结构型设计模式

结构型设计模式主要总结了一些类与对象组合在一起的景点接口,其中模式包括:适配器模式,桥接模式,装饰器模式,组合模式,门面模式,享元模式,代理模式

适配器模式

基本介绍:比如现实中手机充电,插座是220V的电压,通过充电器的适配之后,转换成了5V的电压给手机充电,适配器模式其实就是**将不兼容的接口转换为可兼容的接口**

类适配器(基于继承)

其中Adaptee属于被适配者,目的是转换成ITarget接口,这时声明一个Adaptor类,继承Adaptee类并实现ITarget接口,根据具体的业务需求,将被适配者适配成我们需要的接口即可.

```
public interface ITarget {
void f1();
void f2();
void fc();
}
public class Adaptee {
public void fa() { //... }
public void fb() { //... }
public void fc() { //... }
}
public class Adaptor extends Adaptee implements ITarget {
public void f1() {
super.fa();
}
public void f2() {
//...重新实现f2()...
}
// 这里fc()不需要实现，直接继承自Adaptee，这是跟对象适配器最大的不同点
}
```

对象适配器(基于聚合)

对象适配器和类适配器的差异就是将继承关系转换为聚合关系,**根据合成复用原则,尽量使用关联关系代替继承关系**,

```
public interface ITarget {
void f1();
void f2();
void fc();
}
public class Adaptee {
public void fa() { //... }
public void fb() { //... }
public void fc() { //... }
}
public class Adaptor implements ITarget {
private Adaptee adaptee;
```

```
public Adaptor(Adaptee adaptee) {
this.adaptee = adaptee;
}
public void f1() {
adaptee.fa(); //委托给Adaptee
}
public void f2() {
//...重新实现f2()...
}
public void fc() {
adaptee.fc();
}
}
```

接口适配器(缺省适配器)

当不需要全部实现接口提供的方法时,先设计一个抽象类实现接口,并为该接口每个方法提供一个默认实现(空方法),那么该抽象类的子类可以有选择的覆盖父类的某些方法来实现需求

```
public interface ITarget {
void f1();
void f2();
void fc();
}
public abstract class AbsTarget{
void f1(){};
void f2(){};
void fc(){};
}

public class client{
    public static void main(String[] args){
        AbsTarget absTarget = new AbsTarget{
            @Override
            public void f1(){
                //todo
            }
        }
    }
}
```

适配器模式的使用场景

如果 Adaptee 接口很多，而且 Adaptee 和 ITarget 接口定义大部分都相同，推荐使用类适配器，因为 Adaptor 复用父类 Adaptee 的接口，比起对象适配器的实现方式，Adaptor 的代码量要少一些。

如果 Adaptee 接口很多，而且 Adaptee 和 ITarget 接口定义大部分都不相同，那我们推荐使用对象适配器，因为组合结构相对于继承更加灵活。

封装有缺陷的接口设计

比如一个外部接口设计方面有缺陷,我们希望对这个接口进行一个二次封装,就可以使用适配器模式

```
public class CD { //这个类来自外部sdk，我们无权修改它的代码
    //...
    public static void staticFunction1() { //... }
```

```
    public void uglyNamingFunction2() { //... }
    public void tooManyParamsFunction3(int paramA, int paramB, ...) { //... }
    public void lowPerformanceFunction4() { //... }
    }
// 使用适配器模式进行重构
public class ITarget {
    void function1();
    void function2();
    void fucntion3(ParamsWrapperDefinition paramsWrapper);
    void function4();
    //...
}
// 注意：适配器类的命名不一定非得末尾带Adaptor
public class CDAdaptor extends CD implements ITarget {
    //...
    public void function1() {
    super.staticFunction1();
    }
    public void function2() {
    super.uglyNamingFucntion2();
    }
    public void function3(ParamsWrapperDefinition paramsWrapper) {
    super.tooManyParamsFunction3(paramsWrapper.getParamA(), ...);
    }
    public void function4() {
    //...reimplement it...
    }
}
```

统一多个类的接口设计

比如我们有一个关键词过滤的接口分别来源A,B,C三个系统,传统代码进行过滤使用代码实现如下:

```
public class ASensitiveWordsFilter { // A敏感词过滤系统提供的接口
//text是原始文本，函数输出用***替换敏感词之后的文本
    public String filterSexyWords(String text) {
    // ...
    }
    public String filterPoliticalWords(String text) {
    // ...
    }
}
public class BSensitiveWordsFilter { // B敏感词过滤系统提供的接口
    public String filter(String text) {
    //...
    }
}
public class CSensitiveWordsFilter { // C敏感词过滤系统提供的接口
    public String filter(String text, String mask) {//...
    }
}
// 未使用适配器模式之前的代码：代码的可测试性、扩展性不好
public class RiskManagement {
    private ASensitiveWordsFilter aFilter = new ASensitiveWordsFilter();
    private BSensitiveWordsFilter bFilter = new BSensitiveWordsFilter();
    private CSensitiveWordsFilter cFilter = new CSensitiveWordsFilter();
    public String filterSensitiveWords(String text) {
```

```
        String maskedText = aFilter.filterSexyWords(text);
        maskedText = aFilter.filterPoliticalWords(maskedText);
        maskedText = bFilter.filter(maskedText);
        maskedText = cFilter.filter(maskedText, "***");
        return maskedText;
    }
}
```

