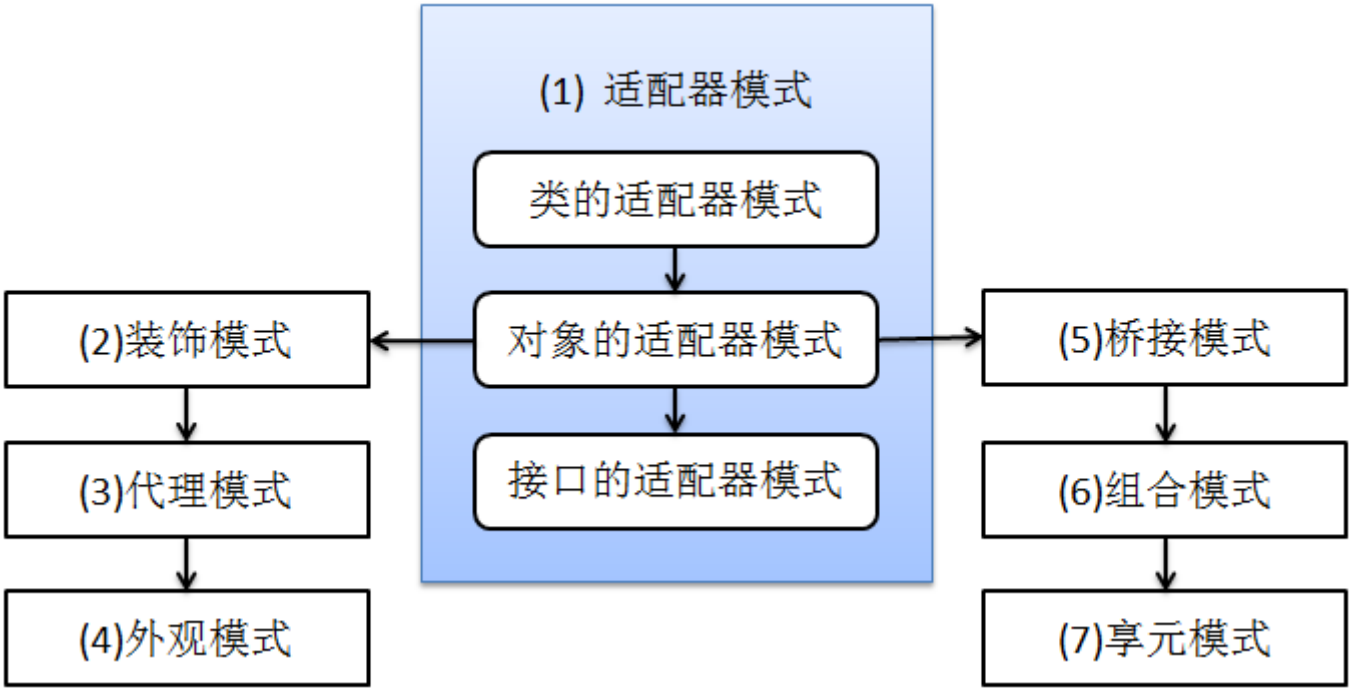
Java之美[从菜鸟到高手演变]之设计模式二

2012年11月30日 15:20:46 终点 阅读数：170906 标签： DesignPattern java 系统架构

在阅读过程中有任何问题，请及时联系：egg。

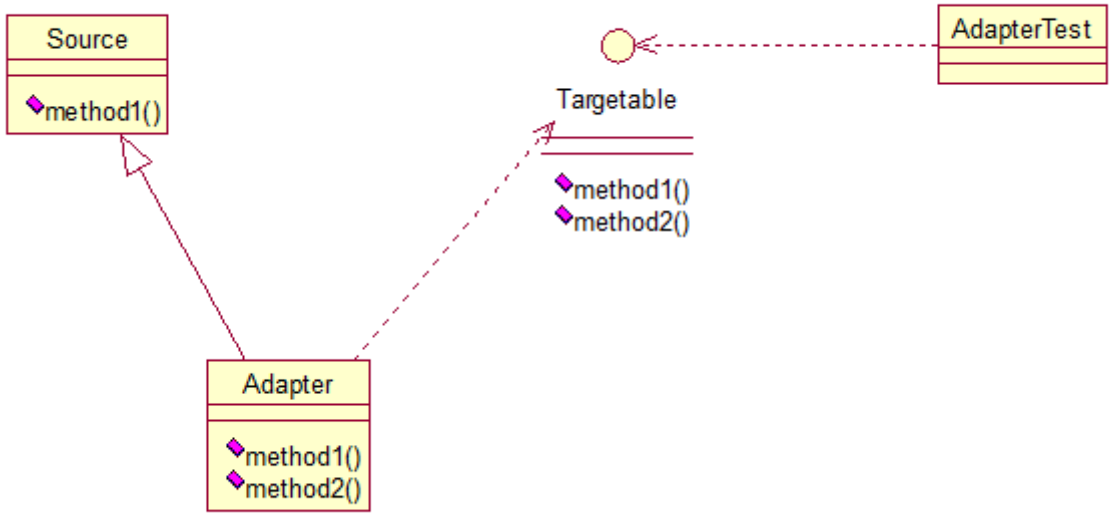
邮箱：**xtfggef@gmail.com** 微博：**http://weibo.com/xtfggef**

我们接着讨论设计模式，上篇文章我讲完了5种创建型模式，这章开始，我将讲下7种结构型模式：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式。其中对象的适配器模式是各种模式的起源，我们看下面的图：



6、适配器模式（Adapter）

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。首先，我们来看看**类的适配器模式**，先看类图：



核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口时Targetable，通过Adapter类，将Source的功能扩展到Targetable里，看代码：

```
public class Source {

	public void method1() {
		System.out.println("this is original method!");
	}
}
```

```
public interface Targetable {

	/* 与原类中的方法相同 */
	public void method1();

	/* 新类的方法 */
	public void method2();
}
```

```
public class Adapter extends Source implements Targetable {

	@Override
	public void method2() {
		System.out.println("this is the targetable method!");
	}
```

```
}
```

Adapter类继承Source类，实现Targetable接口，下面是测试类：

```
public class AdapterTest {

        public static void main(String[] args) {
                Targetable target = new Adapter();
                target.method1();
                target.method2();
        }
}
```

