面向对象设计原则

可维护性：系统能够易于扩展和修改

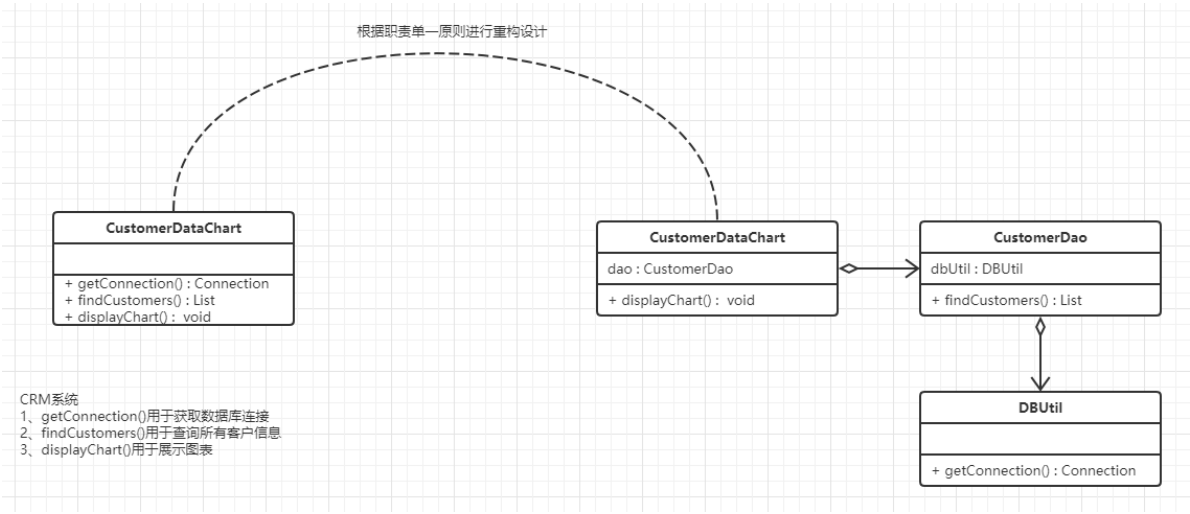
可复用性：设计方案或源代码的复用

单一职责原则

定义：一个类的职责尽量单一，控制类的粒度大小。高内聚，低耦合

原因：一个类有多个职责，则意味着有多个变化因素，职责与职责之间是耦合在一起的，一个职责的变化可能导致其他职责的变化，所以应该将不同的职责单独封装为一个类，达到职责与职责之间的解耦。

应用示例：

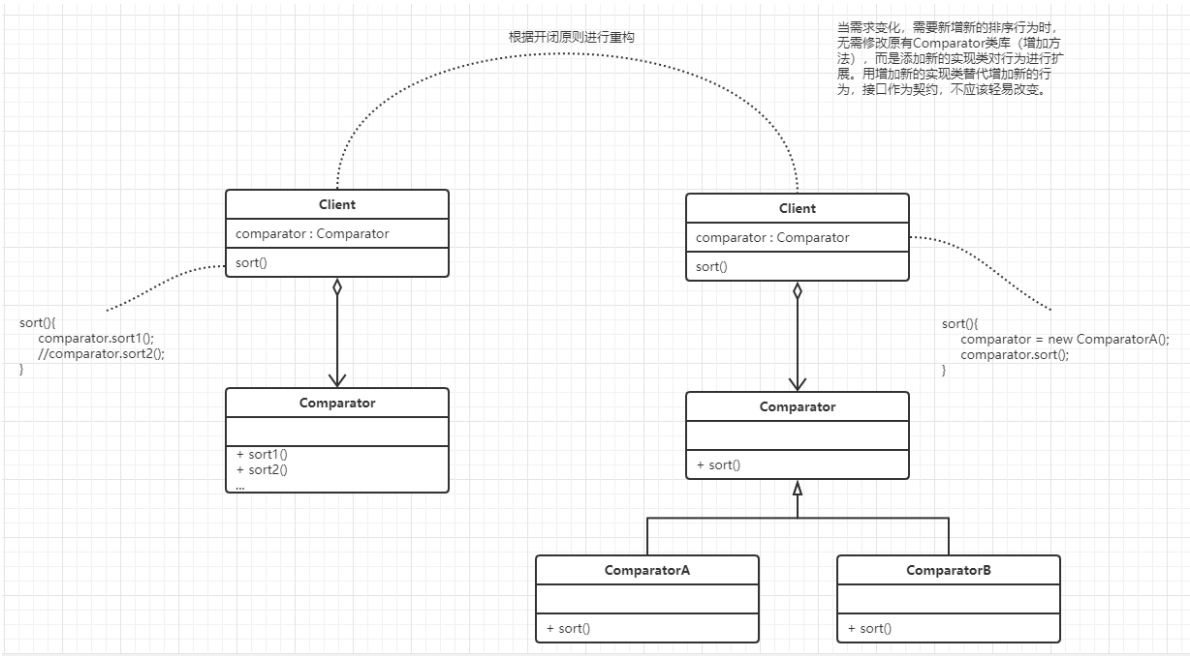


开闭原则

定义：系统对扩展开放，对修改关闭。对设计框架进行抽象化设计，具体的不同的实现行为放在实现层进行实现，从而保证设计框架的稳定性。对于新的行为，无须修改原有类库（代码），增加新的不同的行为实现即可实现扩展新的行为。用抽象构建框架，用实现扩展细节。

原因：因为随着时间的推移，1、需求是一定会发生变化的，2、软件规模越来越大，3、维护成本越来越高。开闭原则可以实现在不改变原有设计框架的基础上，通过添加新的具体行为类来达到扩展系统的目的。

应用示例：Collections.sort(list, Comparator)方法，新增自定义的Comparator实现类，达到对list根据新的行为进行排序，而没有对原有类库进行任何改变。策略模式。



里氏替换原则

定义：所有引用基类的地方一定可以由子类替换。

原因：用抽象类定义父类，子类实现抽象类。使用基类对对象进行定义，在运行时在确定真正的子类类型，用子类对象代替基类对象。通过增加新的子类扩展系统功能。

应用实例：java语法天生支持。

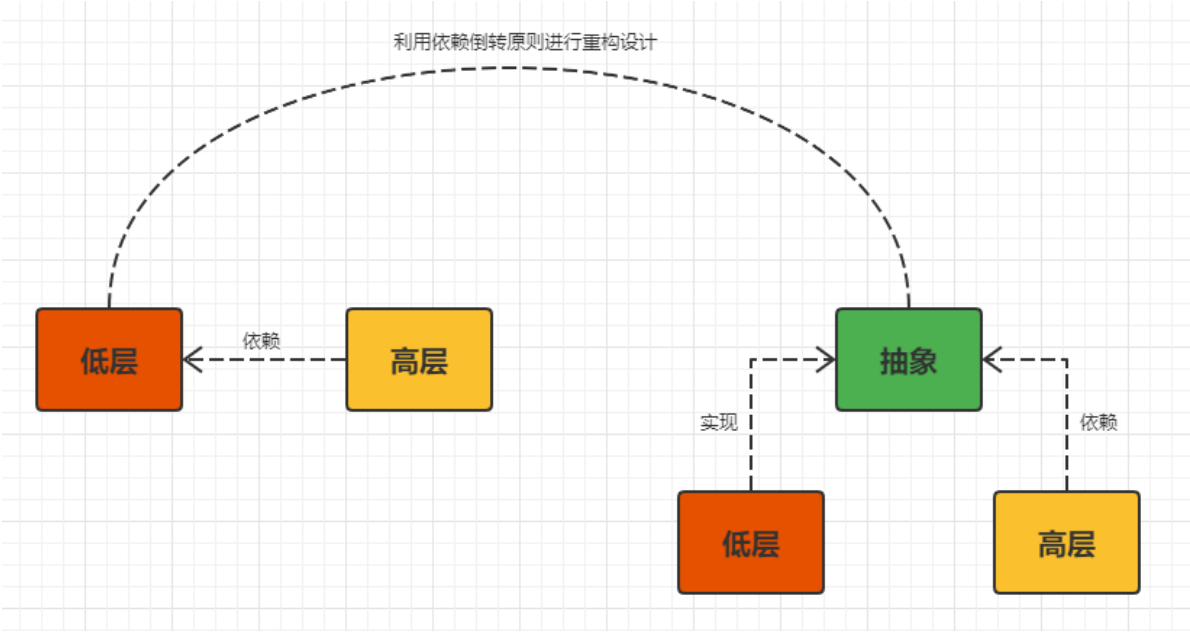
```
//多态写法
Service  service = new ServiceImpl();
service.do();

// 定义方法
void method(Service service){...};
```

依赖倒转原则

定义：高层模块不应该依赖低层模块，两者都应该依赖抽象。简单来说就是面向接口编程，不要面向实现编程。

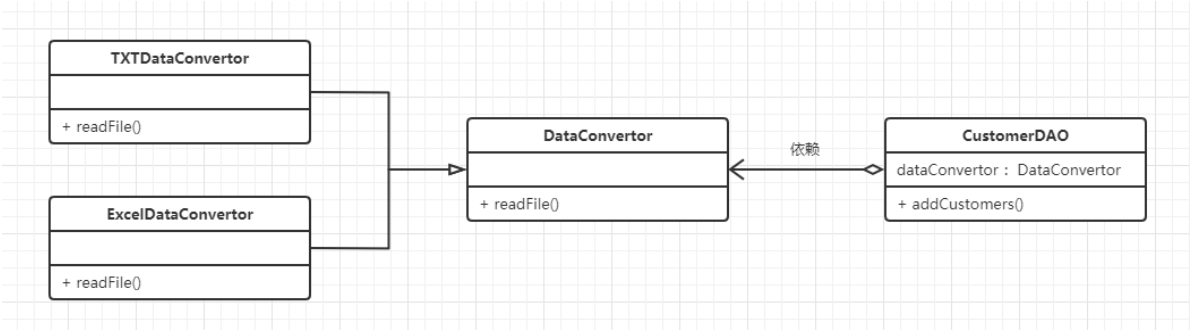
原因：高层模块和低层模块之间添加一层抽象，进行了松耦合设计，保证了设计框架的稳定性，在新增低层模块时，只需改动高层模块，无需改动其他低层模块。

应用示例：



变量类型声明、参数类型声明、方法返回类型声明，数据类型转换等，不用用具体类去声明，而是用抽象层去声明（变量有两个类型，一个是表面类型，一个是实际类型）。所以具体类不要有除了抽象层声明的方法外的其他方法，否则无法调用到。依赖注入方法有：构造注入、setter注入、接口注入。

开闭原则、里氏替换原则、依赖倒转原则的应用示例：



解释：

1、CustomerDAO为高层模块，Convertor相关类为低层模块，CustomerDAO和具体的子Convertor类两者都依赖于DataConvertor接口，依赖倒转原则体现

2、CustomerDAO声明的变量dataConvertor的表面类型为抽象接口，具体子类Convertor可以在运行时覆盖父类对象，里氏替换原则体现

3、新增低层模块类，比如SqlDataConvertor时，只需实现接口DataConvertor，其他低层模块无需修改，高层模块直接使用，影响面小，危险性低，系统稳定性高

