在阎宏博士的《JAVA与模式》一书中开头是这样描述访问者（Visitor）模式的：

**访问者模式是对象的行为模式。访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构则可以保持不变。**

**分派的概念**

变量被声明时的类型叫做变量的静态类型(Static Type)，有些人又把静态类型叫做明显类型(Apparent Type)；而变量所引用的对象的真实类型又叫做变量的实际类型(Actual Type)。比如：

List list = null;

list = new ArrayList();

　　声明了一个变量list，它的静态类型（也叫明显类型）是List，而它的实际类型是ArrayList。

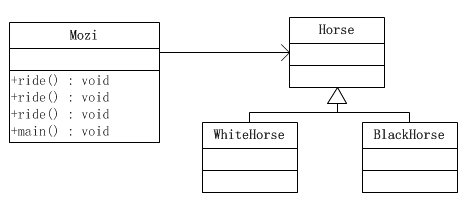
　　根据对象的类型而对方法进行的选择，就是分派(Dispatch)，分派(Dispatch)又分为两种，即**静态分派**和**动态分派**。

**静态分派(Static Dispatch)**发生在编译时期，分派根据静态类型信息发生。静态分派对于我们来说并不陌生，方法重载就是静态分派。

**动态分派(Dynamic Dispatch)**发生在运行时期，动态分派动态地置换掉某个方法。

**静态分派**

　　Java通过方法重载支持静态分派。用墨子骑马的故事作为例子，墨子可以骑白马或者黑马。墨子与白马、黑马和马的类图如下所示：



　　在这个系统中，墨子由Mozi类代表

public class Mozi {

public void ride(Horse h){

System.out.println("骑马");

}

public void ride(WhiteHorse wh){

System.out.println("骑白马");

}

public void ride(BlackHorse bh){

System.out.println("骑黑马");

}

public static void main(String[] args) {

Horse wh = new WhiteHorse();

Horse bh = new BlackHorse();

Mozi mozi = new Mozi();

mozi.ride(wh);

mozi.ride(bh);

}

}

　　显然，Mozi类的ride()方法是由三个方法重载而成的。这三个方法分别接受马(Horse)、白马(WhiteHorse)、黑马(BlackHorse)等类型的参数。

　　那么在运行时，程序会打印出什么结果呢？结果是程序会打印出相同的两行“骑马”。换言之，墨子发现他所骑的都是马。

　　为什么呢？两次对ride()方法的调用传入的是不同的参数，也就是wh和bh。它们虽然具有不同的真实类型，但是它们的静态类型都是一样的，均是Horse类型。

　　重载方法的分派是根据静态类型进行的，这个分派过程在编译时期就完成了。

**动态分派**

　　Java通过方法的重写支持动态分派。用马吃草的故事作为例子，代码如下所示：

public class Horse {

public void eat(){

System.out.println("马吃草");

}

}

public class BlackHorse extends Horse {

@Override

public void eat() {

System.out.println("黑马吃草");

}

}

public class Client {

public static void main(String[] args) {

Horse h = new BlackHorse();

h.eat();

}

}

　　变量h的静态类型是Horse，而真实类型是BlackHorse。如果上面最后一行的eat()方法调用的是BlackHorse类的eat()方法，那么上面打印的就是“黑马吃草”；相反，如果上面的eat()方法调用的是Horse类的eat()方法，那么打印的就是“马吃草”。

　　所以，问题的核心就是Java编译器在编译时期并不总是知道哪些代码会被执行，因为编译器仅仅知道对象的静态类型，而不知道对象的真实类型；而方法的调用则是根据对象的真实类型，而不是静态类型。这样一来，上面最后一行的eat()方法调用的是BlackHorse类的eat()方法，打印的是“黑马吃草”。

**分派的类型**

　　一个方法所属的对象叫做方法的接收者，方法的接收者与方法的参数统称做方法的宗量。比如下面例子中的Test类

public class Test {

public void print(String str){

System.out.println(str);

