结构型模式（一）

2020年4月13日　　11:45

结构型模式（一）

作业要求：

1、试画出适配器模式实例的结构图和实现代码，并对模式进行分析。

1、试画出桥接模式实例的结构图和实现代码，并对模式进行分析。

3、使用Java语言实现一个双向适配器实例，使得猫可以学狗叫，狗可以学猫抓老鼠，绘制相应类图并使用代码编程模拟。

- 作业与笔记github地址：https://github.com/baobaotql/CCNU_DesignPattern

- **结构模式总述**

  结构型模式描述如何将类或对象按某种布局组成更大的结构。它分为类结构型模式和对象结构型模式，前者采用继承机制来组织接口和类，后者采用组合或聚合来组合对象。

  由于组合关系或聚合关系比继承关系耦合度低，满足“合成复用原则”，所以对象结构型模式比类结构型模式具有更大的灵活性。

  - 结构型模式分为以下 7 种：
    - **代理（Proxy）模式**：为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。
    - **适配器（Adapter）模式**：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。
    - **桥接（Bridge）模式**：将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现的，从而降低了抽象和实现这两个可变维度的耦合度。
    - **装饰（Decorator）模式**：动态地给对象增加一些职责，即增加其额外的功能。
    - **外观（Facade）模式**：为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。
    - **享元（Flyweight）模式**：运用共享技术来有效地支持大量细粒度对象的复用。
    - **组合（Composite）模式**：将对象组合成树状层次结构，使用户对单个对象和组合对象具有一致的访问性。

  以上 7 种结构型模式，除了适配器模式分为类结构型模式和对象结构型模式两种，其他的全部属于对象结构型模式，下面我们会分别、详细地介绍它们的特点、结构与应用。

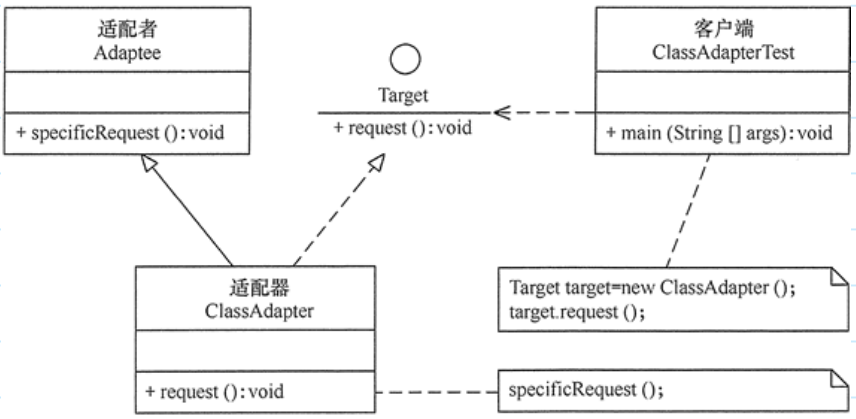
- **适配器模式**
- **Q1：试画出适配器模式实例的结构图和实现代码，并对模式进行分析。**
- **A1：**
  - **适配器模式（Adapter）的定义：**

    将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。适配器模式分为类结构型模式和对象结构型模式两种，前者类之间的耦合度比后者