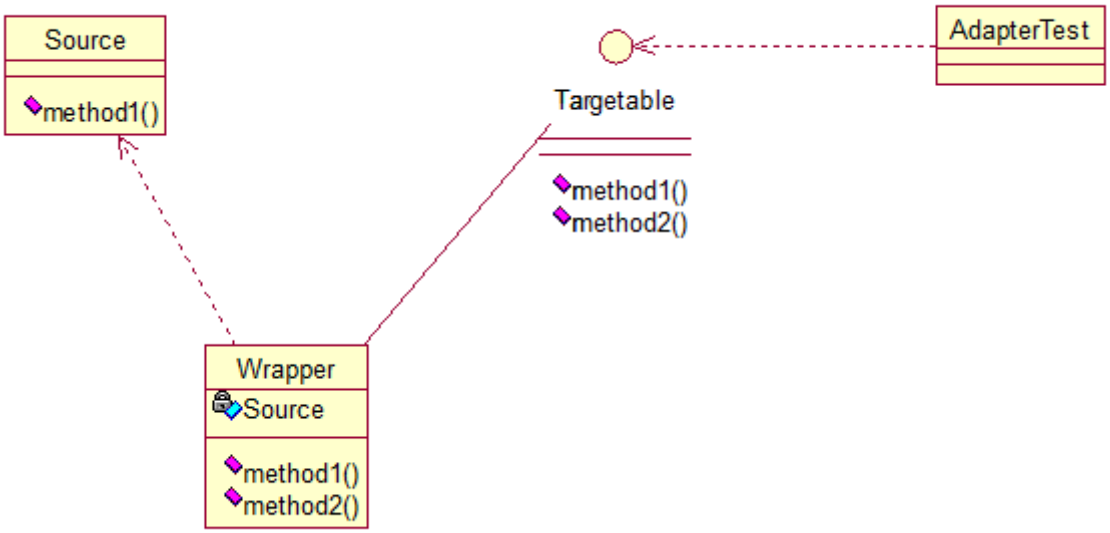
输出：

this is original method!
this is the targetable method!

这样Targetable接口的实现类就具有了Source类的功能。

对象的适配器模式

基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题。看图：



只需要修改Adapter类的源码即可：

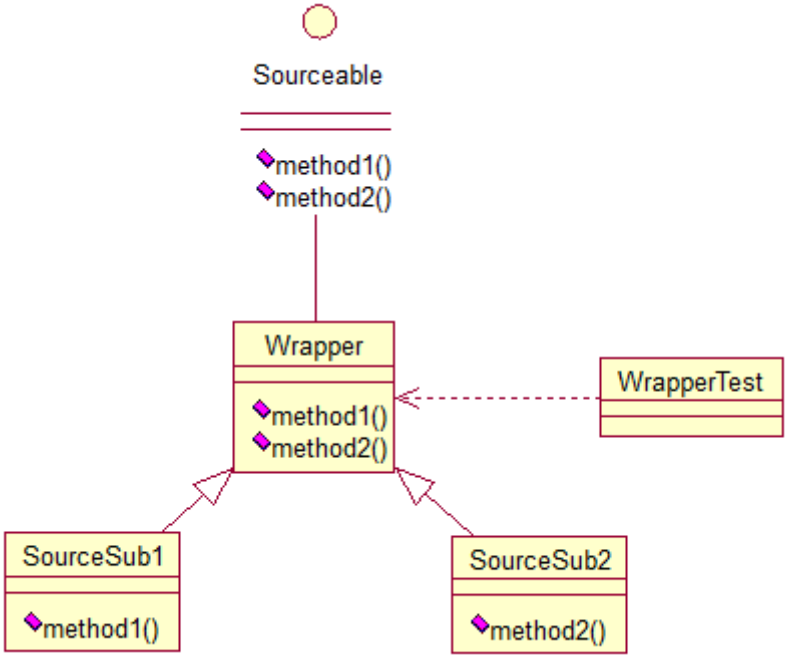
```
public class Wrapper implements Targetable {

        private Source source;

        public Wrapper(Source source){
                super();
                this.source = source;
        }
        @Override
        public void method2() {
                System.out.println("this is the targetable method!");
        }

        @Override
        public void method1() {
                source.method1();
        }
}
```

测试类：

```
public class AdapterTest {

        public static void main(String[] args) {
                Source source = new Source();
                Targetable target = new Wrapper(source);
                target.method1();
                target.method2();
        }
}
```

输出与第一种一样，只是适配的方法不同而已。

第三种适配器模式是**接口的适配器模式**，接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。看一下类图：



这个很好理解，在实际开发中，我们也常会遇到这种接口中定义了太多的方法，以致于有时我们在一些实现类中并不是都需要。看代码：

```
public interface Sourceable {

        public void method1();
        public void method2();
}
```

抽象类Wrapper2：

```
public abstract class Wrapper2 implements Sourceable{

        public void method1(){}
        public void method2(){}
}
```

```
public class SourceSub1 extends Wrapper2 {
        public void method1(){
                System.out.println("the sourceable interface's first Sub1!");
        }
}
```

```
public class SourceSub2 extends Wrapper2 {
        public void method2(){
                System.out.println("the sourceable interface's second Sub2!");
        }
}
```

```
public class WrapperTest {

        public static void main(String[] args) {
                Sourceable source1 = new SourceSub1();
                Sourceable source2 = new SourceSub2();

                source1.method1();
                source1.method2();
                source2.method1();
                source2.method2();
        }
}
```

测试输出：

the sourceable interface's first Sub1!
the sourceable interface's second Sub2!

