

享元模式

整理自：《java与模式》之享元模式

在阎宏博士的《JAVA与模式》一书中开头是这样描述享元（Flyweight）模式的：

Flyweight在拳击比赛中指最轻量级，即“蝇量级”或“雨量级”，这里选择使用“享元模式”的意译，是因为这样更能反映模式的用意。享元模式是对象的结构模式。享元模式以共享的方式高效地支持大量的细粒度对象。

Java中的String类型

在JAVA语言中，String类型就是使用了享元模式。String对象是final类型，对象一旦创建就不可改变。在JAVA中字符串常量都是存在常量池中的，JAVA会确保一个字符串常量在常量池中只有一个拷贝。String a="abc"，其中"abc"就是一个字符串常量。

```
public class Test {  
  
    public static void main(String[] args) {  
  
        String a = "abc";  
        String b = "abc";  
        System.out.println(a==b);  
  
    }  
}
```

上面的例子中结果为：true，这就说明a和b两个引用都指向了常量池中的同一个字符串常量“abc”。这样的设计避免了在创建N多相同对象时所产生的不必要的大量的资源消耗。

享元模式的结构

享元模式采用一个共享来避免大量拥有相同内容对象的开销。这种开销最常见、最直观的就是内存的损耗。享元对象能做到共享的关键是区分内蕴状态(Internal State)和外蕴状态(External State)。

一个内蕴状态是存储在享元对象内部的，并且是不会随环境的改变而有所不同。因此，

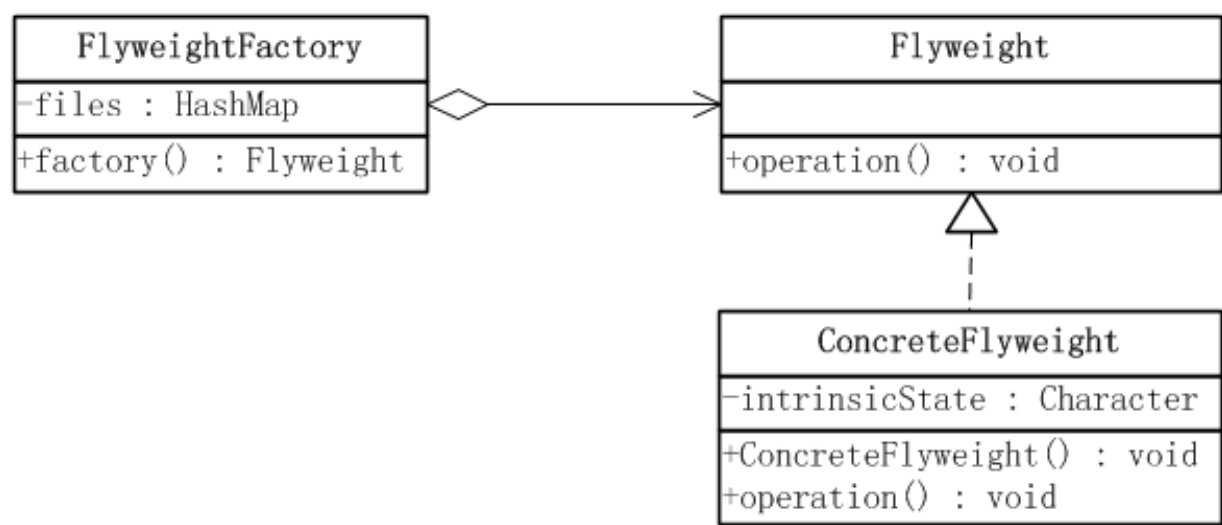
一个享元可以具有内蕴状态并可以共享。

一个外蕴状态是随环境的改变而改变的、不可以共享的。享元对象的外蕴状态必须由客户端保存，并在享元对象被创建之后，在需要使用的時候再传入到享元对象内部。外蕴状态不可以影响享元对象的內蕴状态，它们是相互独立的。

享元模式可以分成**单纯享元模式**和**复合享元模式**两种形式。

单纯享元模式

在单纯的享元模式中，所有的享元对象都是可以共享的。



单纯享元模式所涉及到的角色如下：

- **抽象享元(Flyweight)角色：** 给出一个抽象接口，以规定出所有具体享元角色需要实现的方法。
- **具体享元(ConcreteFlyweight)角色：** 实现抽象享元角色所规定出的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。
- **享元工厂(FlyweightFactory)角色：** 本角色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象的时候，享元工厂角色会检查系统中是否已经有一个符合要求的享元对象。如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个合适的享元对象。