}

}

　　在上面的类中，print()方法属于Test对象，所以它的接收者也就是Test对象了。print()方法有一个参数是str,它的类型是String。

　　根据分派可以基于多少种宗量，可以将面向对象的语言划分为单分派语言(Uni-Dispatch)和多分派语言(Multi-Dispatch)。单分派语言根据一个宗量的类型进行对方法的选择，多分派语言根据多于一个的宗量的类型对方法进行选择。

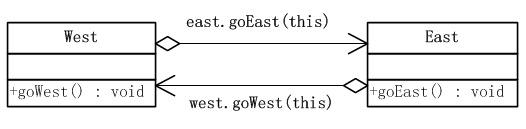
　　C++和Java均是单分派语言，多分派语言的例子包括CLOS和Cecil。按照这样的区分，Java就是**动态的单分派语言**，因为这种语言的动态分派仅仅会考虑到方法的接收者的类型，同时又是**静态的多分派语言**，因为这种语言对重载方法的分派会考虑到方法的接收者的类型以及方法的所有参数的类型。

　　在一个支持动态单分派的语言里面，有两个条件决定了一个请求会调用哪一个操作：一是请求的名字，而是接收者的真实类型。单分派限制了方法的选择过程，使得只有一个宗量可以被考虑到，这个宗量通常就是方法的接收者。在Java语言里面，如果一个操作是作用于某个类型不明的对象上面，那么对这个对象的真实类型测试仅会发生一次，这就是动态的单分派的特征。

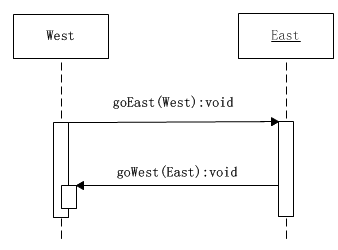
**双重分派**

　　一个方法根据两个宗量的类型来决定执行不同的代码，这就是“双重分派”。Java语言不支持动态的多分派，也就意味着Java不支持动态的双分派。但是通过使用设计模式，也可以在Java语言里实现动态的双重分派。

　　在Java中可以通过两次方法调用来达到两次分派的目的。类图如下所示：



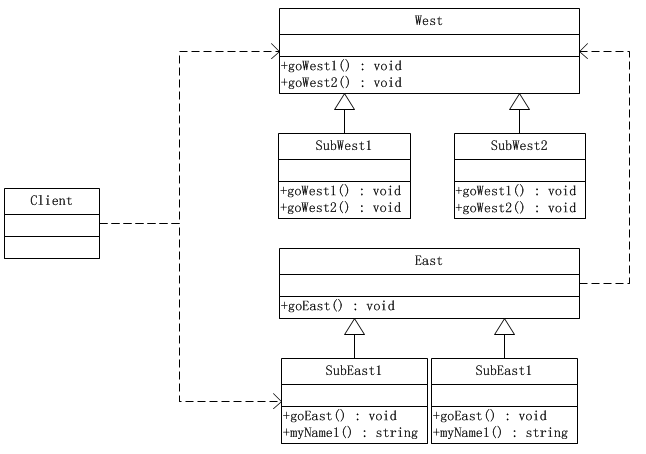
　　在图中有两个对象，左边的叫做West，右边的叫做East。现在West对象首先调用East对象的goEast()方法，并将它自己传入。在East对象被调用时，立即根据传入的参数知道了调用者是谁，于是反过来调用“调用者”对象的goWest()方法。通过两次调用将程序控制权轮番交给两个对象，其时序图如下所示：



　　这样就出现了两次方法调用，程序控制权被两个对象像传球一样，首先由West对象传给了East对象，然后又被返传给了West对象。

　　但是仅仅返传了一下球，并不能解决双重分派的问题。关键是怎样利用这两次调用，以及Java语言的动态单分派功能，使得在这种传球的过程中，能够触发两次单分派。

　　动态单分派在Java语言中是在子类重写父类的方法时发生的。换言之，West和East都必须分别置身于自己的类型等级结构中，如下图所示：



**源代码**

　　West类

public abstract class West {

public abstract void goWest1(SubEast1 east);

public abstract void goWest2(SubEast2 east);