达到了我们的效果！

讲了这么多，总结一下三种适配器模式的应用场景：

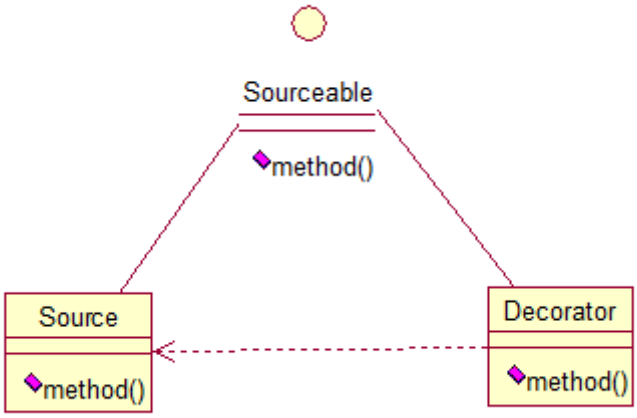
类的适配器模式：当希望将**一个类**转换成满足**另一个新接口**的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

7、装饰模式（Decorator）

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例，关系图如下：



Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能，代码如下：

```
public interface Sourceable {
        public void method();
}
```

```
public class Source implements Sourceable {

        @Override
        public void method() {
                System.out.println("the original method!");
        }
}
```

```
public class Decorator implements Sourceable {

        private Sourceable source;

        public Decorator(Sourceable source){
                super();
                this.source = source;
        }
        @Override
        public void method() {
                System.out.println("before decorator!");
                source.method();
                System.out.println("after decorator!");
        }
}
```

测试类：

```
public class DecoratorTest {

        public static void main(String[] args) {
                Sourceable source = new Source();
                Sourceable obj = new Decorator(source);
                obj.method();
        }
}
```

输出：

before decorator!
the original method!
after decorator!

装饰器模式的应用场景：

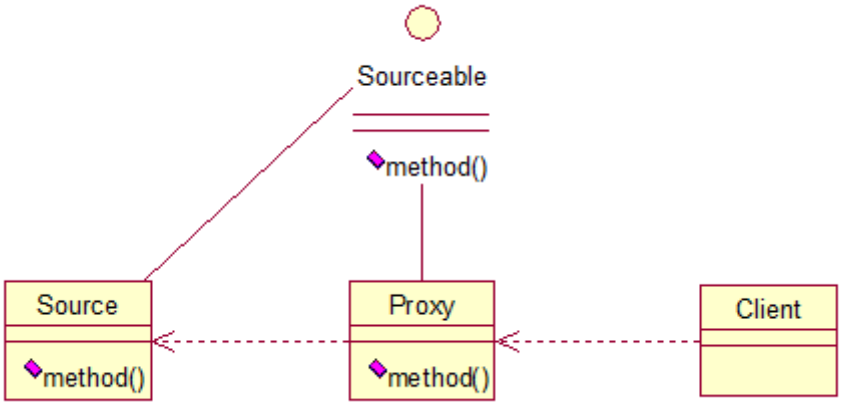
1、需要扩展一个类的功能。

2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

缺点：产生过多相似的对象，不易排错！

8、代理模式（Proxy）

其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作，比如我们在租房子的时候回去找中介，为什么呢？因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。先来看看关系图：



根据上文的阐述，代理模式就比较容易的理解了，我们看下代码：

```
public interface Sourceable {
        public void method();
}
```

```
public class Source implements Sourceable {

        @Override
        public void method() {
                System.out.println("the original method!");
        }
}
```

```
public class Proxy implements Sourceable {

        private Source source;
        public Proxy(){
                super();
                this.source = new Source();
        }
        @Override
        public void method() {
                before();
                source.method();
                atfer();
        }
        private void atfer() {
                System.out.println("after proxy!");
```

```
        }
            private void before() {
                System.out.println("before proxy!");
        }
}
```

测试类：

```
public class ProxyTest {

        public static void main(String[] args) {
                Sourceable source = new Proxy();
                source.method();
        }

}
```

输出：

before proxy!
the original method!
after proxy!