4、这三个原则是经常同时出现的，里氏替换原则是基础，依赖倒转原则是手段，开闭原则是目标

接口隔离原则

定义：客户端不要依赖于它不需要的接口。简单的说就是接口的方法不应该太多，当然也不能粒度太细。

与职责单一原则的区别：职责单一原则的角度是类的职责，方法数量不限制。接口隔离原则是在职责单一的基础上，还要求接口的方法数量不能太多。

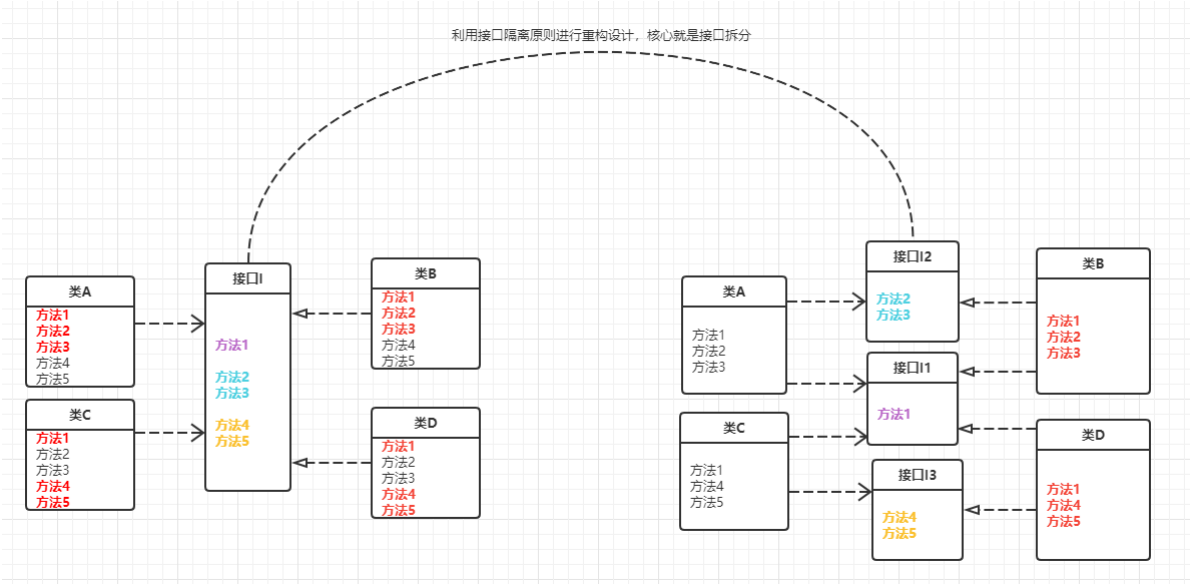
原因：

1、实现类可能需要实现很多不用的方法

2、向客户端暴露了很多它不需要的方法

3、接口臃肿

应用示例：



组合复用原则

定义：优先使用组合对象的方式，而不是继承的方式对代码进行复用。

解释：

1、复用有两种方式，一是组合关系（将已有对象纳入新的对象，成为新对象的一部分），一是继承（需严格遵守里氏替换原则）。

2、两个类的关系是“Has - A”关系，则用组合复用，若是“Is - A”关系，则用继承复用。

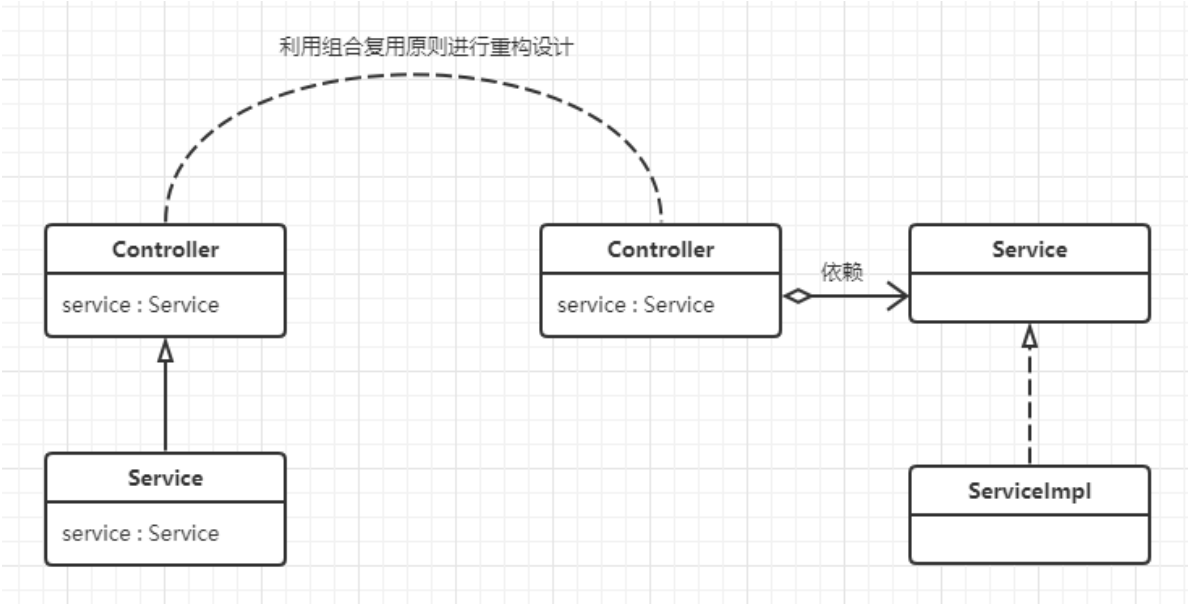
原因：

1、组合复用对象，系统灵活性更高。

2、继承破坏了类的封装性，改动基类时容易导致影响到所有子类，耦合性高于组合关系复用

3、继承属于静态复用、“白箱”复用，组合复用属于动态复用（组合对象为基类，程序运行时才能确定具体子类），“黑箱”复用（方法的复用封装在组合对象内部，不泄露细节实现）。

应用示例：



迪米特法则

定义：最少知识原则。通俗的说法就是一个对象只和他的直接朋友交流，不要和陌生人说话。

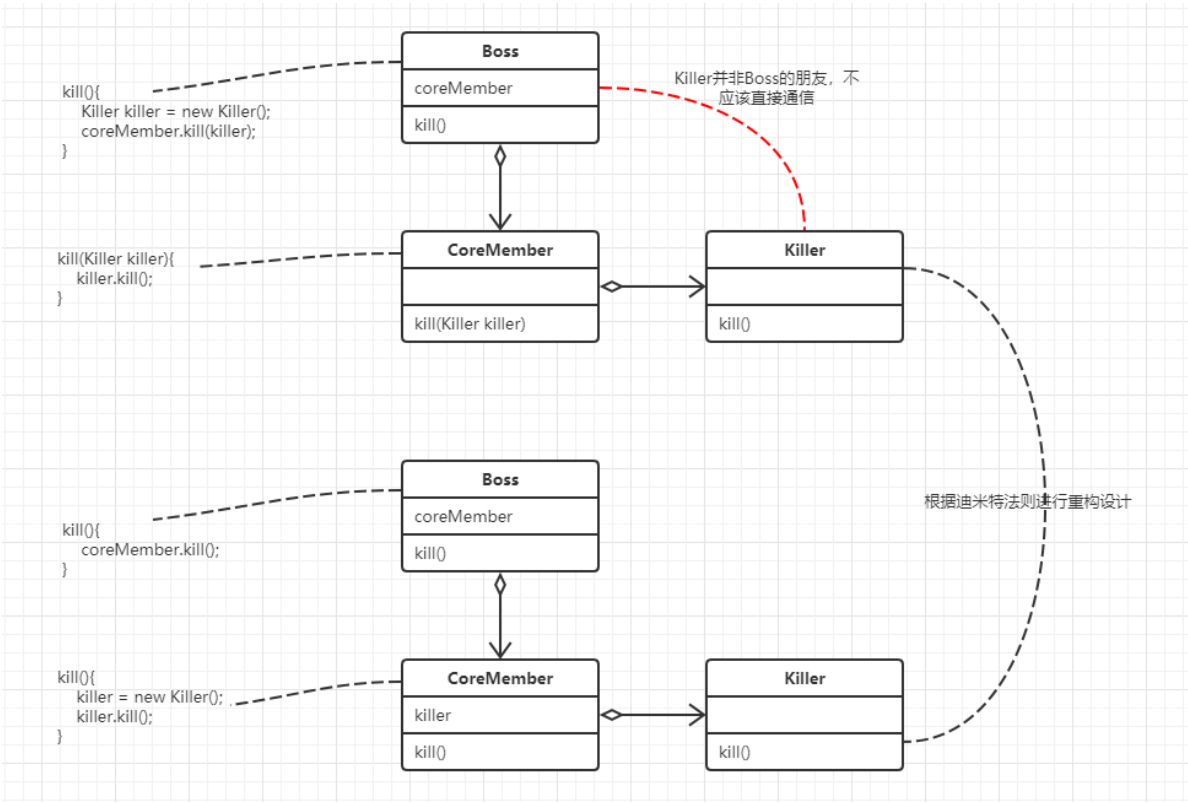
解释：

1、两个对象之间不需要直接通信的话，往往可以加入“第三者”，应用迪米特法则

2、对象的成员变量，对象方法的出入参对象，对象自己都是他的直接朋友

原因：高内聚，低耦合

应用示例：



创建型模式

关注对象的创建

单例模式Singleton

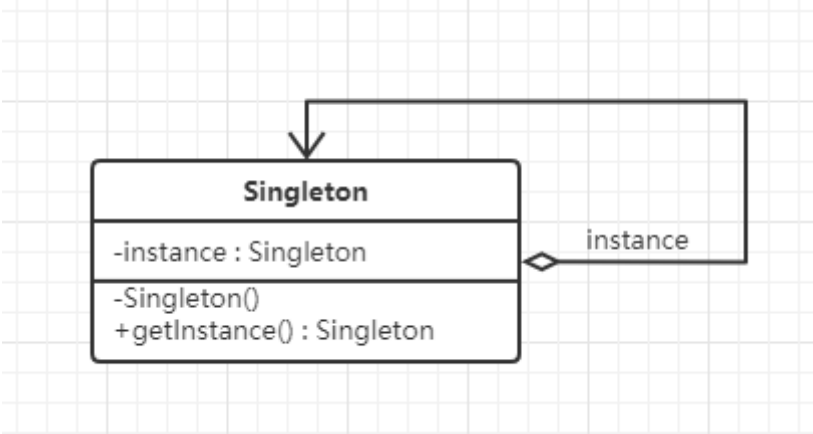
- ## 定义

保证一个类只有一个实例，并提供一个全局访问点

- ## 应用场景

在线人数统计。

- ## UML类图



- ## 代码示例

```
//饱汉式
public class Singleton {
 private Singleton(){}
 //在自己内部定义自己一个实例，是不是很奇怪？
 //注意这是 private 只供内部调用
 private static Singleton instance = new Singleton();
 //这里提供了一个供外部访问本 class 的静态方法，可以直接访问
 public static Singleton getInstance() {
    return instance;
 }
}
```

```
//懒汉式
public class Singleton {
 private static Singleton instance = null;
 public static synchronized Singleton getInstance() {
     //这个方法比上面有所改进，不用每次都进行生成对象，只是第一次
     //使用时生成实例，提高了效率！
     if (instance==null){
         instance=new Singleton();
     }
     return instance;
 }
}
```

懒汉式为保证线程安全需加sychronized，否则在getInstance()时可能会获取多个Singleton实例，如果需要提高懒汉式性能，可以通过volatile+DCL方式。目前一般采用静态内部类的方式实现单例模式。

```
//volatile+DCL
public class Singleton {
    // volatile保证线程间变量可见性
    private static volatile Singleton instance = null;
    public static Singleton getInstance(){
        if (instance == null){  //第一重判断
            synchronized (Singleton.class){ //  进行锁定
                if (instance == null) { //第二重判断
                    return new Singleton(); //  创建实例
                }
            }
        }
        return instance;
    }
}
```