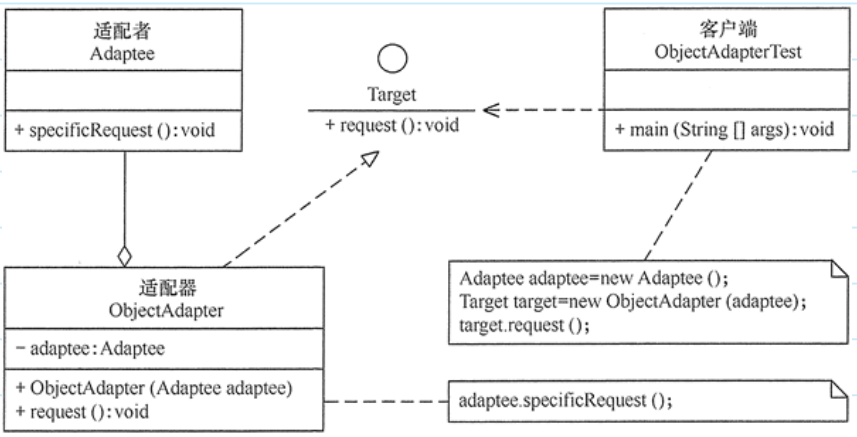
高，且要求程序员了解现有组件库中的相关组件的内部结构，所以应用相对较少些。

- **优点：**
  - 客户端通过适配器可以透明地调用目标接口。
  - 复用了现存的类，程序员不需要修改原有代码而重用现有的适配者类。
  - 将目标类和适配者类解耦，解决了目标类和适配者类接口不一致的问题。
- **缺点：**
  - 对类适配器来说，更换适配器的实现过程比较复杂。
- **模式结构**
  - 目标接口：当前系统业务所期待的接口，它可以是抽象类或接口。
  - 适配者类：它是被访问和适配的现存组件库中的组件接口。
  - 适配器类：它是一个转换器，通过继承或引用适配者的对象，把适配者接口转换成目标接口，让客户按目标接口的格式访问适配者。

  - 类适配器模式结构图



  - 对象适配器模式



- **模式分析**
  - **应用场景**
    - 以前开发的系统存在满足新系统功能需求的类，但其接口同新系统的接口不一致。
    - 使用第三方提供的组件，但组件接口定义和自己要求的接口定义不同。
  - **类适配器模式代码**
    - 对象适配器模式中的“目标接口”和“适配者类”的代码同类适配器模式一样，只要修改适配器类和客户端的代码即可

```
package adapter;
```

```
//目标接口
interface Target
{
    public void request();
}
//适配者接口
class Adaptee
{
    public void specificRequest()
    {
        System.out.println("适配者中的业务代码被调用！");
    }
}
//类适配器类
class ClassAdapter extends Adaptee implements Target
{
    public void request()
    {
        specificRequest();
    }
}
//客户端代码
public class ClassAdapterTest
{
    public static void main(String[] args)
    {
        System.out.println("类适配器模式测试：");
        Target target = new ClassAdapter();
        target.request();
    }
}
```

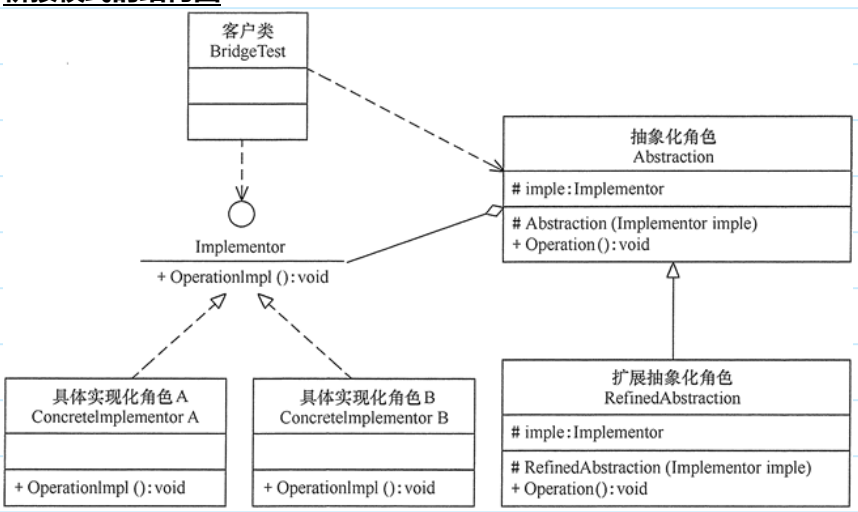
- 对象适配器模式代码

```
package adapter;
//对象适配器类
class ObjectAdapter implements Target
{
    private Adaptee adaptee;
    public ObjectAdapter(Adaptee adaptee)
    {
        this.adaptee=adaptee;
    }
    public void request()
    {
        adaptee.specificRequest();
    }
}
//客户端代码
public class ObjectAdapterTest
{
    public static void main(String[] args)
    {
        System.out.println("对象适配器模式测试：");
        Adaptee adaptee = new Adaptee();
        Target target = new ObjectAdapter(adaptee);
```

```
            target.request();
        }
    }
```