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

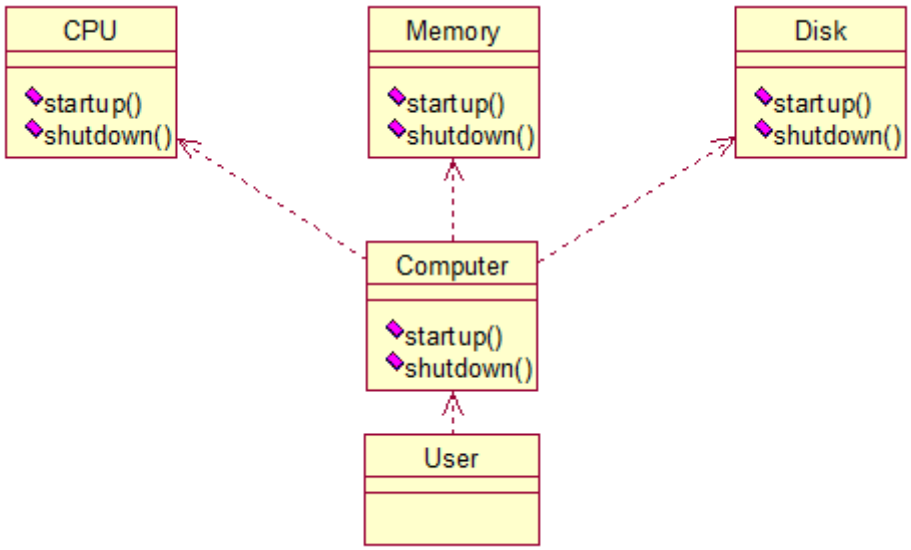
1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。

2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

9、外观模式（Facade）

外观模式是为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口，看下类图：（我们以一个计算机的启动过程为例）



我们先看下实现类：

```
public class CPU {

	public void startup(){
		System.out.println("cpu startup!");
	}

	public void shutdown(){
		System.out.println("cpu shutdown!");
	}
}
```

```
public class Memory {

	public void startup(){
		System.out.println("memory startup!");
	}

	public void shutdown(){
		System.out.println("memory shutdown!");
	}
}
```

```
public class Disk {
```

```
        public void startup(){
             System.out.println("disk startup!");         }

        public void shutdown(){
                System.out.println("disk shutdown!");
        }
}
```

```
public class Computer {
        private CPU cpu;
        private Memory memory;
        private Disk disk;

        public Computer(){
                cpu = new CPU();
                memory = new Memory();
                disk = new Disk();
        }

        public void startup(){
                System.out.println("start the computer!");
                cpu.startup();
                memory.startup();
                disk.startup();
                System.out.println("start computer finished!");
        }

        public void shutdown(){
                System.out.println("begin to close the computer!");
                cpu.shutdown();
                memory.shutdown();
                disk.shutdown();
                System.out.println("computer closed!");
        }
}
```

User类如下：

```
public class User {

        public static void main(String[] args) {
```

```
        Computer computer = new Computer();
        computer.startup();
        computer.shutdown();
    }
}
```