简单工厂模式SimpleFactory

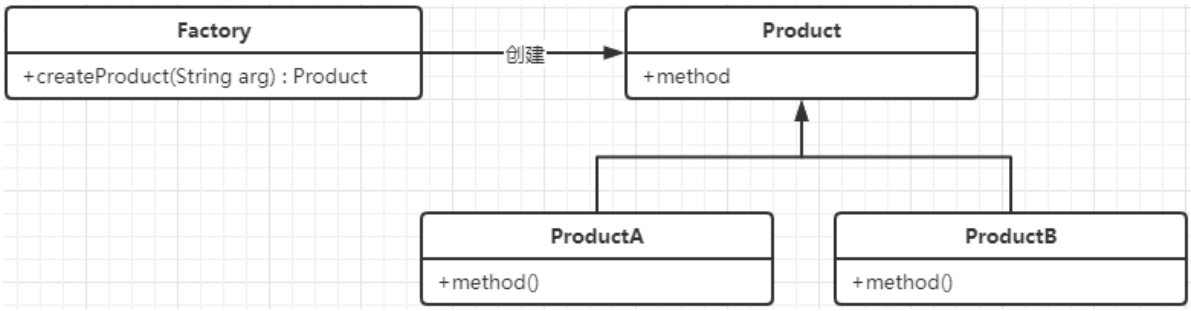
- ## 定义

提供一个工厂类，根据参数的不同创建不同的具体产品类

- ## 应用场景

new Sample(参数)时，在构造对象时，有大量的初始化工作需要处理，使用sample实例的类无需关心对象的创建，只关心sample实例的使用，sample对象的创建交给sample工厂，分离对象的创建和使用。

- ## UML类图



- ## 代码实例

```
//  工厂类
public class Factory {
    public static Product createProduct(String name){
        Product product = null;
        if (name.equals("A")) {
            product = new ProductA();
        } else if (name.equals("B")){
            product = new ProductB();
        }
        return product;
    }
}
```

客户端代码

```
//  通过工厂类创建具体产品对象
Product product = Factory.createProduct("A");
```

- ## 优点

1. 工厂角色分离了产品对象的创建和使用，客户端只关注使用，符合职责单一原则

- ## 缺点

1. 工厂类扩展困难，当有新的产品子类ProductC时，需要重新修改工厂类代码，违背开闭原则

工厂方法模式FactoryMethod

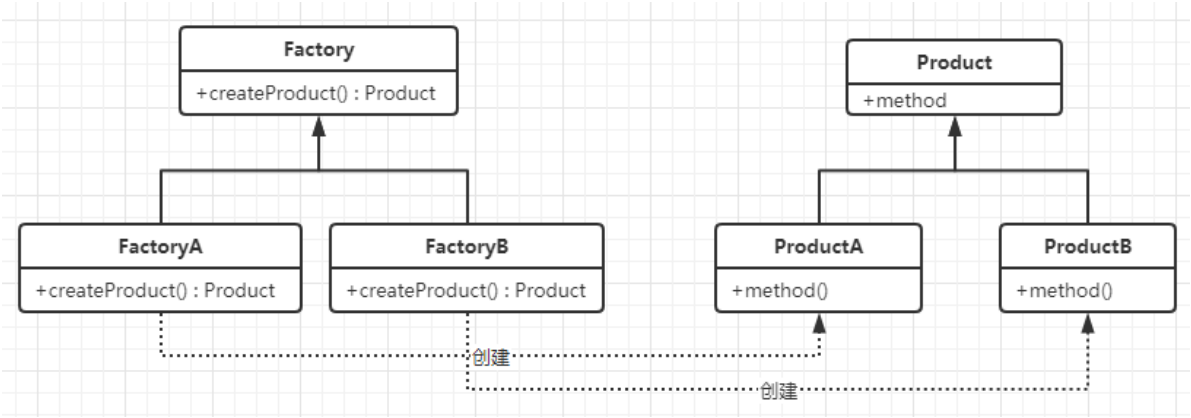
- ## 定义

定义一个创建对象的接口，将具体产品的创建延迟到工厂子类

- ## 应用场景

略

- ## UML类图



- ## 代码示例

```
//  工厂接口
public interface Factory {
    Product createProduct();
}
```

```
//  具体工厂类
public class FactoryA implements Factory {
    public Product createProduct() {
        return new ProductA();
    }
}
```

客户端代码

```
Factory factory = new FactoryA();
Product product = factory.createProduct();
```

- **优点**

1. 工厂角色和产品角色进行了多态性设计，因为工厂类和产品类均有抽象父类
2. 扩展新的具体产品容易，无需修改工厂相关类，只需添加新的具体工厂类和具体产品类即可，符合开闭原则

- **缺点**

1. 具体产品和具体工厂成对出现，类的个数增多，增加了系统复杂度。

抽象工厂模式AbstractFactory

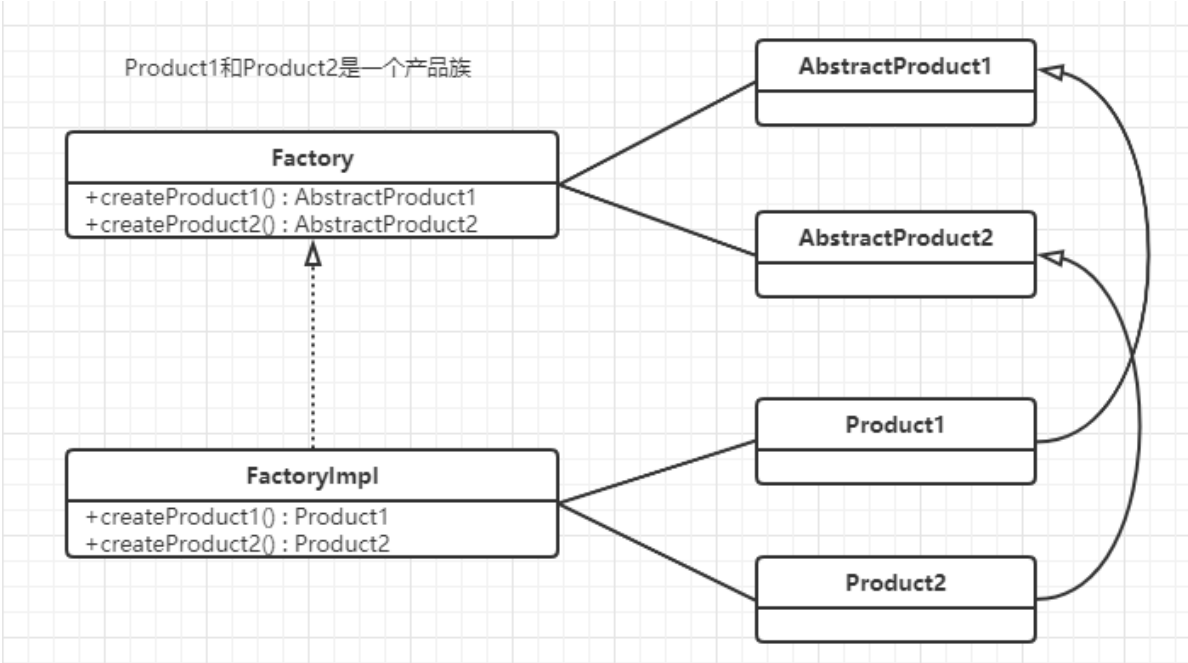
- **定义**

提供一个创建一系列相关或者相互依赖的产品对象族的接口

- **应用场景**

工厂创建的是一个产品族，产品族中的产品需要一起工作，协同工作。

- **UML类图**



- **代码示例**

```
public interface Factory {
    //Product1和Product2属于一个产品族
    Product1 createProduct1();
    Product2 createProduct2();
}

public class FactoryA implements Factory {
    public Product1 createProduct1() {
        return new ProductA1();
    }
```

```
    public Product2 createProduct2() {
        return new ProductA2();
    }
}

public interface Product1 {
}

public interface Product2 {
}

public class ProductA1 implements Product1 {
}

public class ProductA2 implements Product2 {
}
```

客户端代码

```
Factory factory = new FactoryA();
Product1 product1 = factory.createProduct1();
Product2 product2 = factory.createProduct2();
```

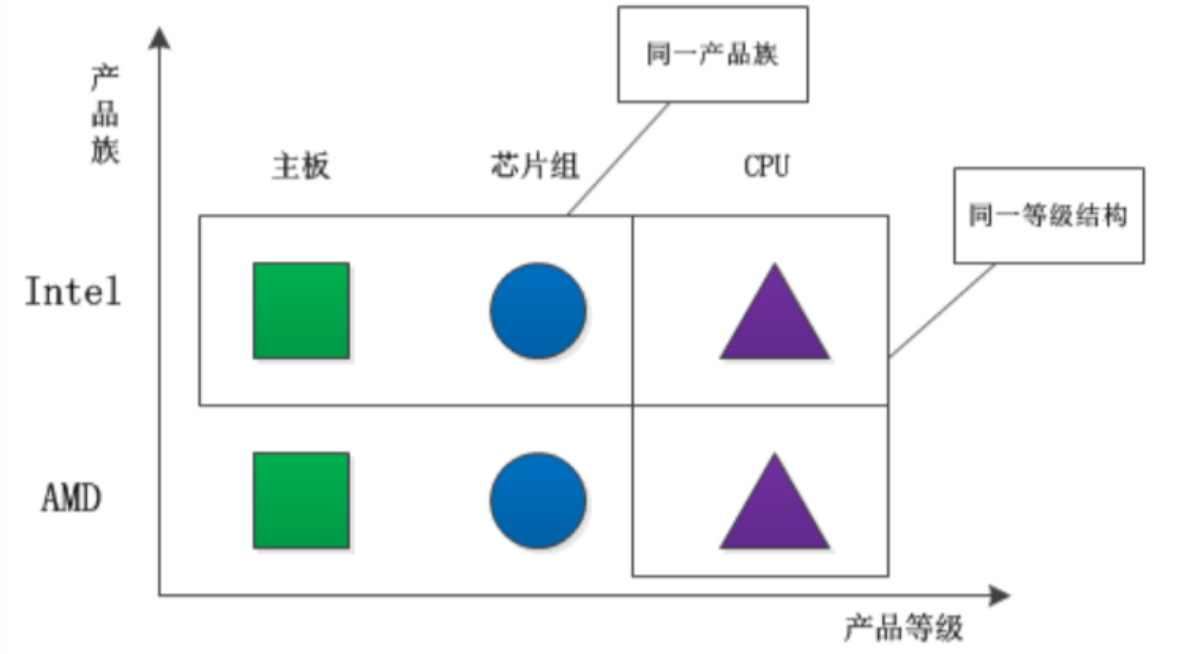
- ## 优点

1. 抽象工厂模式和工厂方法模式最大的区别时，工厂方法模式针对的是一个产品等级结构，而抽象工厂模式针对的是多个产品等级结构
2. 保证客户端可以使用同一个产品族中的多个产品进行协同工作
3. 增加产品族很容易，符合开闭原则

