ 观察者模式是一种对象行为型模式

模式结构



观察者模式包含如下角色：

Subject：目标

ConcreteSubject：具体目标

Observer：观察者

ConcreteObserver：具体观察者

模式分析

✓ 说明：

- 有时候在具体观察者类ConcreteObserver中需要使用到具体目标类ConcreteSubject中的状态（属性），会存在关联或依赖关系
- 如果在具体层之间具有关联关系，系统的扩展性将受到一定的影响，增加新的具体目标类有时候需要修改原有观察者的代码，在一定程度上违背了开闭原则，但是如果原有观察者类无须关联新增的具体目标，则系统扩展性不受影响

模式分析

✓ 抽象目标类示例代码：

```
import java.util.*;
public abstract class Subject {
    //定义一个观察者集合用于存储所有观察者对象
    protected ArrayList observers<Observer> = new ArrayList();

    //注册方法，用于向观察者集合中增加一个观察者
    public void attach(Observer observer) {
        observers.add(observer);
    }

    //注销方法，用于在观察者集合中删除一个观察者
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    //声明抽象通知方法
    public abstract void notify();
}
```

模式分析

✓ 具体目标类示例代码：

```
public class ConcreteSubject extends Subject {
    //实现通知方法
    public void notify() {
        //遍历观察者集合，调用每一个观察者的响应方法
        for(Object obs:observers) {
            ((Observer)obs).update();
        }
    }
}
```