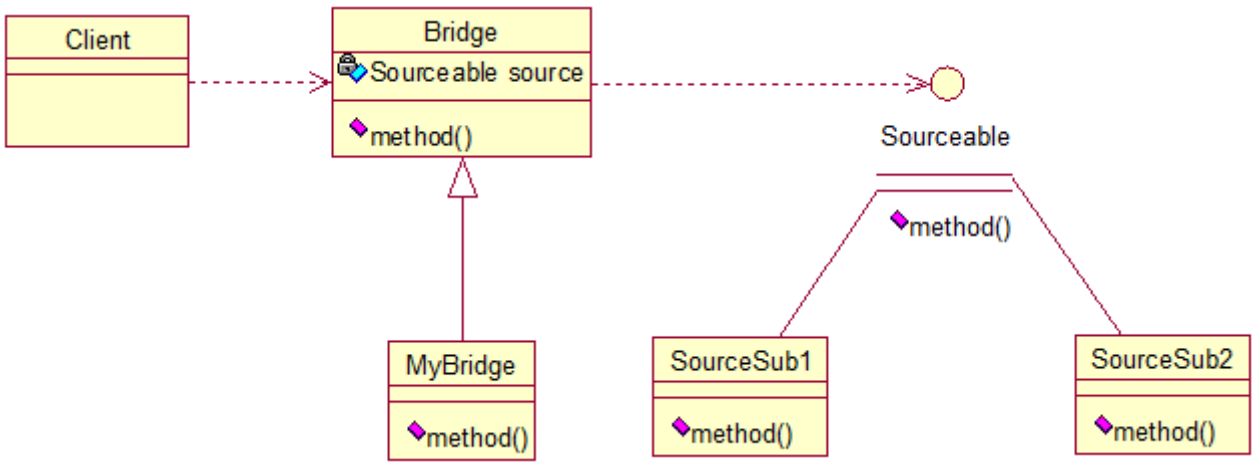
输出：

start the computer!
cpu startup!
memory startup!
disk startup!
start computer finished!
begin to close the computer!
cpu shutdown!
memory shutdown!
disk shutdown!
computer closed!

如果我们没有Computer类，那么，CPU、Memory、Disk他们之间将会相互持有实例，产生关系，这样会造成严重的依赖，修改一个类，可能会带来其他类的修改，这不是我们想要看到的，有了Computer类，他们之间的关系被放在了Computer类里，这样就起到了解耦的作用，这，就是外观模式！

10、桥接模式（Bridge）

桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：**将抽象化与实现化解耦，使得二者可以独立变化**，像我们常用的JDBC桥DriverManager一样，JDBC进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，原因就是JDBC提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了。我们来看看关系图：



实现代码：

先定义接口：

```
public interface Sourceable {
        public void method();
}
```

分别定义两个实现类：

```
public class SourceSub1 implements Sourceable {

        @Override
        public void method() {
                System.out.println("this is the first sub!");
        }
}
```

```
public class SourceSub2 implements Sourceable {

        @Override
        public void method() {
                System.out.println("this is the second sub!");
        }
}
```

定义一个桥，持有Sourceable的一个实例：

```
public abstract class Bridge {
        private Sourceable source;

        public void method(){
                source.method();
        }

        public Sourceable getSource() {
                return source;
        }
```

```
        public void setSource(Sourceable source) {
            this.source = source;
        }
}
```

```
public class MyBridge extends Bridge {
        public void method(){
                getSource().method();
        }
}
```