- **桥接模式**
- **Q2：试画出桥接模式实例的结构图和实现代码，并对模式进行分析**
- **A2：**
  - **桥接（Bridge）模式的定义：**
    将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。
  - **优点：**
    - 由于抽象与实现分离，所以扩展能力强；
    - 其实现细节对客户透明。
  - **缺点：**
    - 由于聚合关系建立在抽象层，要求开发者针对抽象化进行设计与编程，这增加了系统的理解与设计难度。
  - **模式结构**
    - **抽象化角色**：定义抽象类，并包含一个对实现化对象的引用。
    - **扩展抽象化角色**：是抽象化角色的子类，实现父类中的业务方法，并通过组合关系调用实现化角色中的业务方法。
    - **实现化角色**：定义实现化角色的接口，供扩展抽象化角色调用。
    - **具体实现化角色**：给出实现化角色接口的具体实现。

    - **桥接模式的结构图**



  - **模式分析**
    - **应用场景**
      - 当一个类存在两个独立变化的维度，且这两个维度都需要进行扩展时。
      - 当一个系统不希望使用继承或因为多层次继承导致系统类的个数急剧增加时。
      - 当一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性时。
    - **扩展**
      - 有时桥接（Bridge）模式可与适配器模式联合使用。当桥接（Bridge）模式的实现化角色的接口与现有类的接口不一致时，可以在二者中间定义一个适配器将二者连接起来
    - **分析桥接模式**

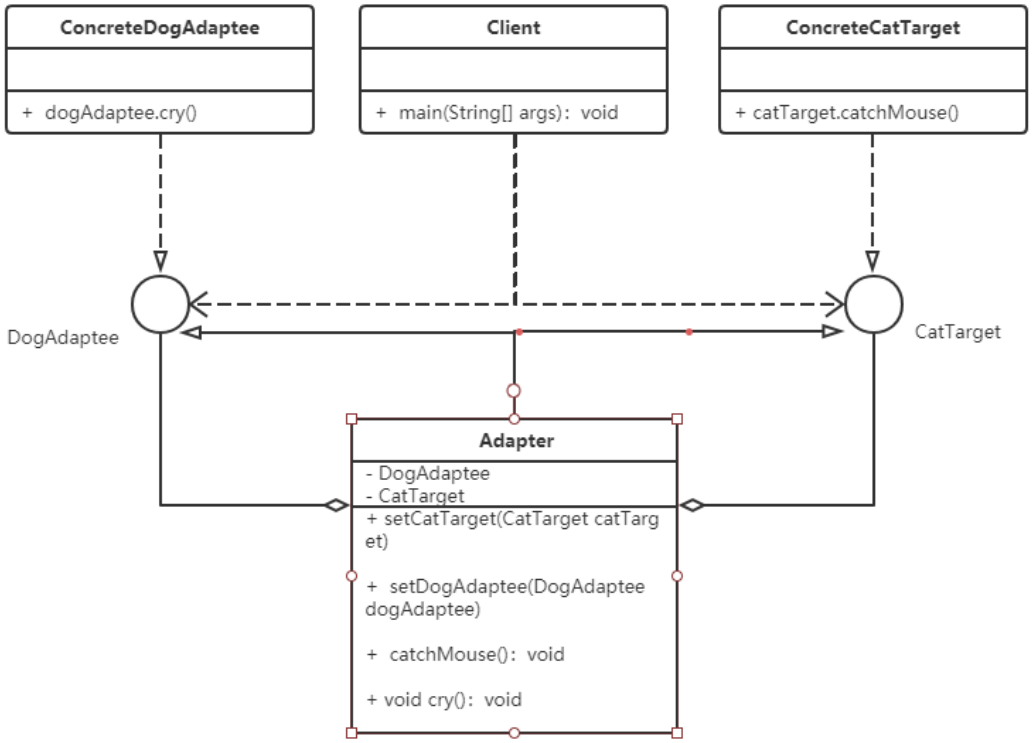
- **抽象化**：将对象的共同性质抽取出来形成类的过程就是抽象化
- **实现化**：针对抽象化给出的具体实现，就是实现化。实现化产生的对象比抽象化更具体，是对抽象化事物的进一步具体化的产物
- **脱耦**：将抽象化和实现化之间的耦合解脱开，或者说是将他们之间的强关联改成弱关联，将两个角色之间的继承关系改为关联关系