}

　　SubWest1类

public class SubWest1 extends West{

@Override

public void goWest1(SubEast1 east) {

System.out.println("SubWest1 + " + east.myName1());

}

@Override

public void goWest2(SubEast2 east) {

System.out.println("SubWest1 + " + east.myName2());

}

}

　　SubWest2类

public class SubWest2 extends West{

@Override

public void goWest1(SubEast1 east) {

System.out.println("SubWest2 + " + east.myName1());

}

@Override

public void goWest2(SubEast2 east) {

System.out.println("SubWest2 + " + east.myName2());

}

}

　　East类

public abstract class East {

public abstract void goEast(West west);

}

　　SubEast1类

public class SubEast1 extends East{

@Override

public void goEast(West west) {

west.goWest1(this);

}

public String myName1(){

return "SubEast1";

}

}

　　SubEast2类

public class SubEast2 extends East{

@Override

public void goEast(West west) {

west.goWest2(this);

}

public String myName2(){

return "SubEast2";

}

}

　　客户端类

public class Client {

public static void main(String[] args) {

//组合1

East east = new SubEast1();

West west = new SubWest1();

east.goEast(west);

//组合2

east = new SubEast1();

west = new SubWest2();

east.goEast(west);

}

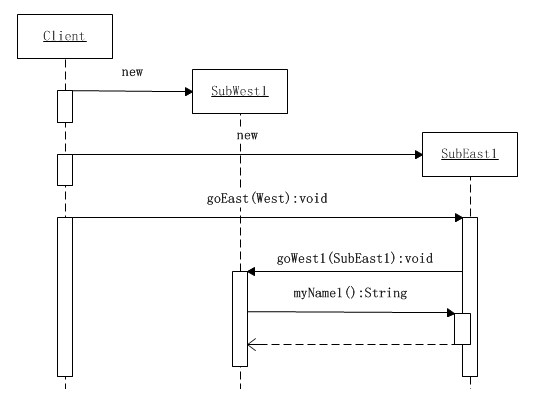
}

　　运行结果如下

SubWest1 + SubEast1  
SubWest2 + SubEast1

　　系统运行时，会首先创建SubWest1和SubEast1对象，然后客户端调用SubEast1的goEast()方法，并将SubWest1对象传入。由于SubEast1对象重写了其超类East的goEast()方法，因此，这个时候就发生了一次动态的单分派。当SubEast1对象接到调用时，会从参数中得到SubWest1对象，所以它就立即调用这个对象的goWest1()方法，并将自己传入。由于SubEast1对象有权选择调用哪一个对象，因此，在此时又进行一次动态的方法分派。

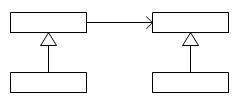
　　这个时候SubWest1对象就得到了SubEast1对象。通过调用这个对象myName1()方法，就可以打印出自己的名字和SubEast对象的名字，其时序图如下所示：



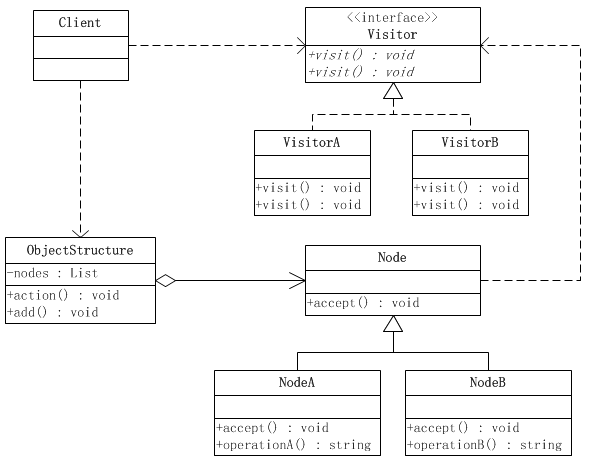
　　由于这两个名字一个来自East等级结构，另一个来自West等级结构中，因此，它们的组合式是动态决定的。这就是动态双重分派的实现机制。