源代码

抽象享元角色类

```
public interface Flyweight {  
    //一个示意性方法，参数state是外蕴状态  
    public void operation(String state);  
}
```

具体享元角色类ConcreteFlyweight有一个内蕴状态，在本例中一个Character类型的intrinsicState属性代表，它的值应当在享元对象被创建时赋予。所有的内蕴状态在对象创建之后，就不会再改变了。

如果一个享元对象有外蕴状态的话，所有的外部状态都必须存储在客户端，在使用享元对象时，再由客户端传入享元对象。这里只有一个外蕴状态，operation()方法的参数state就是由外部传入的外蕴状态。

```
public class ConcreteFlyweight implements Flyweight {  
    private Character intrinsicState = null;  
    /**  
     * 构造函数，内蕴状态作为参数传入  
     * @param state  
     */  
    public ConcreteFlyweight(Character state){  
        this.intrinsicState = state;  
    }  
  
    /**  
     * 外蕴状态作为参数传入方法中，改变方法的行为，  
     * 但是并不改变对象的内蕴状态。  
     */  
    @Override  
    public void operation(String state) {  
        // TODO Auto-generated method stub  
        System.out.println("Intrinsic State = " + this.intrinsicState);  
        System.out.println("Extrinsic State = " + state);  
    }  
}
```

享元工厂角色类，必须指出的是，客户端不可以直接将具体享元类实例化，而必须通过一个工厂对象，利用一个factory()方法得到享元对象。一般而言，享元工厂对象在整个系统中只有一个，因此也可以使用单例模式。

当客户端需要单纯享元对象的时候，需要调用享元工厂的factory()方法，并传入所需的单纯享元对象的内蕴状态，由工厂方法产生所需要的享元对象。

```
public class FlyweightFactory {
    private Map<Character, Flyweight> files = new HashMap<Character, Flyweight>();

    public Flyweight factory(Character state){
        //先从缓存中查找对象
        Flyweight fly = files.get(state);
        if(fly == null){
            //如果对象不存在则创建一个新的Flyweight对象
            fly = new ConcreteFlyweight(state);
            //把这个新的Flyweight对象添加到缓存中
            files.put(state, fly);
        }
        return fly;
    }
}
```

客户端类

```
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        FlyweightFactory factory = new FlyweightFactory();
        Flyweight fly = factory.factory(new Character('a'));
        fly.operation("First Call");

        fly = factory.factory(new Character('b'));
        fly.operation("Second Call");

        fly = factory.factory(new Character('a'));
        fly.operation("Third Call");
    }
}
```