- **桥接模式代码**

```
package bridge;
public class BridgeTest
{
    public static void main(String[] args)
    {
        Implementor imple=new ConcreteImplementorA();
        Abstraction abs=new RefinedAbstraction(imple);
        abs.Operation();
    }
}
//实现化角色
interface Implementor
{
    public void OperationImpl();
}
//具体实现化角色
class ConcreteImplementorA implements Implementor
{
    public void OperationImpl()
    {
        System.out.println("具体实现化(Concrete Implementor)角色被访问");
    }
}
//抽象化角色
abstract class Abstraction
{
   protected Implementor imple;
   protected Abstraction(Implementor imple)
   {
       this.imple=imple;
   }
   public abstract void Operation();
}
//扩展抽象化角色
class RefinedAbstraction extends Abstraction
{
   protected RefinedAbstraction(Implementor imple)
   {
       super(imple);
   }
   public void Operation()
   {
       System.out.println("扩展抽象化(Refined Abstraction)角色被访问");
       imple.OperationImpl();
   }
}
```

Q3：使用Java语言实现一个双向适配器实例，使得猫可以学狗叫，狗可以学猫抓老鼠，绘制相应类图并使用代码编程模拟

A3：




```
public interface CatTarget {
    public abstract void catchMouse();
}
public class ConcreteCatTarget implements CatTarget {
    @Override
    public void catchMouse() {
        System.out.println("抓老鼠！");
    }
}

public interface DogAdaptee {
    public abstract void cry();
}
public class ConcreteDogAdaptee implements DogAdaptee {
    @Override
    public void cry() {
        System.out.println("汪汪叫！");
    }
}

public class Adapter implements CatTarget,DogAdaptee {

    private CatTarget catTarget;
    private DogAdaptee dogAdaptee;
```

```
    public Adapter() {
    }

    public void setCatTarget(CatTarget catTarget) {
        this.catTarget = catTarget;
    }

    public void setDogAdaptee(DogAdaptee dogAdaptee) {
        this.dogAdaptee = dogAdaptee;
    }

    //即目标类调用适配者中的方法
    public void catchMouse() {
        System.out.print("猫学狗叫");
        dogAdaptee.cry();

    }

    //即适配者调用目标类中的方法
    public void cry(){
        System.out.print("狗学猫抓老鼠");
        catTarget.catchMouse();
    }

}

public class Client {
    public static void main(String[] args) {
        //适配器
        Adapter adapter = (Adapter) XMLUtils.getBean("adapterPattern");

        //目标类通过适配器调用适配者方法
        CatTarget concreteCatTarget = (ConcreteCatTarget) XMLUtils.getBean("adapterPatternTarger");
        adapter.setCatTarget(concreteCatTarget);
        adapter.cry();

        //适配者通过适配器调用目标类方法
        DogAdaptee concreteDogAdaptee = (ConcreteDogAdaptee) XMLUtils.getBean("adapterPatternAdaptee");
        adapter.setDogAdaptee(concreteDogAdaptee);
        adapter.catchMouse();
    }
}
```