- ## 缺点

1. 增加产品等级结构很麻烦，需要调整工厂类，违背开闭原则

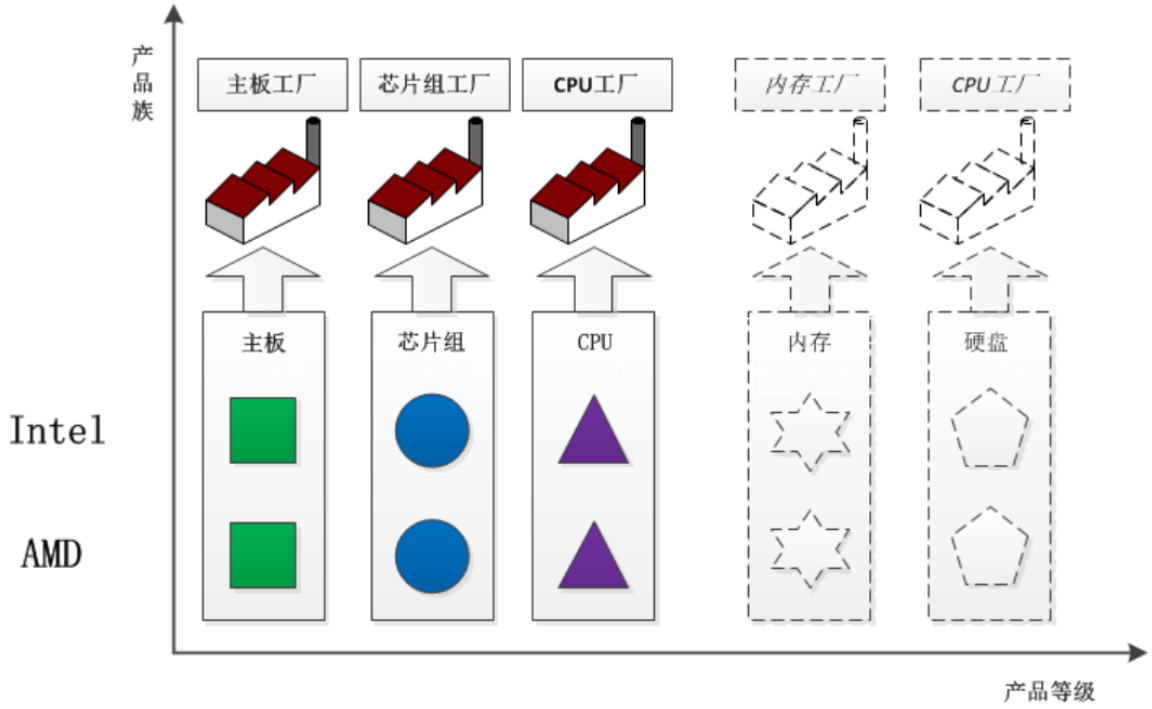
产品族和产品等级结构示意图



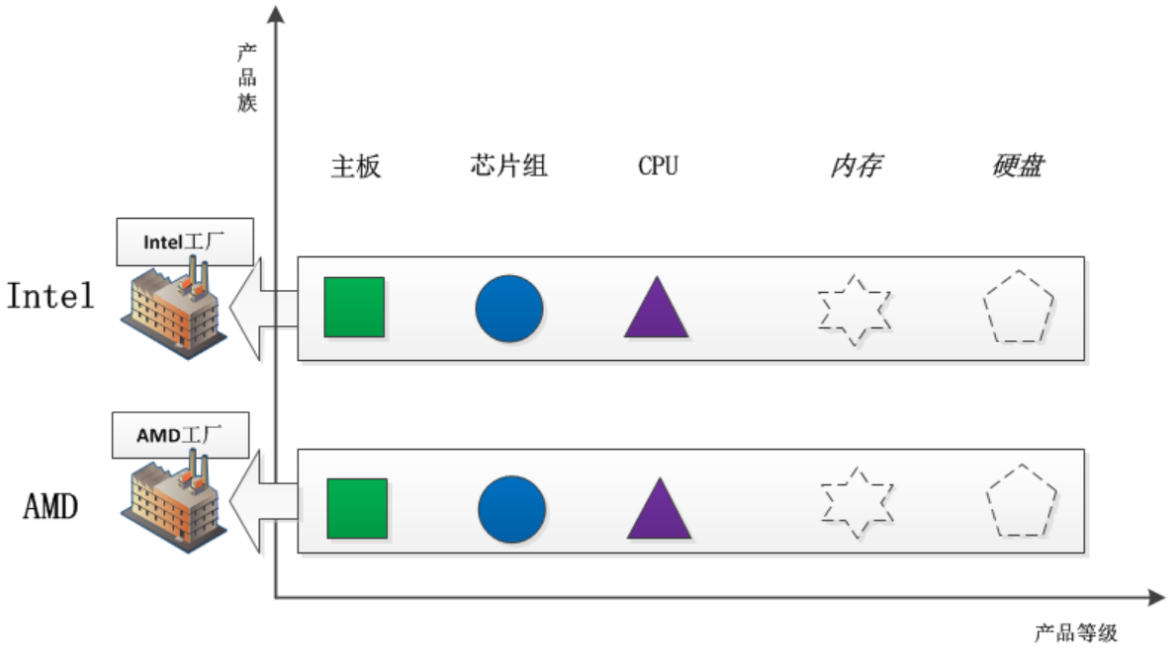
如果用工厂方法模式解决上面问题，则需要三个独立的、平行的工厂等级结构来分别表示主板工厂，芯片工厂，CPU工厂。

问题：

1、当产品等级结构增加时（如增加内存、硬盘等），需要不断调整工厂的产品等级结构来解决。



针对上面问题，使用抽象工厂模式，利用同一个工厂等级结构负责三个不同产品等级中的产品对象的创建。



建造者模式Builder

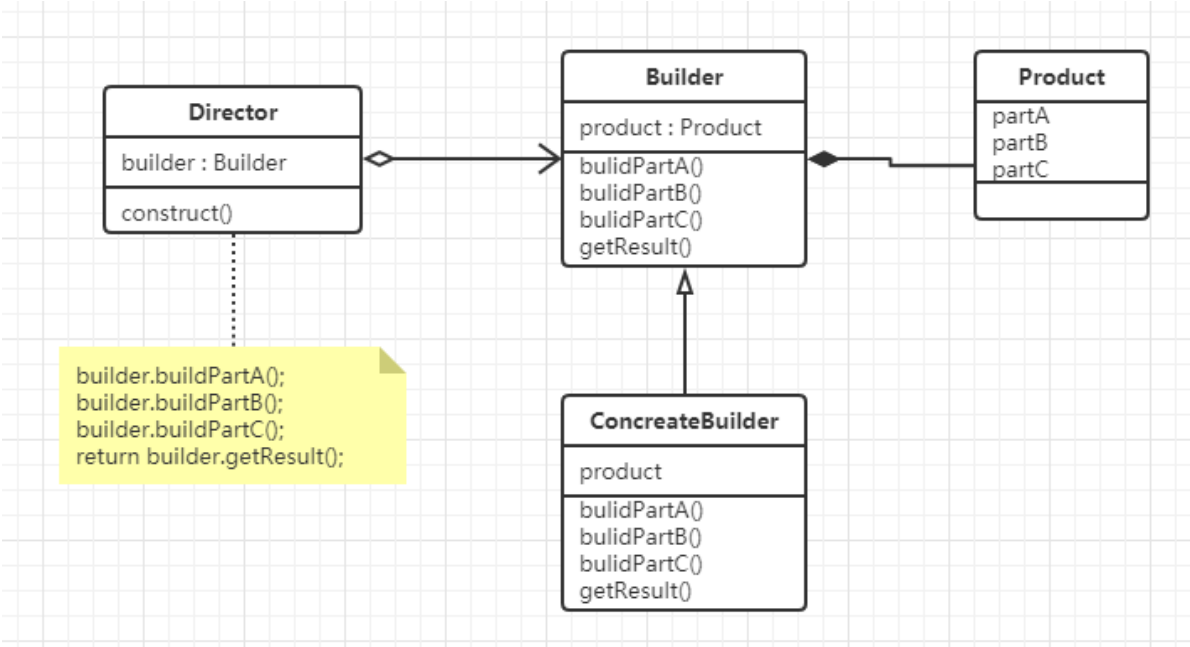
- **定义**

将一个复杂对象的创建与表示分离，与抽象工厂模式的区别，抽象工厂更关注产品对象和工厂对象，而build模式更关注复杂对象的零部件的装配、组合、封装过程。

- **应用场景**

比如生产一个车，需要将车的各个零部件一步一步装配、组合起来，最后得到一个车。

- **UML类图**



- **代码示例**

```
public class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder = builder;
    }

    //  产品对象的构建与组装过程
    public Product construct(){
        builder.buildPartA();
        builder.buildPartB();
        builder.buildPartC();
        return builder.getResult();
    }
}

public abstract class Builder {
    private Product product = new Product();
    public abstract void buildPartA();
    public abstract void buildPartB();
    public abstract void buildPartC();
    //  返回产品对象
    public Product getResult(){
        return product;
    }
```

```
}

public class Product {
    private String partA;
    private String partB;
    private String partC;
}
```

客户端代码

```
Director director = new Director(new ConcreateBuilder());
Product product = director.construct();
```

- ## 优点

1. 分解了产品对象的创建和表示，客户端无需关心对象内部组成细节，只关注对象本身的使用
2. 将对象的创建过程一步一步分解开来，更加精细化的控制对象的创建步骤
3. 相同的创建过程得到不同的产品对象

- ## 缺点

1. 产品对象需要内部组成相似，差异不能太大，否则不适用

原型模式Prototype

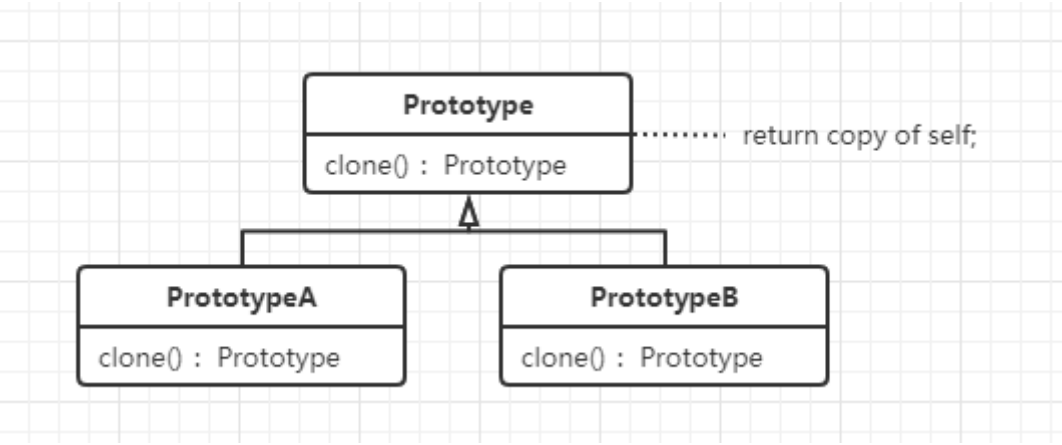
- ## 定义

对一个已有实例进行克隆，得到一个新对象进行使用。借助于java中的clone()和clonable接口可以更快捷的实现原型模式。克隆有浅克隆和深克隆区别。

- ## 应用场景

OA中的日报填写，新的日报实例和昨天的日报内容相差不多，可以直接进行克隆昨天的日报实例。

- ## UML类图



- ## 代码示例

```
public class Prototype {
    private String attr;

    public Prototype clone(){
        Prototype prototype = new Prototype();
        prototype.setAttr(this.attr);
        return prototype;
    }

    public void setAttr(String attr) {
        this.attr = attr;
    }
}
```

客户端代码

```
Prototype prototype = new Prototype();
prototype.setAttr("abc");
Prototype clone = prototype.clone();
```