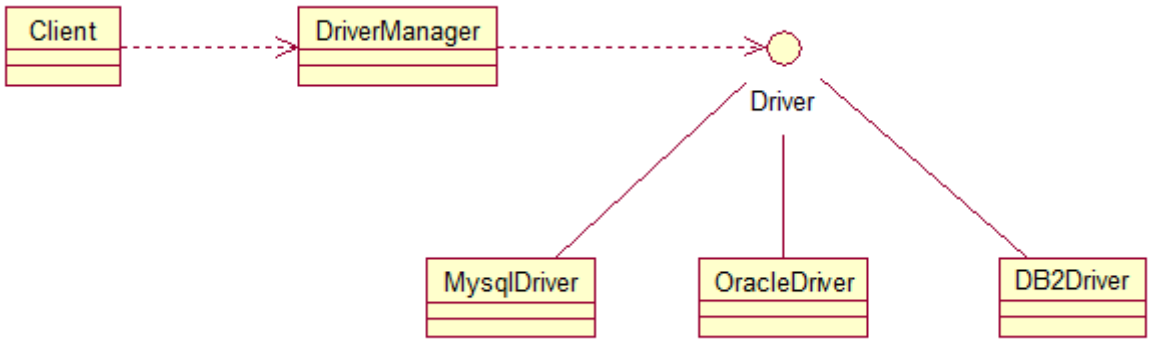
测试类：

```
public class BridgeTest {

        public static void main(String[] args) {

                Bridge bridge = new MyBridge();

                /*调用第一个对象*/
                Sourceable source1 = new SourceSub1();
                bridge.setSource(source1);
                bridge.method();

                /*调用第二个对象*/
                Sourceable source2 = new SourceSub2();
                bridge.setSource(source2);
                bridge.method();
        }
}
```

output：

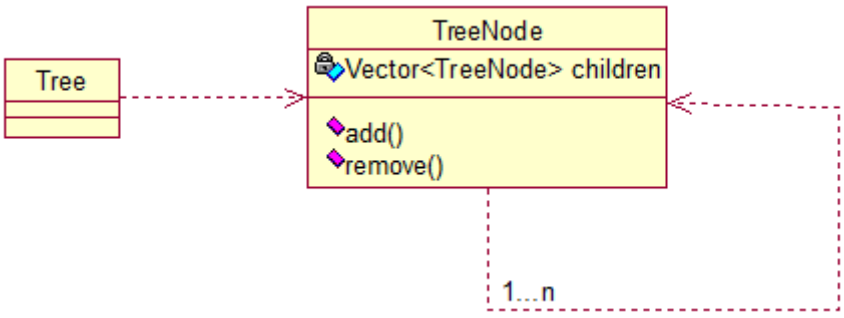
this is the first sub!
this is the second sub!

这样，就通过对Bridge类的调用，实现了对接口Sourceable的实现类SourceSub1和SourceSub2的调用。接下来我再画个图，大家就应该明白了，因为这个图是我们JDBC连接的原理，有数据库学习基础的，一结合就都懂了。



11、组合模式（Composite）

组合模式有时又叫**部分-整体**模式在处理类似树形结构的问题时比较方便，看看关系图：



直接来看代码：

```
public class TreeNode {

        private String name;
        private TreeNode parent;
        private Vector<TreeNode> children = new Vector<TreeNode>();

        public TreeNode(String name){
                this.name = name;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public TreeNode getParent() {
                return parent;
```

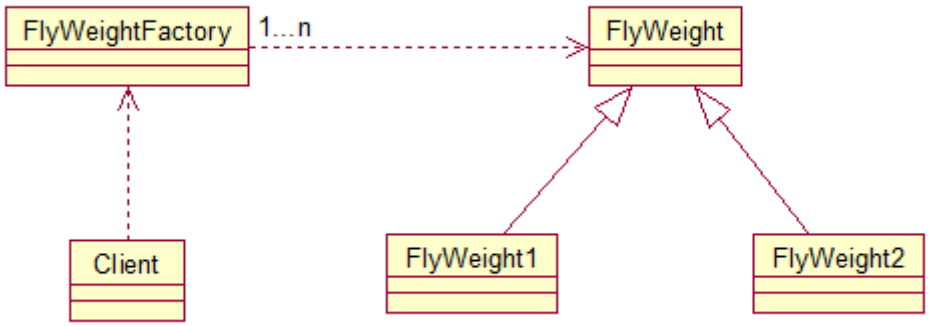
```
        }

        public void setParent(TreeNode parent) {
                this.parent = parent;
        }

        //添加孩子节点
        public void add(TreeNode node){
                children.add(node);
        }

        //删除孩子节点
        public void remove(TreeNode node){
                children.remove(node);
        }

        //取得孩子节点
        public Enumeration<TreeNode> getChildren(){
                return children.elements();
        }
}
```

```
public class Tree {

        TreeNode root = null;

        public Tree(String name) {
                root = new TreeNode(name);
        }

        public static void main(String[] args) {
                Tree tree = new Tree("A");
                TreeNode nodeB = new TreeNode("B");
                TreeNode nodeC = new TreeNode("C");

                nodeB.add(nodeC);
                tree.root.add(nodeB);
                System.out.println("build the tree finished!");
        }
}
```