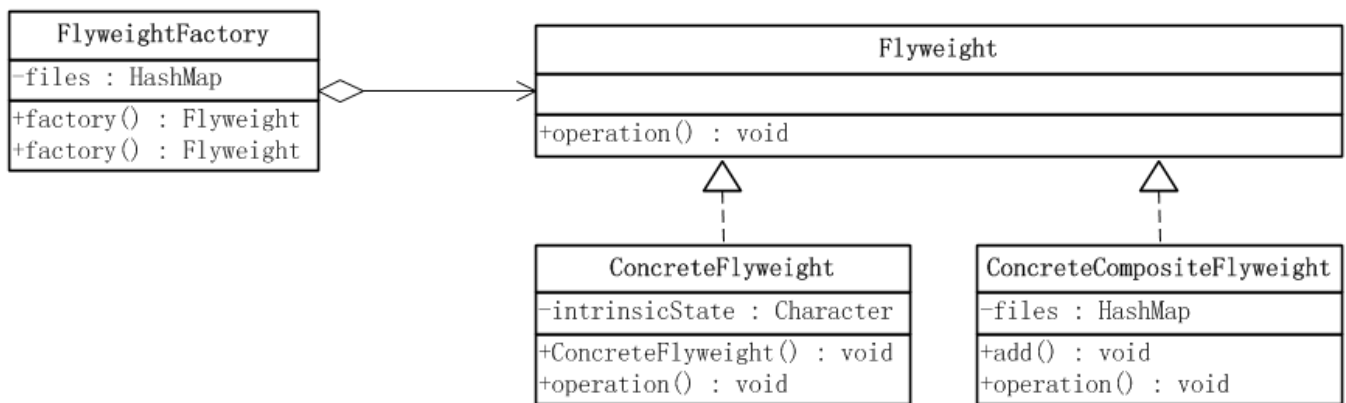
```
}
```

虽然客户端申请了三个享元对象，但是实际创建的享元对象只有两个，这就是共享的含义。运行结果如下：

```
Problems @ Javadoc Declaration Console Search
<terminated> Client (9) [Java Application] E:\Genuittec\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin\javaw.exe (2012-4-25 下午09:54:20)
Intrinsic State = a
Extrinsic State = First Call
Intrinsic State = b
Extrinsic State = Second Call
Intrinsic State = a
Extrinsic State = Third Call
```

复合享元模式

在单纯享元模式中，所有的享元对象都是单纯享元对象，也就是说都是可以直接共享的。还有一种较为复杂的情况，将一些单纯享元使用合成模式加以复合，形成复合享元对象。这样的复合享元对象本身不能共享，但是它们可以分解成单纯享元对象，而后者则可以共享。



复合享元角色所涉及到的角色如下：

- 抽象享元(Flyweight)角色：给出一个抽象接口，以规定出所有具体享元角色需要实现的方法。
- 具体享元(ConcreteFlyweight)角色：实现抽象享元角色所规定出的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。
- 复合享元(ConcreteCompositeFlyweight)角色：复合享元角色所代表的对象是不可以共享的，但是一个复合享元对象可以分解成为多个本身是单纯享元对象的组合。复合享元角色又称作不可共享的享元对象。

- 享元工厂(FlyweightFactory)角色：本角色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象的时候，享元工厂角色会检查系统中是否已经有一个符合要求的享元对象。如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个合适的享元对象。

源代码

抽象享元角色类

```
public interface Flyweight {  
    //一个示意性方法，参数state是外蕴状态  
    public void operation(String state);  
}
```

具体享元角色类

```
public class ConcreteFlyweight implements Flyweight {  
    private Character intrinsicState = null;  
    /**  
     * 构造函数，内蕴状态作为参数传入  
     * @param state  
     */  
    public ConcreteFlyweight(Character state){  
        this.intrinsicState = state;  
    }  
  
    /**  
     * 外蕴状态作为参数传入方法中，改变方法的行为，  
     * 但是并不改变对象的内蕴状态。  
     */  
    @Override  
    public void operation(String state) {  
        // TODO Auto-generated method stub  
        System.out.println("Intrinsic State = " + this.intrinsicState);  
        System.out.println("Extrinsic State = " + state);  
    }  
}
```

复合享元对象是由单纯享元对象通过复合而成的，因此它提供了add()这样的聚集管理方法。由于一个复合享元对象具有不同的聚集元素，这些聚集元素在复合享元对象被创建之后加入，这本身就意味着复合享元对象的状态是会改变的，因此复合享元对象是不能共享的。

复合享元角色实现了抽象享元角色所规定的接口，也就是operation()方法，这个方法有一个参数，代表复合享元对象的外蕴状态。一个复合享元对象的所有单纯享元对象元素的外蕴状态都是与复合享元对象的外蕴状态相等的；而一个复合享元对象所含有的单纯享元对象的内蕴状态一般是不相等的，不然就没有使用价值了。

```
public class ConcreteCompositeFlyweight implements Flyweight {

    private Map<Character, Flyweight> files = new HashMap<Character, Flyweight>();
    /**
     * 增加一个新的单纯享元对象到聚集中
     */
    public void add(Character key , Flyweight fly){
        files.put(key, fly);
    }
    /**
     * 外蕴状态作为参数传入到方法中
     */
    @Override
    public void operation(String state) {
        Flyweight fly = null;
        for(Object o : files.keySet()){
            fly = files.get(o);
            fly.operation(state);
        }
    }
}
```

享元工厂角色提供两种不同的方法，一种用于提供单纯享元对象，另一种用于提供复合享元对象。

```
public class FlyweightFactory {
    private Map<Character, Flyweight> files = new HashMap<Character, Flyweight>();
}
```

```

ight>());
    /**
     * 复合享元工厂方法
     */
    public Flyweight factory(List<Character> compositeState){
        ConcreteCompositeFlyweight compositeFly = new ConcreteCompositeF
lyweight();

        for(Character state : compositeState){
            compositeFly.add(state, this.factory(state));
        }

        return compositeFly;
    }
    /**
     * 单纯享元工厂方法
     */
    public Flyweight factory(Character state){
        //先从缓存中查找对象
        Flyweight fly = files.get(state);
        if(fly == null){
            //如果对象不存在则创建一个新的Flyweight对象
            fly = new ConcreteFlyweight(state);
            //把这个新的Flyweight对象添加到缓存中
            files.put(state, fly);
        }
        return fly;
    }
}