使用适配器模式构造后:

```
public interface ISensitiveWordsFilter { // 统一接口定义
    String filter(String text);
}
public class ASensitiveWordsFilterAdaptor implements ISensitiveWordsFilter {
    private ASensitiveWordsFilter aFilter;
    public String filter(String text) {
        String maskedText = aFilter.filterSexyWords(text);
        maskedText = aFilter.filterPoliticalWords(maskedText);
        return maskedText;
    }
}
//...省略BSensitiveWordsFilterAdaptor、CSensitiveWordsFilterAdaptor...
// 扩展性更好，更加符合开闭原则，如果添加一个新的敏感词过滤系统，
// 这个类完全不需要改动；而且基于接口而非实现编程，代码的可测试性更好。
public class RiskManagement {
    private List<ISensitiveWordsFilter> filters = new ArrayList<>();
    public void addSensitiveWordsFilter(ISensitiveWordsFilter filter) {
        filters.add(filter);
    }
    public String filterSensitiveWords(String text) {
        String maskedText = text;
        for (ISensitiveWordsFilter filter : filters) {
            maskedText = filter.filter(maskedText);
        }
        return maskedText;
    }
}
```

替换原有接口

```
// 外部系统A
public interface IA {
//...
    void fa();
}
public class A implements IA {
//...
    public void fa() { //... }
}
// 在我们的项目中，外部系统A的使用示例
public class Demo {
    private IA a;
    public Demo(IA a) {
    this.a = a;
    }
    //...
```

```
    }
    Demo d = new Demo(new A());
// 将外部系统A替换成外部系统B
public class BAdaptor implemnts IA {
    private B b;
    public BAdaptor(B b) {
        this.b= b;
    }
    public void fa() {
        //...
        b.fb();
    }
}
    // 借助BAdaptor，Demo的代码中，调用IA接口的地方都无需改动，
    // 只需要将BAdaptor如下注入到Demo即可。
    Demo d = new Demo(new BAdaptor(new B()));
```

兼容老版本的接口

在做版本升级的时候，对于一些要废弃的接口，我们不直接将其删除，而是暂时保留，并且标注为deprecated，并将内部实现逻辑委托为新的接口实现。这样做的好处是，让使用它的项目有个过渡期，而不是强制进行代码修改。这也可以粗略地看作适配器模式的一个应用场景

适配不同格式的数据

比如一个主数据系统对接多个上游系统,各个系统的数据格式都不一样,我们需要对数据进行清洗转换成一个相同的格式.Java 中的 Arrays.asList() 也可以看作一种数据适配器，将数组类型的数据转化为集合容器类型

```
List<String> stooges = Arrays.asList("Larry", "Moe", "Curly");
```

springMVC中使用适配器模式

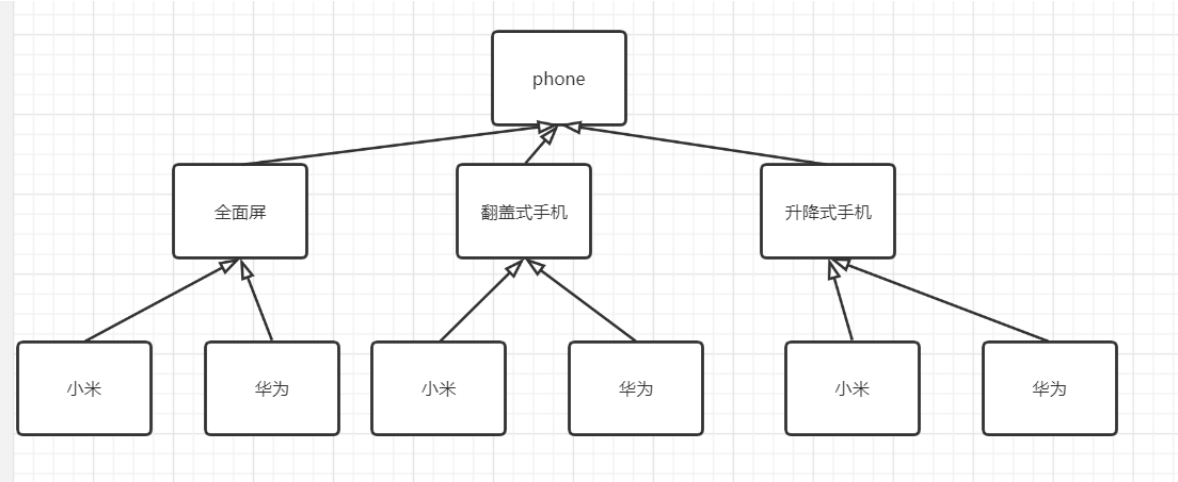
例如springMVC中的`org.springframework.web.servlet.DispatcherServlet#doDispatch`方法,其中去调用具体的handle方法就用到了适配器模式,先获取对应的controller,然后根据对应controller的类型获取具体的HandlerAdapter类型,最后通过HandlerAdapter调用执行具体的方法,这样就可以达到面向接口编程