- ## 优点

1. 如果创建一个新的对象较为复杂，开销较大，则通过复制一个已有对象得到一个新的对象，可以提供创建效率，不必关注对象的创建过程。

- ## 缺点

1. 当对象需要进行深克隆时，对象存在嵌套引用时，要求每一层对象都要支持克隆，实现起来麻烦。

结构型模式

关注于如何将一组类或者对象组织在一起形成更大的结构。

研究对象的结构、关联关系、继承、依赖关系等，影响程序的扩展性，耦合性，可维护性。

适配器模式Adapter

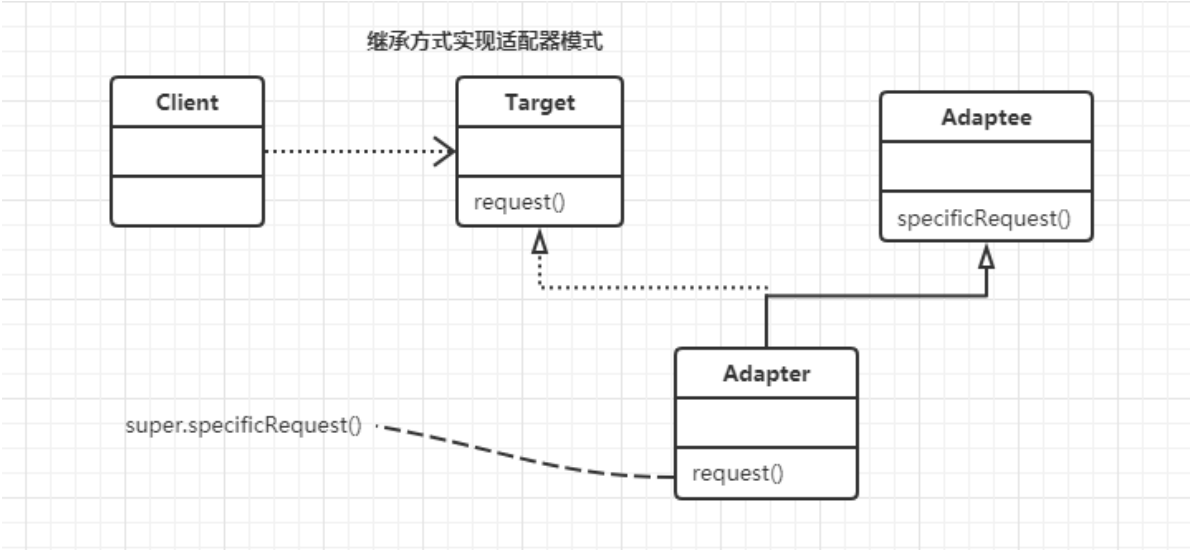
- ## 定义

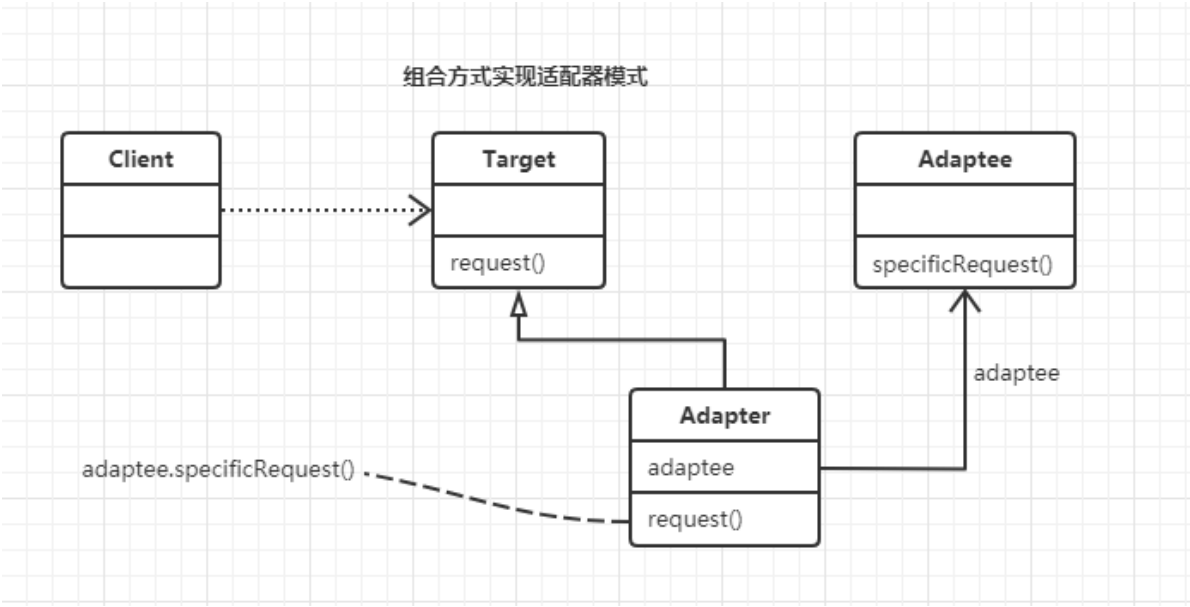
将一个接口转换为另一个客户希望的接口。满足让接口不兼容的类一起工作。

- ## 应用场景

数据线转接头

- ## UML类图





- **代码示例**

一、适配器继承被适配者方式实现适配器模式

```
//  适配器继承被适配者方式实现适配器模式
public class Adapter extends Adaptee implements Target {
    //  将对目标对象的请求转换为被适配者的请求
    public void request() {
        super.specificRequest();
    }
}

public class Adaptee {
    void specificRequest(){
        System.out.println("我是被适配者");
    }
}

public interface Target {
    void request();
}
```

客户端代码

```
public class Client {
    public static void main(String[] args) {
        Target target = new Adapter();
        target.request();
    }
}
```

二、适配器与被适配者进行组合的方式实现适配器模式

```
//  适配器与被适配者进行组合的方式实现适配器模式
public class Adapter implements Target {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    //  将对目标对象的请求转换为被适配者的请求
    public void request() {
        adaptee.specificRequest();
    }
}

public class Adaptee {
    void specificRequest(){
        System.out.println("我是被适配者");
    }
}

public interface Target {
    void request();
}
```

客户端代码

```
public class Client {
    public static void main(String[] args) {
        Target target = new Adapter(new Adaptee());
        target.request();
    }
}
```

- **优点**

1. 通过适配器类，将目标类和被适配者类进行解耦，扩展性好

- **缺点**

1. 对于继承实现的适配器模式，由于基于继承，而java不可以多重继承，所以适配器一次只能适配一种被适配者（适配器类需继承被适配类）
2. 对于继承实现的适配器模式，目标类必须为接口（如果为类，适配器类已经继承被适配者，无法再继承目标类），局限性较大。

桥接模式Bridge

- **定义**
- **应用场景**
- **UML类图**
- **代码示例**

客户端代码

- **优点**
- **缺点**

组合模式Composite

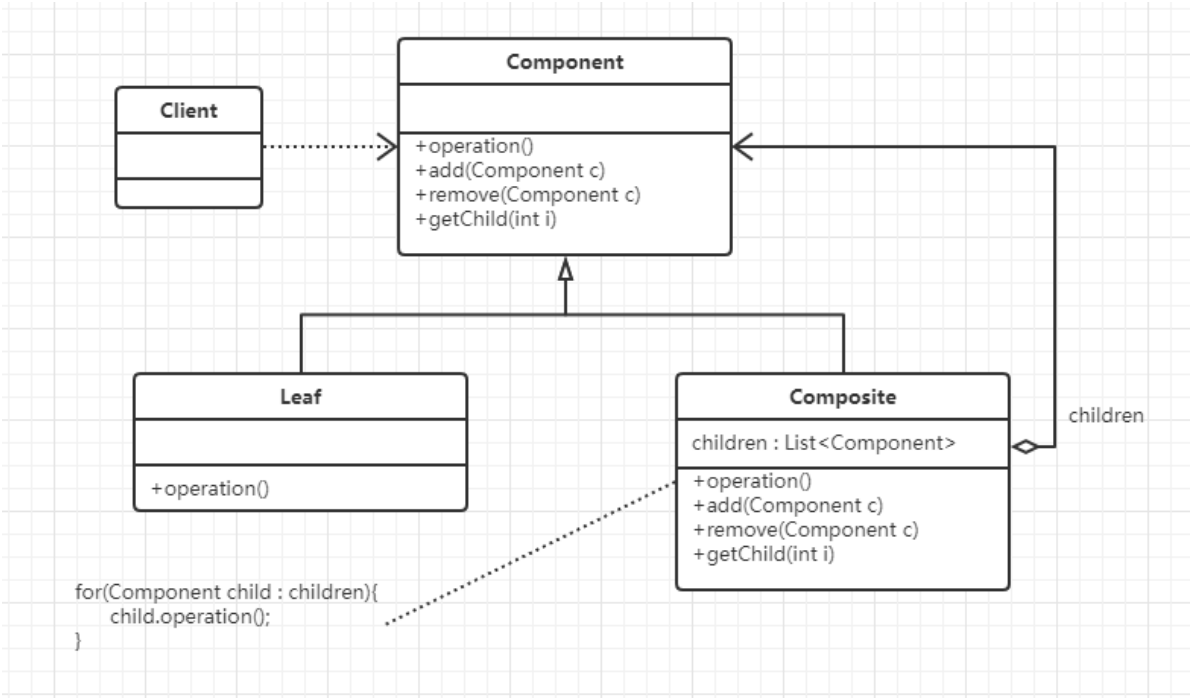
- **定义**

组合多个对象形成树形结构，部分-整体的层次关系。使得客户端可以统一的对待单个对象和组合对象。

- **应用场景**

计算合并订单的金额，合并订单>子订单>商品条目；文件目录结构

- **UML类图**



- **代码示例**

```
public abstract class Component {
    abstract void add(Component c);
    abstract void remove(Component c);
    abstract Component get(int i);
    abstract void operation();
}

public class Leaf extends Component {
    void add(Component c) {
        //  异常处理或者错误提示
    }

    void remove(Component c) {
        //  异常处理或者错误提示
    }

    Component get(int i) {
        //  异常处理或者错误提示
        return null;
    }

    void operation() {
        //  叶子构件具体业务方法实现
    }
}

public class Composite extends Component {

    List<Component> children = new ArrayList<Component>();

    void add(Component c) {
        children.add(c);
    }

    void remove(Component c) {
        children.remove(c);
    }

    Component get(int i) {
        return children.get(i);
    }

    //  容器构件具体业务方法实现，将递归调用成员构件的业务方法
    void operation() {
        for (Component child : children){
            child.operation();
        }
    }
}
```