**访问者模式的结构**

　　访问者模式适用于数据结构相对未定的系统，它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对自由地演化。访问者模式的简略图如下所示：



　　数据结构的每一个节点都可以接受一个访问者的调用，此节点向访问者对象传入节点对象，而访问者对象则反过来执行节点对象的操作。这样的过程叫做“双重分派”。节点调用访问者，将它自己传入，访问者则将某算法针对此节点执行。访问者模式的示意性类图如下所示：



　　访问者模式涉及到的角色如下：

　　●　　**抽象访问者(Visitor)角色：**声明了一个或者多个方法操作，形成所有的具体访问者角色必须实现的接口。

　　●　　**具体访问者(ConcreteVisitor)角色：**实现抽象访问者所声明的接口，也就是抽象访问者所声明的各个访问操作。

　　●　　**抽象节点(Node)角色：**声明一个接受操作，接受一个访问者对象作为一个参数。

　　●　　**具体节点(ConcreteNode)角色：**实现了抽象节点所规定的接受操作。

　　●　　**结构对象(ObjectStructure)角色：**有如下的责任，可以遍历结构中的所有元素；如果需要，提供一个高层次的接口让访问者对象可以访问每一个元素；如果需要，可以设计成一个复合对象或者一个聚集，如List或Set。

**源代码**

　　可以看到，抽象访问者角色为每一个具体节点都准备了一个访问操作。由于有两个节点，因此，对应就有两个访问操作。

public interface Visitor {

/\*\*

\* 对应于NodeA的访问操作

\*/

public void visit(NodeA node);

/\*\*

\* 对应于NodeB的访问操作

\*/

public void visit(NodeB node);

}

　　具体访问者VisitorA类

public class VisitorA implements Visitor {

/\*\*

\* 对应于NodeA的访问操作

\*/

@Override

public void visit(NodeA node) {

System.out.println(node.operationA());

}

/\*\*

\* 对应于NodeB的访问操作

\*/

@Override

public void visit(NodeB node) {

System.out.println(node.operationB());

}

}

　　具体访问者VisitorB类

public class VisitorB implements Visitor {

/\*\*

\* 对应于NodeA的访问操作

\*/

@Override

public void visit(NodeA node) {

System.out.println(node.operationA());

}

/\*\*

\* 对应于NodeB的访问操作

\*/

@Override

public void visit(NodeB node) {

System.out.println(node.operationB());

}

}

　　抽象节点类

public abstract class Node {

/\*\*

\* 接受操作

\*/

public abstract void accept(Visitor visitor);

}

　　具体节点类NodeA

public class NodeA extends Node{

/\*\*

\* 接受操作

\*/

@Override

public void accept(Visitor visitor) {

visitor.visit(this);

}

/\*\*

\* NodeA特有的方法

\*/

public String operationA(){

return "NodeA";

}

}

　　具体节点类NodeB

public class NodeB extends Node{

/\*\*

\* 接受方法

\*/

@Override

public void accept(Visitor visitor) {

visitor.visit(this);

}

/\*\*

\* NodeB特有的方法

\*/

public String operationB(){

return "NodeB";

}

}

　　结构对象角色类，这个结构对象角色持有一个聚集，并向外界提供add()方法作为对聚集的管理操作。通过调用这个方法，可以动态地增加一个新的节点。

public class ObjectStructure {

private List<Node> nodes = new ArrayList<Node>();

/\*\*

\* 执行方法操作

\*/

public void action(Visitor visitor){

for(Node node : nodes)

{

node.accept(visitor);

}

}

/\*\*

\* 添加一个新元素

\*/

public void add(Node node){

nodes.add(node);

}

}

　　客户端类

public class Client {

public static void main(String[] args) {

//创建一个结构对象

ObjectStructure os = new ObjectStructure();

//给结构增加一个节点

os.add(new NodeA());

//给结构增加一个节点

os.add(new NodeB());