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws Exception {
                // 获取对应的具体的controller类
                mappedHandler = getHandler(processedRequest);
                // Determine handler adapter for the current request.
                HandlerAdapter ha =
getHandlerAdapter(mappedHandler.getHandler());
                //使用具体的handle去调用controller类中具体的方法
                    mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());
}
```

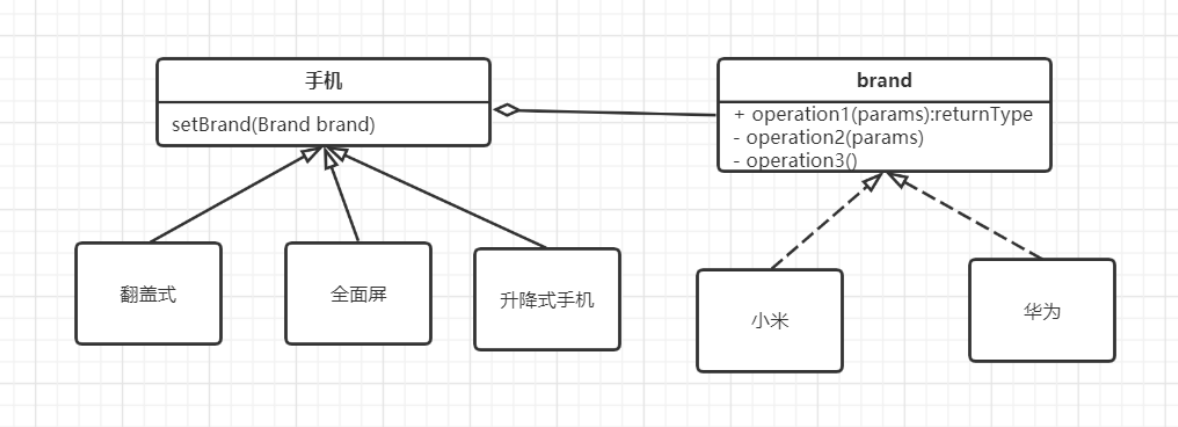
桥接模式

基本介绍:桥接模式的定义是**"将抽象和实现解耦，让它们可以独立变化","一个类存在两个（或多个）独立变化的维度，我们通过组合的方式，让这两个（或多个）维度可以独立进行扩展。"通过组合关系来替代继承关系，避免继承层次的指数级爆炸.**

例如我们需要使用一个手机,手机的类型有翻盖,全面屏,升降式手机,手机品牌也有多钟类型,xiaomi,huawei,apple等,如果正常情况下我们直接泛化继承的使用类图如:



这时候如果需要加一个品牌,那么需要增加三个类,导致扩展不便,如果层次过多的话,每加一个类都会导致类爆炸,所以引入了桥接模式,引入桥接模式之后的UML图:



这样将手机的样式与品牌给剥离出来,样式也是一个抽象,手机品牌是具体的实现,这样增加一个品牌只需要增加一个类即可.

JDBC中使用桥接模式的样例

使用简单的jdbc:

```
Class.forName("com.mysql.jdbc.Driver");//加载及注册JDBC驱动程序
String url = "jdbc:mysql://localhost:3306/sample_db?
user=root&password=your_pass";
Connection con = DriverManager.getConnection(url);
Statement stmt = con.createStatement();
String query = "select * from test";
ResultSet rs=stmt.executeQuery(query);
while(rs.next()) {
rs.getString(1);
rs.getInt(2);
}
```

这里如果我们将mysql切换至oracle,只需要将第一行和第二行的内容稍微改动一下就可以了,非常方便,这里也是使用了桥接模式,看看源码中的实现:

com.mysql.jdbc.Driver:这里就是Class.forName("com.mysql.jdbc.Driver") 将对应的驱动类加载及注册到DriverManager中

```
public class Driver extends NonRegisteringDriver implements java.sql.Driver {
    //
    // Register ourselves with the DriverManager
    //
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }
}
```

DriverManager类:

```
public class DriverManager {
private final static CopyOnWriteArrayList<DriverInfo> registeredDrivers = new
    //...
    static {
        loadInitialDrivers();
        println("JDBC DriverManager initialized");
    }
    //...
    public static synchronized void registerDriver(java.sql.Driver driver)
throws
        if (driver != null) {
            registeredDrivers.addIfAbsent(new DriverInfo(driver));
        } else {
         throw new NullPointerException();
        }
    }
    public static Connection getConnection(String url, String user, String
passwo
        java.util.Properties info = new java.util.Properties();
        if (user != null) {
            info.put("user", user);
        }
        if (password != null) {
            info.put("password", password);
        }
        return (getConnection(url, info, Reflection.getCallerClass()));
    }
//...
}
```

DriverManager将Driver组合起来,这样更换不同数据库的Driver之后,DriverManager执行所有的sql最终都被Driver的实现类给执行了,这样切换数据库的时候就很方便,不需要更改很多的代码

装饰器模式

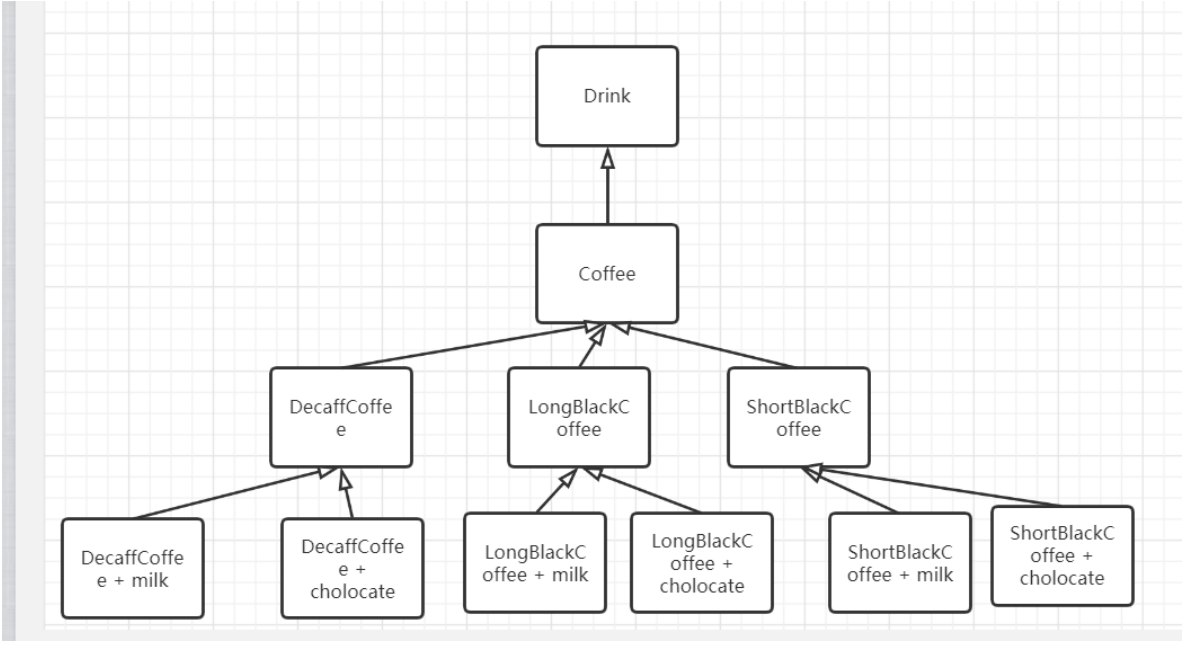
基本介绍:动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活

特点:装饰器类和原始类继承同样的父类，这样我们可以对原始类“嵌套”多个装饰器类,装饰器类是对功能的增强

订购咖啡实例

假如我们咖啡店里有多种咖啡,DecaffCoffee,LongBlackCoffee,ShortBlackCoffee,然后有多种配料,milk,Chocolate等

如果我们正常使用继承处理,需要创建很多个类:



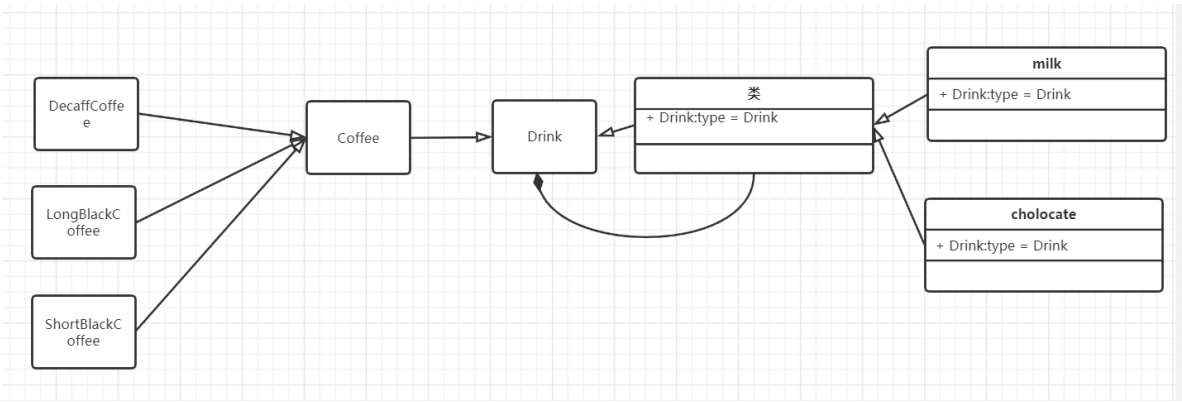
这种情况下,如果增加一个咖啡种类或者增加一个配料种类,都会导致类爆炸的问题,所以引入了装饰者模式

装饰者模式角色

装饰者就像包装一个快递,主体是快递物本身,装饰者给他包装一层又一层的保护措施

抽象主体:Compoent 主体,被包装的角色

抽象装饰者:Decorator,装饰主体的角色



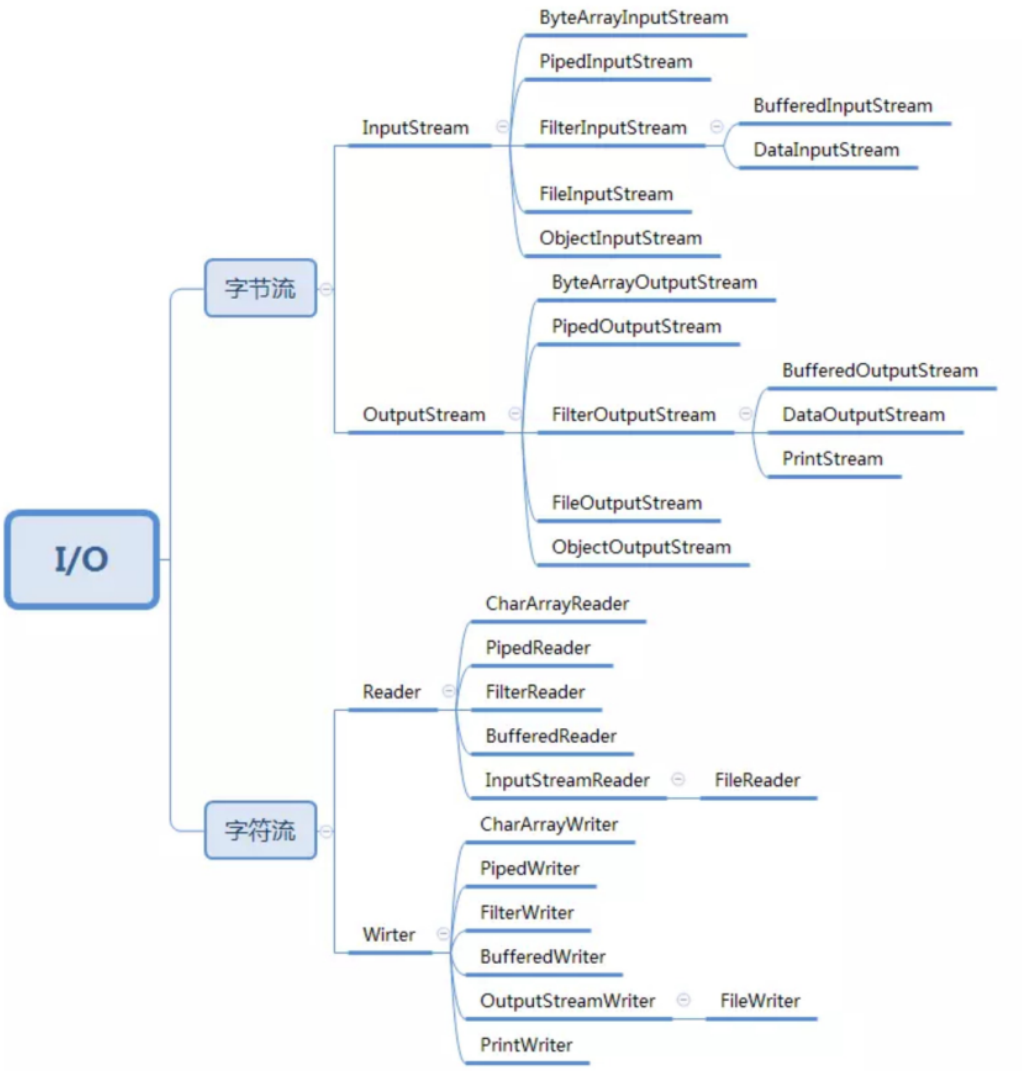
使用组合+继承的方式,装饰者继承主体并组合主体,这样就可以层层包装,达到动态给对象添加属性的功能,最终Client如下调用即可:

```
public static void main(String[] args) {
        // TODO Auto-generated method stub
        // 装饰者模式下的订单：2份巧克力+一份牛奶的LongBlack
```

```
        // 1. 点一份 LongBlack
        Drink order = new LongBlack();
        System.out.println("费用1=" + order.cost());
        System.out.println("描述=" + order.getDes());

        // 2. order 加入一份牛奶
        order = new Milk(order);

        System.out.println("order 加入一份牛奶 费用 =" + order.cost());
        System.out.println("order 加入一份牛奶 描述 = " + order.getDes());

        // 3. order 加入一份巧克力

        order = new Chocolate(order);

        System.out.println("order 加入一份牛奶 加入一份巧克力  费用 =" +
order.cost());
        System.out.println("order 加入一份牛奶 加入一份巧克力 描述 = " +
order.getDes());

        // 3. order 加入一份巧克力

        order = new Chocolate(order);

        System.out.println("order 加入一份牛奶 加入2份巧克力   费用 =" +
order.cost());
        System.out.println("order 加入一份牛奶 加入2份巧克力 描述 = " +
order.getDes());

        System.out.println("==========================");

        Drink order2 = new DeCaf();

        System.out.println("order2 无因咖啡  费用 =" + order2.cost());
        System.out.println("order2 无因咖啡 描述 = " + order2.getDes());

        order2 = new Milk(order2);

        System.out.println("order2 无因咖啡 加入一份牛奶  费用 =" + order2.cost());
        System.out.println("order2 无因咖啡 加入一份牛奶 描述 = " +
order2.getDes());


    }
```

装饰器模式在IO流中的应用

JDK中IO流的类图:



用IO流读取文件时,我们经常会用如下代码进行读取:

```
InputStream in = new FileInputStream("test.txt");
InputStream bin = new BufferedInputStream(in);
byte[] data = new byte[128];
while (bin.read(data) != -1) {
    //...
}
```

这里就是使用到了装饰器模式,打开FilterInputStream源码查看,FilterInputStream里面都包含有InputStream这个主体,并且FilterInputStream还继承了InputStream,那么FilterInputStream其实就是一个装饰者,在FilterInputStream下的子类,都是用于对InputStream增强的一些流,这样就达到了我们对多个主体进行动态多种增强的目的

```
public class FilterInputStream extends InputStream {
    /**
     * The input stream to be filtered.
     */
    protected volatile InputStream in;
}
```

组合模式

基本介绍:将一组对象组织（Compose）成树形结构，以表示一种"部分 - 整体"的层次结构。组合让客户端可以统一单个对象和组合对象的处理逻辑.

我们如果建立一个组织结构,例如:清华大学下有计算机学院和信息工程学院,计算机学院下面又有计算机科学与技术,软件工程,网络工程三个专业,信息工程学院下有通信工程,信息工程两个专业,如果我们想统一管理学院与专业的增减,遍历,就可以用上组合模式.

组合模式中的角色

1. Component:组合模式中对象声明的接口,在适当情况下,实现所有类公有的接口默认行为,用于管理子级
2. Leaf:最下级的叶子节点,叶子节点,叶子节点没有子节点(例子中的专业)
3. Composite:非叶子节点,实现接口中的增删查询动作(例子中的学校与学院)

组织结构管理实例代码:

```
//抽象一个公共Component类出来
public abstract class OrganizationComponent {
    private String name; // 名字
    private String des; // 说明
    protected  void add(OrganizationComponent organizationComponent) {
        //默认实现
        throw new UnsupportedOperationException();
    }
    protected  void remove(OrganizationComponent organizationComponent) {
        //默认实现
        throw new UnsupportedOperationException();
    }
    //构造器
    public OrganizationComponent(String name, String des) {
        super();
        this.name = name;
        this.des = des;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDes() {
        return des;
    }
    public void setDes(String des) {
        this.des = des;
    }
    //方法print, 做成抽象的, 子类都需要实现
    protected abstract void print();
}
//学校节点
//University 就是 Composite , 可以管理College
public class University extends OrganizationComponent {
    List<OrganizationComponent> organizationComponents = new
ArrayList<OrganizationComponent>();
    // 构造器
    public University(String name, String des) {
```

```
        super(name, des);
    }

    // 重写add
    @Override
    protected void add(OrganizationComponent organizationComponent) {
        organizationComponents.add(organizationComponent);
    }

    // 重写remove
    @Override
    protected void remove(OrganizationComponent organizationComponent) {
        organizationComponents.remove(organizationComponent);
    }

    @Override
    public String getName() {
        return super.getName();
    }

    @Override
    public String getDes() {

        return super.getDes();
    }

    // print方法，就是输出University 包含的学院
    @Override
    protected void print() {
        System.out.println("--------------" + getName() + "--------------");
        //遍历 organizationComponents
        for (OrganizationComponent organizationComponent :
organizationComponents) {
            organizationComponent.print();
        }
    }
}
//学院节点,基本与学校节点相同
public class College extends OrganizationComponent {

    //List 中 存放的Department
    List<OrganizationComponent> organizationComponents = new
ArrayList<OrganizationComponent>();

    // 构造器
    public College(String name, String des) {
        super(name, des);
    }

    // 重写add
    @Override
    protected void add(OrganizationComponent organizationComponent) {
        //  将来实际业务中，Colleage 的 add 和  University add 不一定完全一样
        organizationComponents.add(organizationComponent);
    }

    // 重写remove
    @Override
```

```
    protected void remove(OrganizationComponent organizationComponent) {
        organizationComponents.remove(organizationComponent);
    }

    @Override
    public String getName() {
        return super.getName();
    }

    @Override
    public String getDes() {
        return super.getDes();
    }

    // print方法，就是输出University 包含的学院
    @Override
    protected void print() {
        System.out.println("--------------" + getName() + "--------------");
        //遍历 organizationComponents
        for (OrganizationComponent organizationComponent :
organizationComponents) {
            organizationComponent.print();
        }
    }
}
//专业节点,即最终的叶子节点
public class Department extends OrganizationComponent {
    //没有集合
    public Department(String name, String des) {
        super(name, des);
    }


    //add , remove 就不用写了，因为他是叶子节点

    @Override
    public String getName() {
        return super.getName();
    }

    @Override
    public String getDes() {
        return super.getDes();
    }

    @Override
    protected void print() {
        System.out.println(getName());
    }
}
//客户端调用
    public static void main(String[] args) {

        //从大到小创建对象 学校
        OrganizationComponent university = new University("清华大学", " 中国顶级大
学 ");

        //创建 学院
```

```
        OrganizationComponent computerCollege = new College("计算机学院", " 计算机学院 ");
        OrganizationComponent infoEngineercollege = new College("信息工程学院", " 信息工程学院 ");


        //创建各个学院下面的系(专业)
        computerCollege.add(new Department("软件工程", " 软件工程不错 "));
        computerCollege.add(new Department("网络工程", " 网络工程不错 "));
        computerCollege.add(new Department("计算机科学与技术", " 计算机科学与技术是老牌的专业 "));

        //
        infoEngineercollege.add(new Department("通信工程", " 通信工程不好学 "));
        infoEngineercollege.add(new Department("信息工程", " 信息工程好学 "));

        //将学院加入到 学校
        university.add(computerCollege);
        university.add(infoEngineercollege);

        //university.print();
        infoEngineercollege.print();
    }
```

门面模式(外观模式)

基本介绍:门面模式为子系统提供一组统一的接口，定义一组高层接口让子系统更易用

假设我们有一个看电影的需求,首先需要打开DVD,放入光盘,打开电视,播放电影,关闭房间灯光,这五步动作,分别对应着DVD类,需要打开和放入光盘方法,电视类需要打开,播放电影方法,房间类需要打开灯光,关闭灯光方法.正常情况下如果我们使用客户端进行调用的话,需要如下代码