客户端代码

略

- **优点**

1. 清晰的定义了树形结构的分层次的复杂对象
2. 客户端操作简单，无需关心操作的是叶子对象还是组合对象
3. 动态添加叶子对象和组合对象容易

- **缺点**

装饰模式Decorator

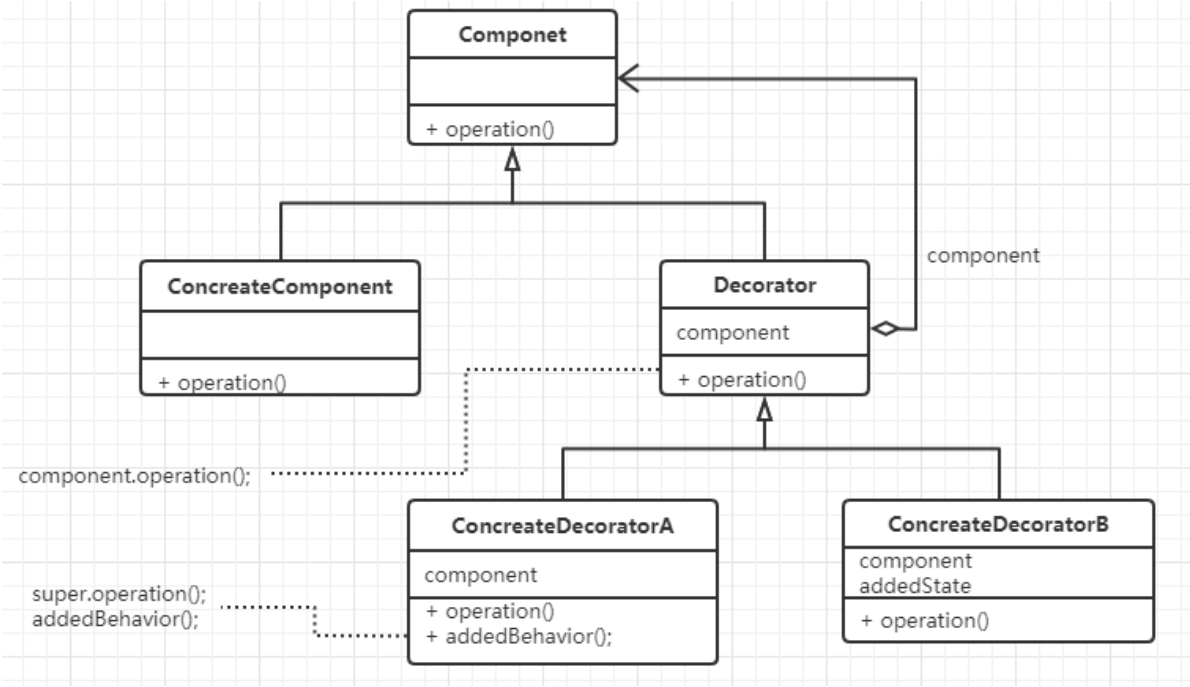
- **定义**

动态给一个对象扩展一些其他职责

- **应用场景**

java的IO流

- **UML类图**



- **代码示例**

```
public interface Component {
    void operation();
}

public class ConcreateComponent implements Component {
    public void operation() {
        System.out.println("被装饰者对象ConcreateComponent执行operation方法");
    }
}

public class Decorator implements Component {
    //  持有一个对被装饰对象的引用
    Component component;
```

```
    //  通过构造函数注入被装饰对象
    public Decorator(Component component) {
        this.component = component;
    }

    //  调用被装饰对象的原业务方法
    public void operation() {
        component.operation();
    }
}

public class ConcreateDecoratorA extends Decorator {
    public ConcreateDecoratorA(Component component) {
        super(component);
    }

    @Override
    public void operation() {
        //  调用原有业务方法
        super.operation();
        //  调用新增业务方法，完成额外职责的添加
        addedBehaviorA();
    }

    //  额外职责
    void addedBehaviorA(){
        System.out.println("这是为被装饰者对象添加的额外行为addedBehaviorA");
    }
}

public class ConcreateDecoratorB extends Decorator {
    public ConcreateDecoratorB(Component component) {
        super(component);
    }

    @Override
    public void operation() {
        //  调用原有业务方法
        super.operation();
        //  调用新增业务方法，完成额外职责的添加
        addedBehaviorB();
    }

    //  额外职责
    void addedBehaviorB(){
        System.out.println("这是为被装饰者对象添加的额外行为addedBehaviorB");
    }
}
```

客户端代码

```
Decorator decorator1 = new ConcreateDecoratorA(new ConcreateComponent());
decorator1.operation();

Decorator decorator2 = new ConcreateDecoratorB(decorator1);
decorator2.operation();
```

- ## 优点

1. 组合比继承更灵活，扩展性强
2. 可以按需添加新的被装饰类和装饰者类，已完成新的需要，而无需修改原有类库，符合开闭原则
3. 可以对一个被装饰者进行多次装饰，通过不同排列组合获得不同的行为。

- ## 缺点

1. 多次装饰的对象，出错时排查麻烦

外观模式Facade

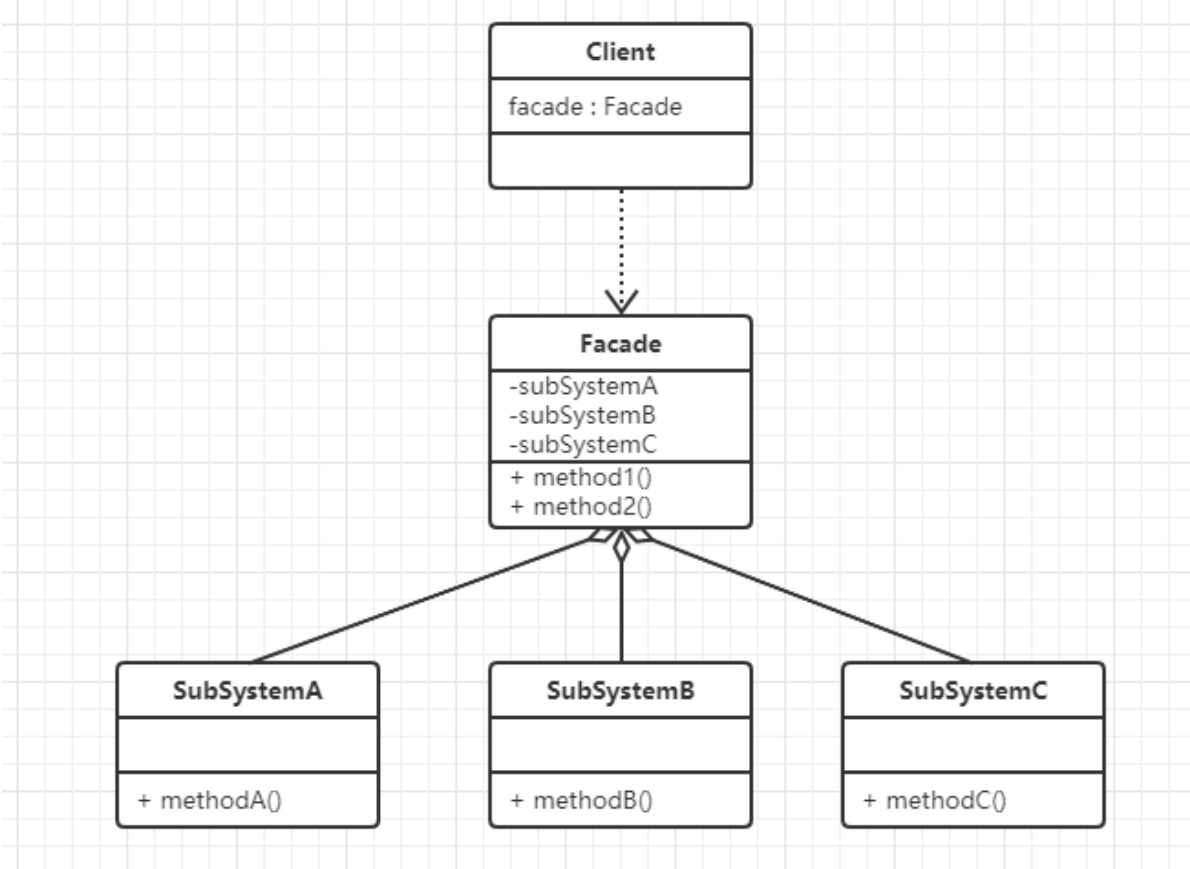
- ## 定义

定义一个高层接口，给一组接口提供一个统一的入口，使得内部接口更容易使用

- ## 应用场景

spring jdbcUtil类，slf4j的日志类

- ## UML类图



- ## 代码示例

```
public class Facade {
    private SubSystemA subSystemA = new SubSystemA();
    private SubSystemB subSystemB = new SubSystemB();
    private SubSystemC subSystemC = new SubSystemC();

    void method1(){
```

```
        subSystemA.methodA();
        subSystemB.methodB();
    }

    void method2(){
        subSystemA.methodA();
        subSystemB.methodB();
        subSystemC.methodC();
    }
}

public class SubSystemA {
    void methodA(){}
}

public class SubSystemB {
    void methodB(){}
}

public class SubSystemC {
    void methodC(){}
}
```

客户端代码

```
Facade facade = new Facade();
facade.method1();
facade.method2();
```

- **优点**

1. 子系统和客户端代码松耦合，子系统的改变不会影响到客户端，只会影响到外观类
2. 简化了客户端代码，客户端无需关心子系统的内部细节等，只使用外观类即可

- **缺点**

1. 子系统改变，可能需要改变外观类，违背开闭原则

享元模式Flyweight

- **定义**

通过共享技术支持大量细粒度对象的复用

- **应用场景**
- **UML类图**

- **代码示例**

客户端代码

- **优点**
- **缺点**

代理模式Proxy

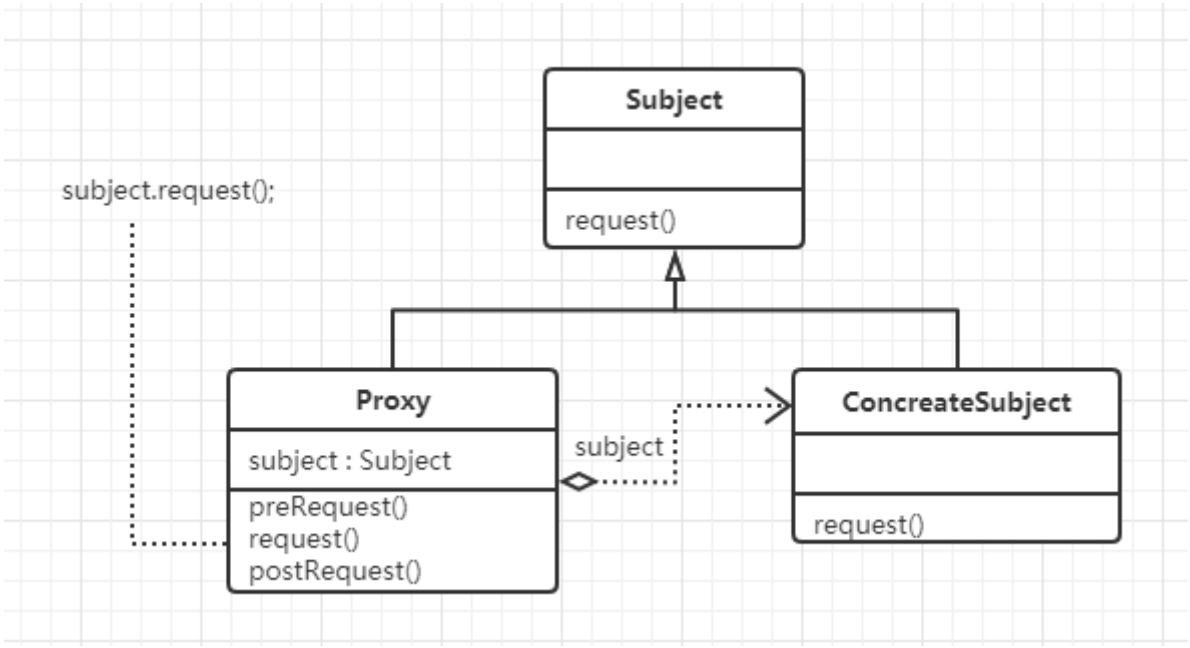
- **定义**

给一个对象提供一个代理或者占位符，并通过代理对象控制对原对象的访问

- **应用场景**

jdk动态代理，spring cglib代理，AOP技术

- **UML类图**



- **代码示例**

```
public interface Subject {
    void request();
}

public class RealSubject implements Subject {
    public void request() {

    }
}

public class Proxy implements Subject {

    private Subject subject;

    public Proxy(Subject subject) {
        this.subject = subject;
```

```
    }

    public void request() {
        preRequest();
        subject.request();  //  请求被代理类的业务方法
        postRequest();
    }

    private void preRequest(){}
    private void postRequest(){}
}
```

客户端代码

```
Proxy proxy = new Proxy(new RealSubject());
proxy.request();
```

- ## 优点

1. 松耦合
2. 增加代理类，无需改变原来的主题类库，符合开闭原则
3. 代理类可以控制对主题类（目标类）的访问

- ## 缺点

1. 可能影响性能

行为型模式

定义系统内对象间的通信，以及复杂程序中的流程控制。

设计各个对象的行为，提高对象之间的协作效率。

责任链模式COR(Chain Of Responsibility)