```

客户端角色

```

public class Client {

    public static void main(String[] args) {
        List<Character> compositeState = new ArrayList<Character>();
        compositeState.add('a');
        compositeState.add('b');
        compositeState.add('c');
        compositeState.add('a');
    }
}

```



```

        compositeState.add('b');

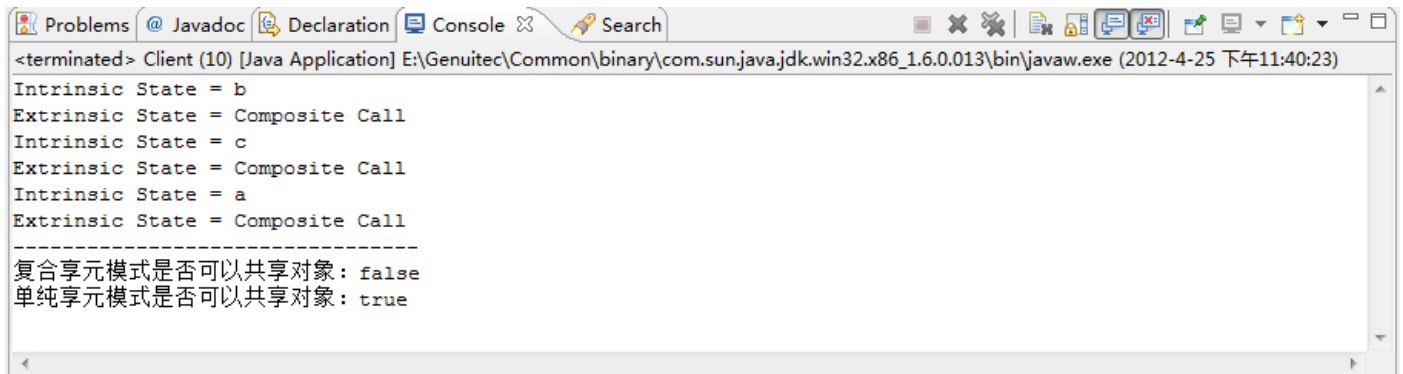
        FlyweightFactory flyFactory = new FlyweightFactory();
        Flyweight compositeFly1 = flyFactory.factory(compositeState);
        Flyweight compositeFly2 = flyFactory.factory(compositeState);
        compositeFly1.operation("Composite Call");

        System.out.println("-----");
        System.out.println("复合享元模式是否可以共享对象：" + (compositeFly1
        == compositeFly2));

        Character state = 'a';
        Flyweight fly1 = flyFactory.factory(state);
        Flyweight fly2 = flyFactory.factory(state);
        System.out.println("单纯享元模式是否可以共享对象：" + (fly1 == fly2))
;
    }
}

```

运行结果如下：



```

<terminated> Client (10) [Java Application] E:\Genuitec\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin\javaw.exe (2012-4-25 下午11:40:23)
Intrinsic State = b
Extrinsic State = Composite Call
Intrinsic State = c
Extrinsic State = Composite Call
Intrinsic State = a
Extrinsic State = Composite Call
-----
复合享元模式是否可以共享对象：false
单纯享元模式是否可以共享对象：true

```

从运行结果可以看出，一个复合享元对象的所有单纯享元对象元素的外蕴状态都是与复合享元对象的外蕴状态相等的。即外运状态都等于Composite Call。

从运行结果可以看出，一个复合享元对象所含有的单纯享元对象的内蕴状态一般是不相等的。即内蕴状态分别为b、c、a。

从运行结果可以看出，复合享元对象是不能共享的。即使用相同的对象compositeState通过工厂分别两次创建出的对象不是同一个对象。

从运行结果可以看出，单纯享元对象是可以共享的。即使用相同的对象state通过工厂分

别两次创建出的对象是同一个对象。

享元模式的优缺点

享元模式的优点在于它大幅度地降低内存中对象的数量。但是，它做到这一点所付出的代价也是很高的：

- 享元模式使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化。
- 享元模式将享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。