```
public static void main(String[] args) {
    Dvd dvd = new Dvd();
    dvd.open();
    dvd.sendCd();
    TV tv = new TV();
    tv.open();
    tv.play();
    Room room = new Room();
    room.closeLight();
}
```

但是这样调用的话,一旦接口过多或者是都是需要前端与后端进行交互时,如果需要调用Dvd,TV,Room三个接口,会造成不必要的网络消耗.

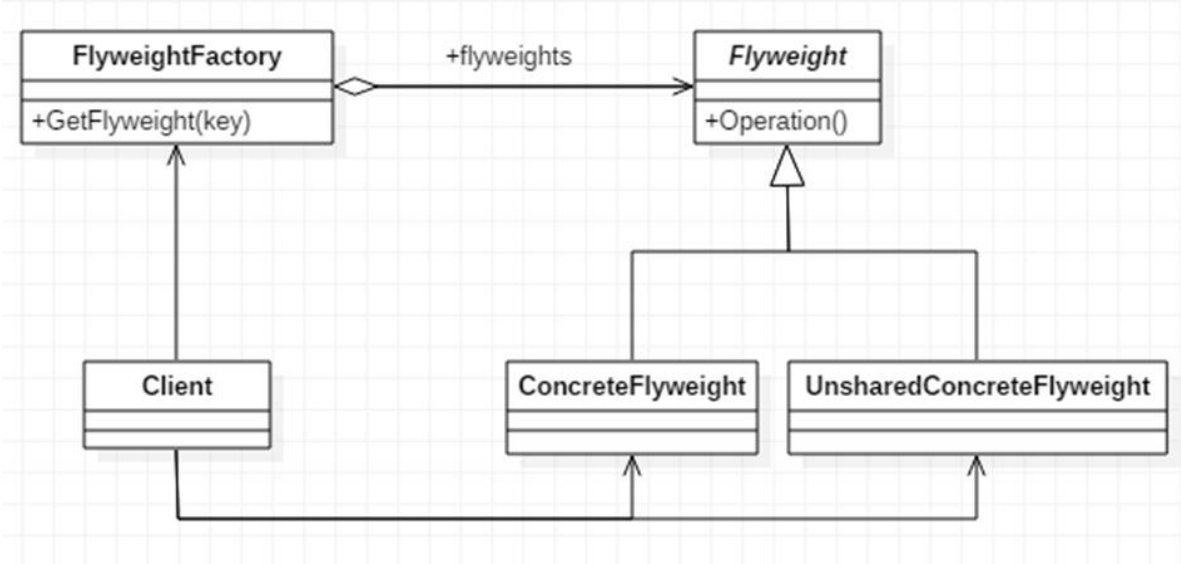
所以将他封装一个Facade类,提供多个子系统的统一接口,这样Client只需要和Facade类进行交互,不需要知道子系统实现的细节,提高复用性,但是使用外观模式还是直接调用具体模块,需要按照具体业务场景具体分析,最终都是要达到系统有层次并且便于维护的目的.

享元模式

基本介绍:享元模式的意图是用于减少创建对象的数量,复用对象，节省内存，

当一个系统中存在大量重复对象的时候，如果这些重复的对象是不可变对象，我们就可以利用享元模式将对象设计成享元，在内存中只保留一份实例，供多处代码引用。这样可以减少内存中对象的数量，起到节省内存的目的。实际上，不仅仅相同对象可以设计成享元，对于相似对象，我们也可以将这些对象中相同的部分（字段）提取出来，设计成享元，让这些大量相似对象引用这些享元.

享元模式中的角色



Flyweight:是一个抽象角色,就是我们所需要复用的对象的抽象类,例如棋子的抽象类,同时定义出对象的内部状态(**内部状态是不可变的,比如棋子的颜色,大小等**)和外部状态(**外部状态就是由用户来定义的,比如棋子的位置**).

ConcreteFlyweight:具体的享元对象角色,比如围棋棋子,象棋棋子

UnsharedConcreteFlyweight:不可共享的享元对象角色,一般不会出现

FlyweightFactory:工厂类,使用hashMap之类的进行缓存对象,构建一个池容器,用户使用时再从中取出相应对象

享元模式在棋盘中的使用

我们围棋总共有棋盘总共有300多个下棋点,基本每盘都需要下棋200个以上,如果我们每次下棋都new一个棋子的话,这样系统里内存会很快不足,所以我们使用享元模式,将棋子按照颜色分类,总共只需要两个对象,棋子的颜色是内部状态,棋子的位置就是外部状态,代码实现就是

```
public abstract class AbsChessPieceUnit {
    public abstract void setPosition(Integer positionX,Integer positionY);
}
public class ChessPieceUnit extends AbsChessPieceUnit {
    private String color;
    private Integer positionX;
    private Integer positionY;
    public ChessPieceUnit(String color){
        this.color = color;
    }

    @Override
    public void setPosition(Integer positionX, Integer positionY) {
        this.positionX = positionX;
        this.positionY = positionY;
        System.out.println("颜色"+color+"的棋子位置为:positionX = [" + positionX +
"] positionY = ["+positionY+"]");
```

```
    }

    public String getColor() {
        return color;
    }
}
public class ChessPieceUnitFactory {
    private static Map<String, AbsChessPieceUnit> map = new HashMap<>();

    public static AbsChessPieceUnit getChessPieceUnit(String color){
        if (map.containsKey(color)) {

            return map.get(color);
        }
        map.put(color,new ChessPieceUnit(color));
        return map.get(color);
    }
}
```

享元模式在Integer的运用

首先来看Integer.valueOf(int i);方法

```
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }
```

这里发现,如果`i >= IntegerCache.low && i <= IntegerCache.high`时,直接返回一个已在Integer缓存池中缓存的对象,否则才是new 一个新的对象,IntegerCache在初始化的时候就生成了一个-128到127(可修改)大小的数组进行缓存,减少多个相同对象导致的内存浪费