使用场景：将多个对象组合在一起进行操作，常用于表示树形结构中，例如二叉树，数等。

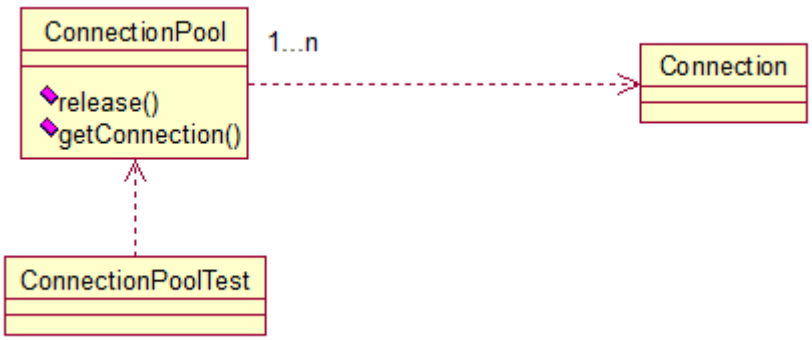
12、享元模式（Flyweight）

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。



FlyWeightFactory负责创建和管理享元单元，当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，就返回已经存在的对象，如果没有，则创建一个新对象，FlyWeight是超类。一提到共享池，我们很容易联想到Java里面的JDBC连接池，想想每个连接的特点，我们不难总结出：适用于作共享的一些个对象，他们有一些共有的属性，就拿数据库连接池来说，url、driverClassName、username、password及dbname，这些属性对于每个连接来说都是一样的，所以就适合用享元模式来处理，建一个工厂类，将上述类似属性作为内部数据，其它的作为外部数据，在方法调用时，当做参数传进来，这样就节省了空间，减少了实例的数量。

看个例子：



看下数据库连接池的代码：

```
public class ConnectionPool {

        private Vector<Connection> pool;

        /*公有属性*/
        private String url = "jdbc:mysql://localhost:3306/test";
        private String username = "root";
```

```
        private String password = "root";
    private String driverClassName = "com.mysql.jdbc.Driver";
        private int poolSize = 100;
        private static ConnectionPool instance = null;
        Connection conn = null;

        /*构造方法，做一些初始化工作*/
        private ConnectionPool() {
                pool = new Vector<Connection>(poolSize);

                for (int i = 0; i < poolSize; i++) {
                        try {
                                Class.forName(driverClassName);

                                conn = DriverManager.getConnection(url, username,
                                pool.add(conn);
                        } catch (ClassNotFoundException e) {
                                e.printStackTrace();
                        } catch (SQLException e) {
                                e.printStackTrace();                        }
                }
        }

        /* 返回连接到连接池 */
        public synchronized void release() {
                pool.add(conn);
        }

        /* 返回连接池中的一个数据库连接 */
        public synchronized Connection getConnection() {
                if (pool.size() > 0) {
                        Connection conn = pool.get(0);
                        pool.remove(conn);
                        return conn;
                } else {
                        return null;
                }
        }
}
```

通过连接池的管理，实现了数据库连接的共享，不需要每一次都重新创建连接，节省了数据库重新创建的开销，提升了系统的性能！本章讲解了7种结构型模式，因为篇幅的问题，剩下的11种行为型模式，

我们将另起篇章，敬请读者朋友们持续关注！