//创建一个访问者

Visitor visitor = new VisitorA();

os.action(visitor);

}

}

　　虽然在这个示意性的实现里并没有出现一个复杂的具有多个树枝节点的对象树结构，但是，在实际系统中访问者模式通常是用来处理复杂的对象树结构的，而且访问者模式可以用来处理跨越多个等级结构的树结构问题。这正是访问者模式的功能强大之处。

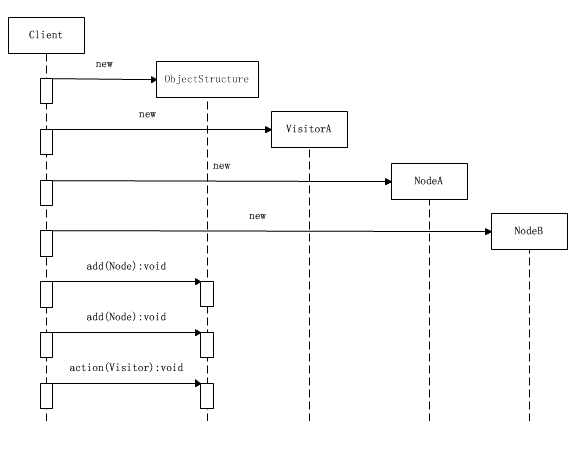
**准备过程时序图**

　　首先，这个示意性的客户端创建了一个结构对象，然后将一个新的NodeA对象和一个新的NodeB对象传入。

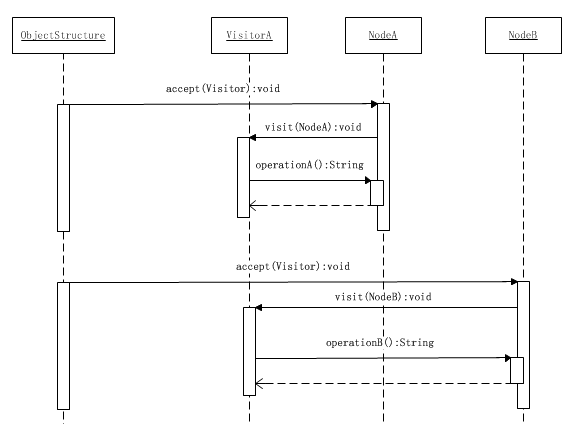
　　其次，客户端创建了一个VisitorA对象，并将此对象传给结构对象。

　　然后，客户端调用结构对象聚集管理方法，将NodeA和NodeB节点加入到结构对象中去。

　　最后，客户端调用结构对象的行动方法action()，启动访问过程。



**访问过程时序图**



　　结构对象会遍历它自己所保存的聚集中的所有节点，在本系统中就是节点NodeA和NodeB。首先NodeA会被访问到，这个访问是由以下的操作组成的：

　　（1）NodeA对象的接受方法accept()被调用，并将VisitorA对象本身传入；

　　（2）NodeA对象反过来调用VisitorA对象的访问方法，并将NodeA对象本身传入；

　　（3）VisitorA对象调用NodeA对象的特有方法operationA()。

　　从而就完成了双重分派过程，接着，NodeB会被访问，这个访问的过程和NodeA被访问的过程是一样的，这里不再叙述。

**访问者模式的优点**

　　●　　**好的扩展性**

　　能够在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能。

　　●　　**好的复用性**

　　可以通过访问者来定义整个对象结构通用的功能，从而提高复用程度。

　　●　　**分离无关行为**

　　可以通过访问者来分离无关的行为，把相关的行为封装在一起，构成一个访问者，这样每一个访问者的功能都比较单一。

**访问者模式的缺点**

　　●　　**对象结构变化很困难**

　　不适用于对象结构中的类经常变化的情况，因为对象结构发生了改变，访问者的接口和访问者的实现都要发生相应的改变，代价太高。

　　●　**破坏封装**

　　访问者模式通常需要对象结构开放内部数据给访问者和ObjectStructrue，这破坏了对象的封装性。