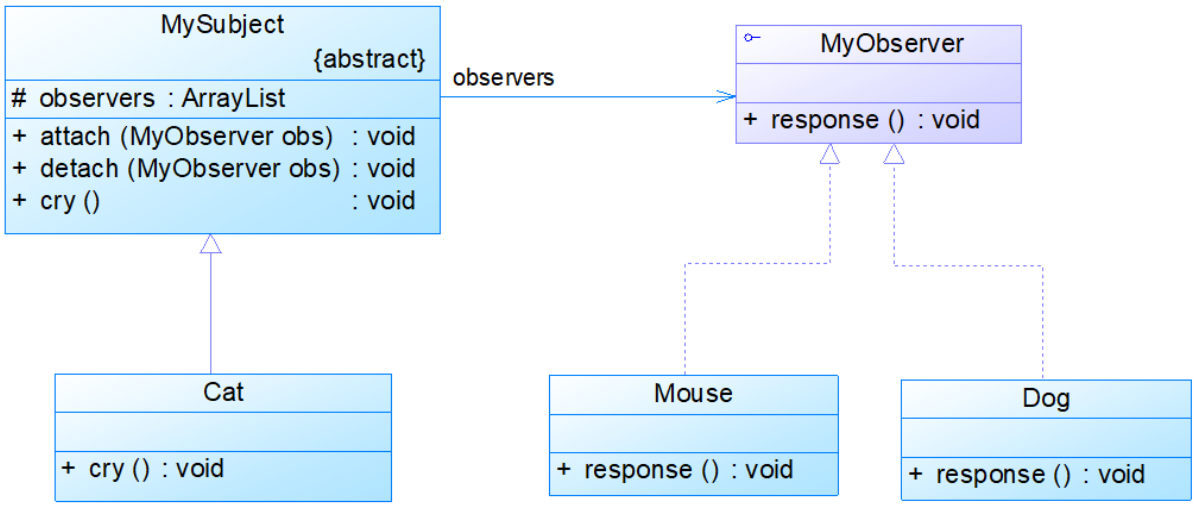
模式实例

✓ 猫、狗与老鼠 ： 实例说明

- 假设猫是老鼠和狗的观察目标，老鼠和狗是观察者，猫叫老鼠跑，狗也跟着叫，使用观察者模式描述该过程。

模式实例

✓ 猫、狗与老鼠： 参考类图



观察者模式优点：

1）可以实现表示层和数据逻辑层的分离

2）在观察目标和观察者之间建立一个抽象的耦合

3）支持广播通信，简化了一对多系统设计的难度

4）符合开闭原则，增加新的具体观察者无须修改原有系统代码，在具体观察者与观察目标之间不存在关联关系的情况下，增加新的观察目标也很方便

观察者模式缺点：

1）将所有的观察者都通知到会花费很多时间

2）如果存在循环依赖时可能导致系统崩溃

3）没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而只是知道观察目标发生了变化

在以下情况下可以使用观察者模式：

1）一个抽象模型有两个方面，其中一个方面依赖于另一个方面，将这两个方面封装在独立的对象中使它们可以各自独立地改变和复用

2）一个对象的改变将导致一个或多个其他对象发生改变，且并不知道具体有多少对象将发生改变，也不知道这些对象是谁

3）需要在系统中创建一个触发链

第25章 策略模式

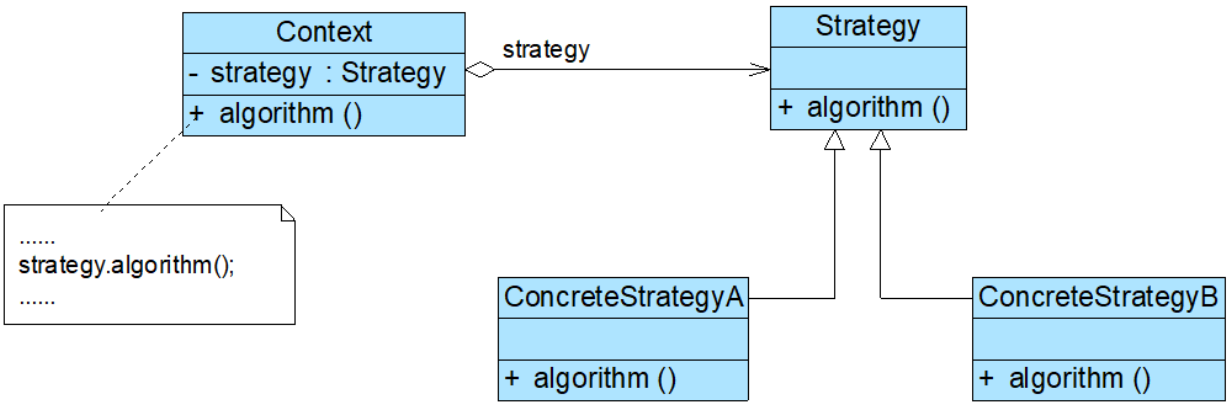
模式动机

- ✓ 实现某个目标的途径不止一条，可根据实际情况选择一条合适的途径
- ✓ 软件开发：
  - 多种算法，例如排序、查找、打折等
  - 使用硬编码(Hard Coding)实现将导致系统违背开闭原则，扩展性差，且维护困难
  - 可以定义一些独立的类来封装不同的算法，每一个类封装一种具体的算法 → 策略类 → 策略模式

模式定义

- ✓ 策略模式(Strategy Pattern)：定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。
- ✓ 策略模式让算法独立于使用它的客户而变化
- ✓ 策略模式是一种对象行为型模式

模式结构



策略模式包含如下角色：

Context：环境类

Strategy：抽象策略类

ConcreteStrategy：具体策略类

模式分析

- ✓ 每一个封装算法的类称之为策略(Strategy)类
- ✓ 策略模式提供了一种可插入式(Pluggable)算法的实现方案
- ✓ 环境类示例代码：

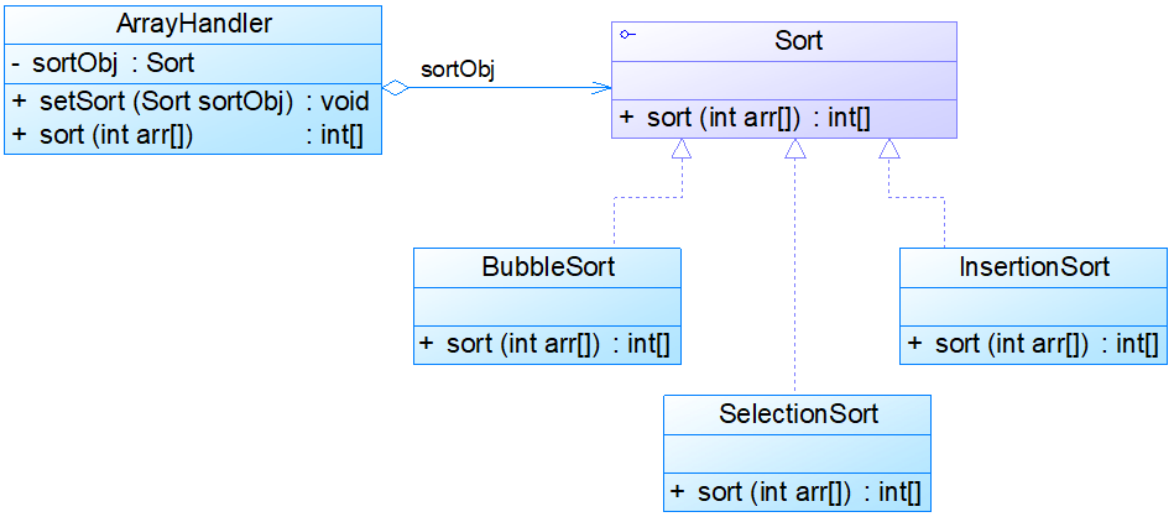
```
public class Context {
   private Strategy strategy; //维持一个对抽象策略类的引用

   //注入策略对象
   public void setStrategy(Strategy strategy) {
      this.strategy= strategy;
   }

   //调用策略类中的算法
   public void algorithm() {
      strategy.algorithm();
   }
}
```

模式实例

- ✓ 排序策略：参考类图



策略模式优点：

- ✓ 提供了对开闭原则的完美支持，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为
- ✓ 提供了管理相关的算法族的办法
- ✓ 提供了一种可以替换继承关系的办法
- ✓ 可以避免多重条件选择语句
- ✓ 提供了一种算法的复用机制，不同的环境类可以方便地复用策略类

策略模式缺点：

- ✓ 客户端必须知道所有的策略类，并自行决定使用哪一个策略类
- ✓ 将造成系统产生很多具体策略类
- ✓ 无法同时在客户端使用多个策略类

在以下情况下可以使用策略模式：

- ✓ 一个系统需要动态地在几种算法中选择一种
- ✓ 避免使用难以维护的多重条件选择语句
- ✓ 不希望客户端知道复杂的、与算法相关的数据结构，提高算法的保密性与安全性