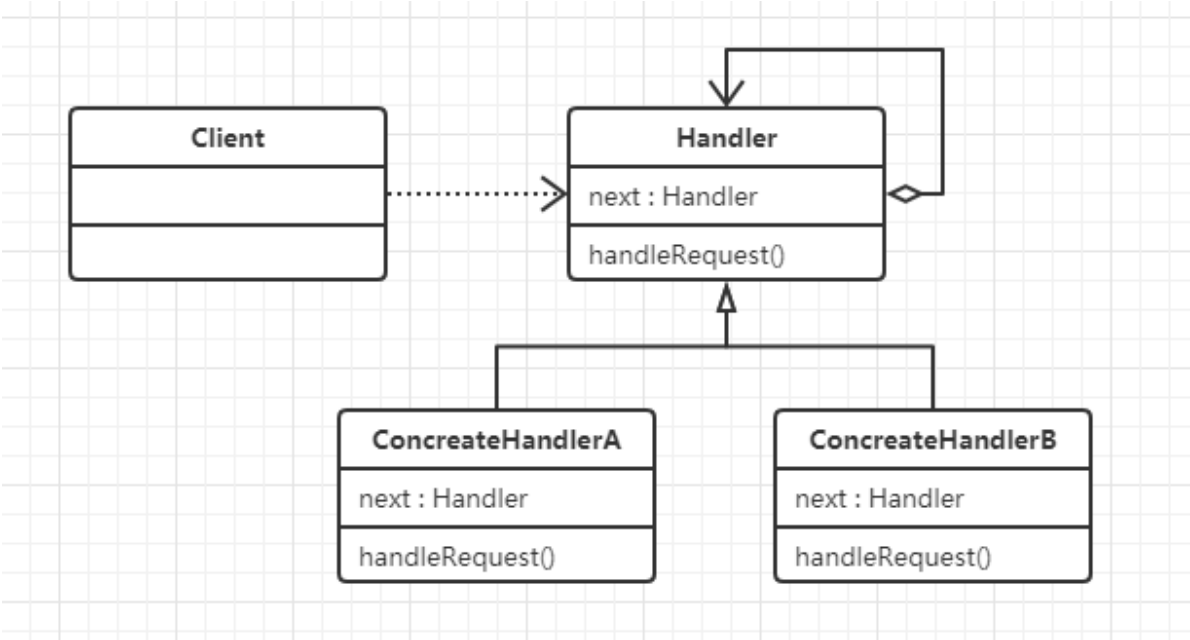
- ## 定义

将处理请求的对象连接成一条链，沿着这条链传递请求

- ## 应用场景

filter过滤器的设计;采购审批场景

UML类图



- **代码示例**

```
public abstract class Handler {
    protected Handler next;

    public void setNext(Handler next) {
        this.next = next;
    }

    public abstract void handleRequest(String request);
}

public class ConcreateHandlerA extends Handler {

    public void handleRequest(String request) {
        if(true){
            //  处理请求
        } else {
            next.handleRequest(request);
        }
    }
}

public class ConcreateHandlerB extends Handler {

    public void handleRequest(String request) {
        if(true){
            //  处理请求
        } else {
            next.handleRequest(request);
        }
    }
}
```

客户端代码

```
Handler handlerA = new ConcreateHandlerA();
Handler handlerB = new ConcreateHandlerB();
//  创建责任链
handlerA.setNext(handlerB);
handlerB.setNext(null);
//  发送请求
handlerA.handleRequest("request");
```

- ## 优点

1. 解耦了请求的接受者和发送者，中间只关注请求本身
2. 添加新的handler容易，符合开闭原则
3. 每个handler只需要持有nextHandler的引用即可
4. 可动态组合，形成不同的责任链

- ## 缺点

1. 责任链过长，系统性能影响较大
2. 避免死循环责任链，建链过程中next指向本身导致

命令模式command

定义

应用场景

UML类图

代码示例

客户端代码

优点

缺点

解释器模式interpreter

定义

应用场景

UML类图

代码示例

客户端代码

优点

缺点

迭代器模式iterator

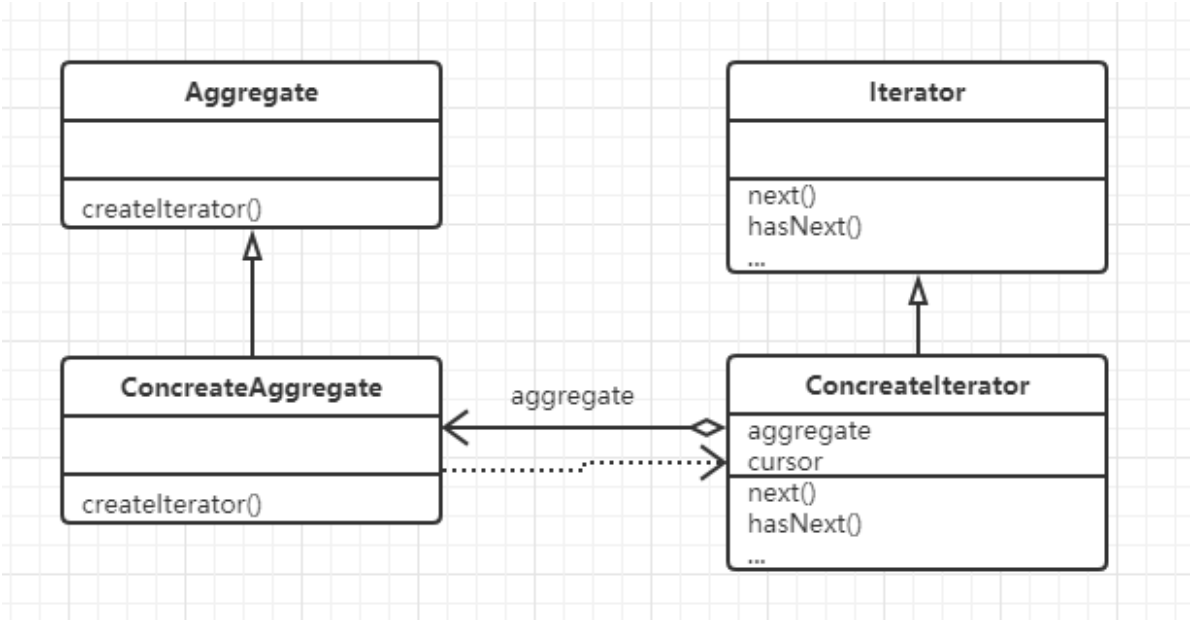
- **定义**

提供一种遍历聚合对象内部元素的方法，且不暴露内部结构。

- **应用场景**

java的迭代器

- **UML类图**



- **代码示例**

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

public class ConcreateAggregate implements Aggregate {
    //  其他属性和方法

    //  获取当前聚合对象的迭代器对象
    public Iterator createIterator() {
        return new ConcreateIterator();
    }
}

public interface Iterator {
    Object next();
    boolean hasNext();
}

public class ConcreateIterator implements Iterator {
    //  持有一个对聚合对象的引用，以便访问存储在聚合对象中的元素
    private Aggregate aggregate;
    //  定义一个游标，记录当前访问的位置
    private int cursor;

    public Object next() {
        return null;
    }
```

```
    public boolean hasNext() {
        return false;
    }
}
```

客户端代码

```
Aggregate aggregate = new ConcreateAggregate();
Iterator iterator = aggregate.createIterator();
while (iterator.hasNext()){
    Object obj = iterator.next();
    //...
}
```

- ## 优点

1. 将数据的存储和遍历职责分离，满足职责单一化原则
2. 提供不同的迭代器类可以满足不同的遍历聚合对象的要求

- ## 缺点

1. 迭代器类和聚合类成对增加，增加了系统复杂性

中介者模式mediator

定义

应用场景

UML类图

代码示例

客户端代码

优点

缺点

备忘录模式memento

定义

应用场景

UML类图

代码示例

客户端代码

优点

缺点

观察者模式observer

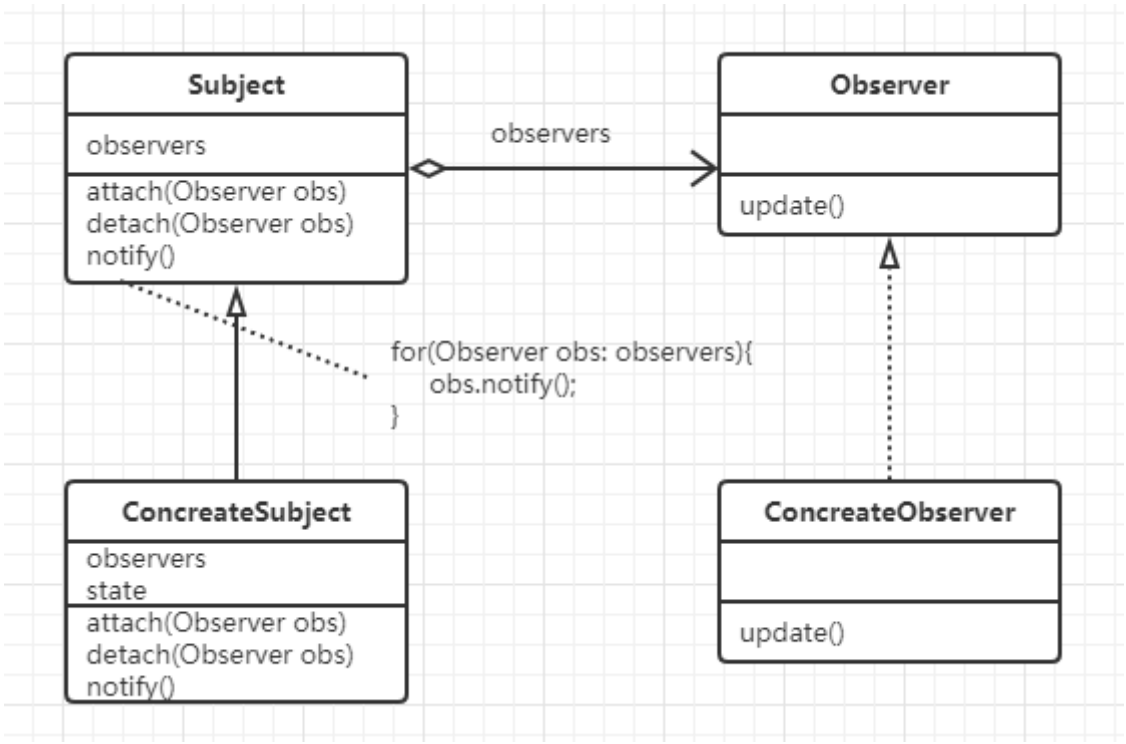
- **定义**

一个对象状态发生改变，通知其他依赖对象进行自动更新。

- **应用场景**

消息中间件；spring事件-监听器机制

- **UML类图**



- **代码示例**

```
public abstract class Subject {
    //  观察者列表
    protected List<Observer> observers;
    //  添加到目标的观察者集合中
    abstract void attach(Observer observer);
    //  从目标的观察者集合中删除
    abstract void detach(Observer observer);
    //  通知目标的所有观察者
    abstract void notifyObservers();
}

public interface Observer {
    //  观察者被通知时执行的业务方法
    void update(Object state);
}

public class ConcreateSubject extends Subject {
    //  表示目标的状态
    private int state;

    void attach(Observer observer) {
        observers.add(observer);
    }
```

```
    void detach(Observer observer) {
        observers.remove(observer);
    }

    void notifyObservers() {
        for (Observer observer:observers){
            observer.update(state);
        }
    }
}

public class ConcreateObserver implements Observer {
    public void update(Object obj) {}
}
```

客户端代码