```
private static class IntegerCache {
        static final int low = -128;
        static final int high;
        static final Integer cache[];

        static {
            // high value may be configured by property
            int h = 127;
            String integerCacheHighPropValue =

 sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
            if (integerCacheHighPropValue != null) {
                try {
                    int i = parseInt(integerCacheHighPropValue);
                    i = Math.max(i, 127);
                    // Maximum array size is Integer.MAX_VALUE
                    h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
                } catch( NumberFormatException nfe) {
                    // If the property cannot be parsed into an int, ignore it.
                }
            }
            high = h;
```

```
            cache = new Integer[(high - low) + 1];
            int j = low;
            for(int k = 0; k < cache.length; k++)
                cache[k] = new Integer(j++);

            // range [-128, 127] must be interned (JLS7 5.1.7)
            assert IntegerCache.high >= 127;
        }

        private IntegerCache() {}
    }
```

代理模式

基本介绍:可以为对象提供一个增强对象,无需修改对象的代码,就可以增强额外的功能操作,扩展目标对象的功能.基本分为静态代理和动态代理,其中动态代理通常又有两种实现方式,接口代理和Cglib代理,动态代理的含义是可以在内存中动态的创建对象,不需要实现接口.

静态代理

静态代理在使用时,需要定义接口或者父类,通过代理对象与被代理对象一同实现接口或者继承相同父类实现.

角色:

- 需要一个接口或者父类
- 一个具体的实现类
- 代理类,与被代理类一同实现相应接口或继承对应父类,并聚合被代理类

```
//接口
public interface ITeacher {
    void teacher();
}
//实现类
public class TeacherImpl implements ITeacher {
    @Override
    public void teacher() {
        System.out.println("开始上课!");
    }
}
//代理类
public class TeacherProxy implements ITeacher {
    private ITeacher target;

    public TeacherProxy(ITeacher object) {
        this.target = object;
    }

    @Override
    public void teacher() {
        //增强对应方法
        System.out.println("上课前!");
        target.teacher();
        System.out.println("上课后!");
    }
}
```

```
//客户端类
public class Client {
    public static void main(String[] args) {
        ITeacher teacher = new TeacherImpl();
        TeacherProxy teacherProxy = new TeacherProxy(teacher);
        teacherProxy.teacher();
    }
}
```

动态代理-接口代理

基本介绍:动态的在内存中生成对象,代理类无需实现接口或继承类,被代理类需要实现对应接口,通过JDK自带的API动态的生成相应对象

角色:

- 接口
- 实现类
- 动态代理类(使用JDK的java.lang.reflect.Proxy#newProxyInstance)
- 中介类,用于增强对应方法

```
//接口
public interface ITeacher {
    void teacher();
}
//实现类
public class TeacherImpl implements ITeacher {
    @Override
    public void teacher() {
        System.out.println("动态代理开始上课!");
    }
}
//中介类
public class DynamicProxyInvocationHandler implements InvocationHandler {

    private Object target;
    public DynamicProxyInvocationHandler(Object object){
        target = object;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println("上课前");
        Object result = method.invoke(target, args);
        System.out.println("上课后");
        return result;
    }
}
//代理类
public class TeacherDynamicProxy {
    private Object target;

    public TeacherDynamicProxy(Object target) {
        this.target = target;
    }
```

```
    public Object getProxyInstance(){
        //调用JDK的API
        return
Proxy.newProxyInstance(target.getClass().getClassLoader(),target.getClass().getI
nterfaces(),
                new DynamicProxyInvocationHandler(target));

    }
}
//客户端类
public class Client {
    public static void main(String[] args) {
        ITeacher teacher = new TeacherImpl();
        TeacherDynamicProxy dynamicProxy = new TeacherDynamicProxy(teacher);
        ITeacher teacherProxy = (ITeacher) dynamicProxy.getProxyInstance();
        teacherProxy.teacher();
    }
}
```

动态代理-CGLIB代理

基本介绍:

- 静态代理与动态代理-接口代理都需要对象实现相应的接口,但是如果对象没有实现相应接口而需要进行代理的话,那我们可以使用目标对象子类来实现代理,称为**CGLIB代理**
- Cglib也称为子类代理,在内存中构建一个子类对象实现对目标对象的功能扩展
- Cglib需要引入相应的jar包,Spring AOP就是使用CGLIB实现方法拦截的
- Cglib底层是通过字节码处理框架(ASM)转换相应字节码并生成新的类

角色:

- 被代理对象
- CGLIB代理创建器，实现MethodInterceptor 接口

```
//被代理对象
public class TeacherDAO {
    public void teacher(){
        System.out.println("cglib上课!");
    }
}
//代理创建器
public class DynamicCglibProxy implements MethodInterceptor {

    private Enhancer enhancer = new Enhancer();
    private Object target;

    public DynamicCglibProxy(Object obj) {
        target = obj;
    }
    public Object getProxy(){
        //设置需要代理的父类
        enhancer.setSuperclass(target.getClass());
        enhancer.setCallback(this);
        //动态创建子类实例
        return enhancer.create();
    }
```

```
    @Override
    public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
        System.out.println("上课前!");
        Object result = method.invoke(target, objects);
        System.out.println("上课后! ");
        return result;
    }
}
//客户端类
public class Client {
    public static void main(String[] args) {
        TeacherDAO teacherDAO = new TeacherDAO();
        DynamicCglibProxy dynamicCglibProxy = new DynamicCglibProxy(teacherDAO);
        TeacherDAO teacherProxy = (TeacherDAO) dynamicCglibProxy.getProxy();
        teacherProxy.teacher();
    }
}
```