```
Subject subject = new ConcreateSubject();
Observer observer = new ConcreateObserver();
//  将observer添加到subject的观察者列表中
subject.attach(observer);
//  subject的状态state发生变化
subject.notifyObservers();
```

- ## 优点

1. 简化一对多关联的系统设计
2. 目标对象和观察者对象属于不同的抽象层次，目标对象只需维持一个抽象观察者的集合，松耦合设计。
3. 增加观察者类不影响原有类库，符合开闭原则

- ## 缺点

1. 避免目标和观察对象之间的循环依赖，导致死循环问题

状态模式state

- ## 定义

一个对象的内在状态改变时改变了其行为。对象似乎改变了它的类。

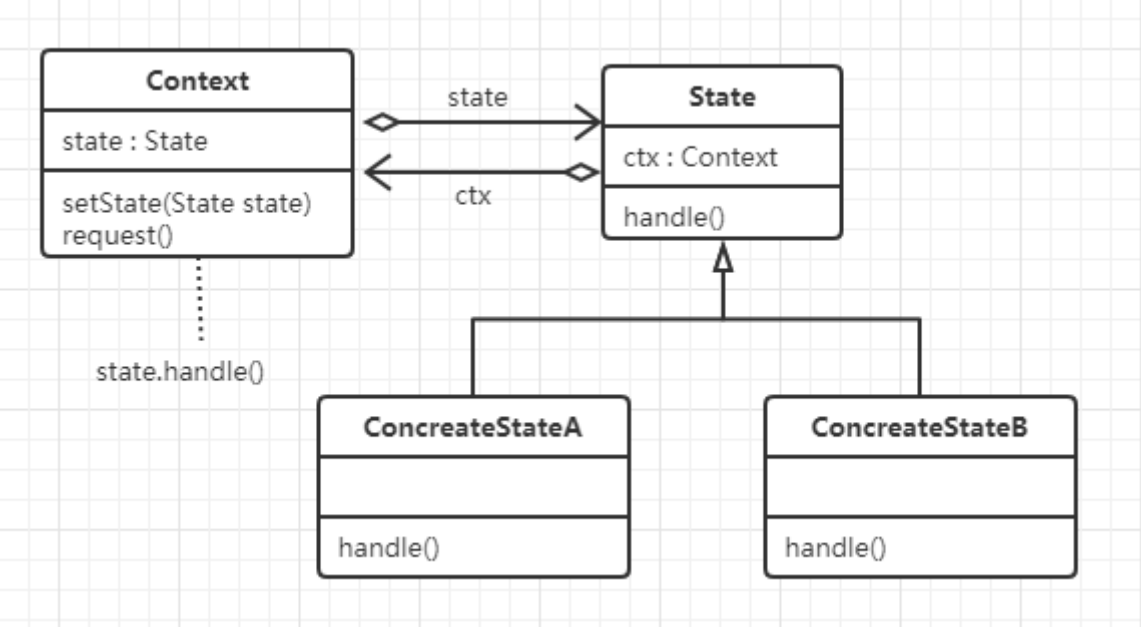
释义：

1. 有一个对象，它是有状态的
2. 状态改变时，行为也会发生变化
3. 状态之间是会自动转换的，状态机

- ## 应用场景

信用卡账户，账户在不同状态（正常、管制、锁定）下可以支持的行为（消费，取现）不同。

- **UML类图**



- **代码示例**

```
//  环境类（存在状态）
public class Context {
    //  表示类的状态，用于接收外界的请求委派给state，并在state内部实现状态的切换
    private State state;

    public void setState(State state) {
        this.state = state;
    }

    void request(){
        state.handle();
    }
}

public abstract class State {
    //  持有一个环境类的引用，用于切换环境对象的状态
    protected Context ctx;
    abstract void handle();
}

public class ConcreateStateA extends State {

    public void handle() {
        System.out.println("行为A");
        //  状态切换到B
        ctx.setState(new ConcreateStateB());
    }
}

public class ConcreateStateB extends State {
    public void handle() {
        System.out.println("行为B");
        //  状态切换到null,可以定义一个终止状态，这里躲懒
        ctx.setState(null);
    }
```

```
}
```

客户端代码

```
Context ctx = new Context();
ctx.setState(new ConcreateStateA());
ctx.request();
```

- **优点**

1. 状态对象封装了状态和该状态下不同的对象行为
2. 添加新的状态类，即可让环境类拥有新的状态下的新的行为
3. 状态转换规则可以统一集中到环境类进行处理，也可以分别放到具体的状态类中处理，避免将状态转换逻辑和业务代码耦合，容易管理

- **缺点**

1. 状态机设计出错时，可能死循环
2. 添加新的状态时，会涉及到修改原有的状态类的行为（因为存在状态机的迁移问题），不符合开闭原则

策略模式strategy

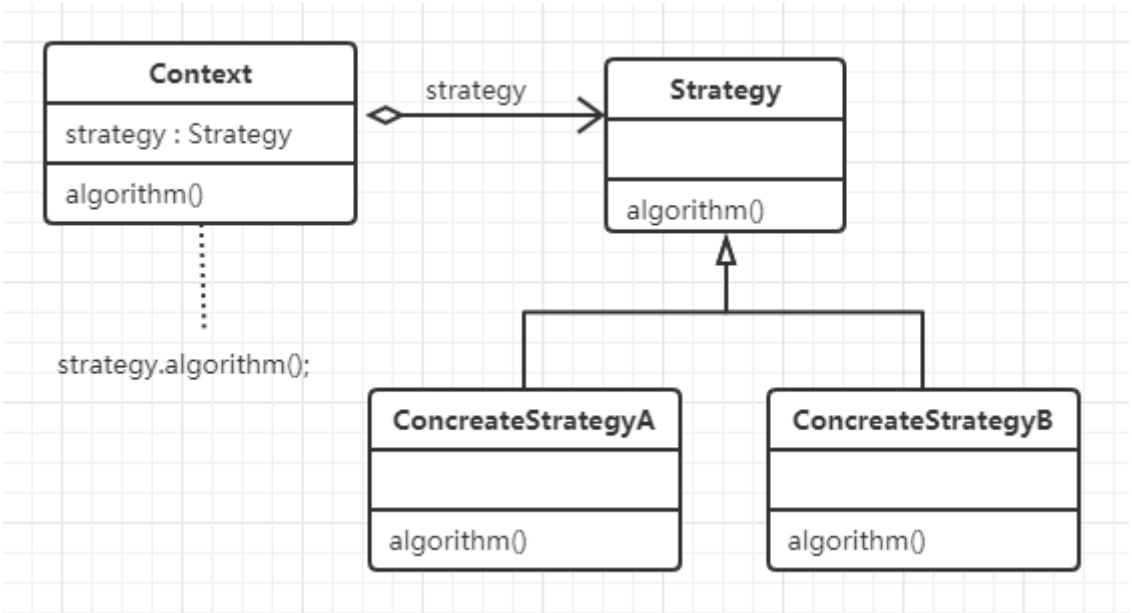
- **定义**

定义算法族，算法之间是可以相互替代的。

- **应用场景**

加密场景，不同加密算法

- **UML类图**



- **代码示例**

```
public class Context {
    //  具体的策略
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    void algorithm(){
        strategy.algorithm();
    }
}

public interface Strategy {
    void algorithm();
}

public class ConcreateStrategyA implements Strategy {
    public void algorithm() {
        System.out.println("策略A...");
    }
}

public class ConcreateStrategyB implements Strategy {
    public void algorithm() {
        System.out.println("策略B...");
    }
}
```

客户端代码

```
//  客户端需要选择某个策略
Context ctx = new Context(new ConcreateStrategyA());
ctx.algorithm();
```

- **优点**

1. 添加新的算法，不用修改原有的类库，符合开闭原则
2. 解决if...else...问题，扩展性强，可维护性强，解决了选择使用的算法和算法本身的分离，符合职责单一原则
3. 封装了算法，达到了算法的复用

- **缺点**

1. 客户端需要知道何时选择用哪种策略，客户需要知道策略之间的区别
2. 客户只可以选择一种策略执行，策略与策略之间是互相替换的，不可以协同工作

模板方法模式TemplateMethod

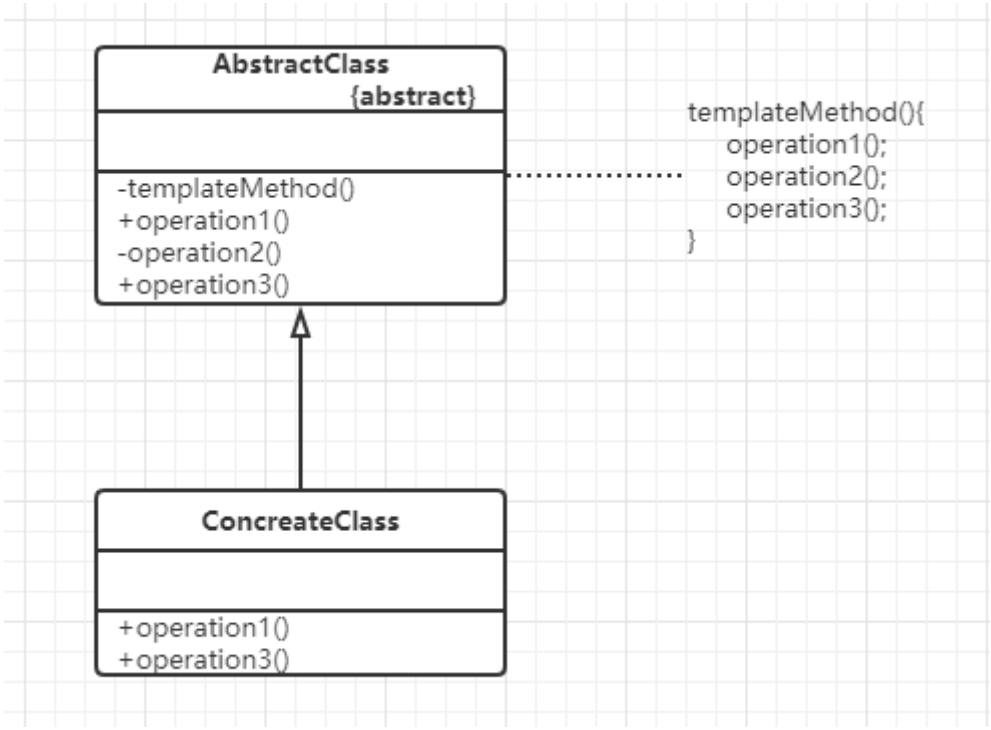
- **定义**

定义了一个算法的框架，其中某些步骤可以延迟到子类中。使得在不改变算法结构的前提下，不同的子类可以重新定义特定步骤的实现。

- **应用场景**

对账文件解析

- **UML类图**



- **代码示例**

```
public abstract class TemplateClass {
    //  模板方法定义为final,避免子类重写
    public final void templateMethod(){
        operation1();
        operation2();
        operation3();
    }

    public abstract void operation1();
    public void operation2(){
        //  共享的方法
    }
    public abstract void operation3();
}

public class ConcrateTemplate extends TemplateClass {
    //  子类对算法结构特定步骤的个性化实现
    public void operation1() {}
    public void operation3() {}
}
```

客户端代码

```
TemplateClass templateClass = new ConcrateTemplate();
//  请求模板方法
templateClass.templateMethod();
```

- **优点**

1. 父类定义算法结构，子类实现特定步骤的细节，不会改变算法结构，符合开闭原则
2. 共用的基本方法在父类声明，提高复用
3. 利用钩子方法，实现子类对父类的反向控制，子类控制父类的算法结构某些特定步骤是否能够执行

- **缺点**

1. 子类代码的执行结果影响了父类

访问者模式visitor

定义

应用场景

UML类图

代码示例

客户端代码

优点

缺点