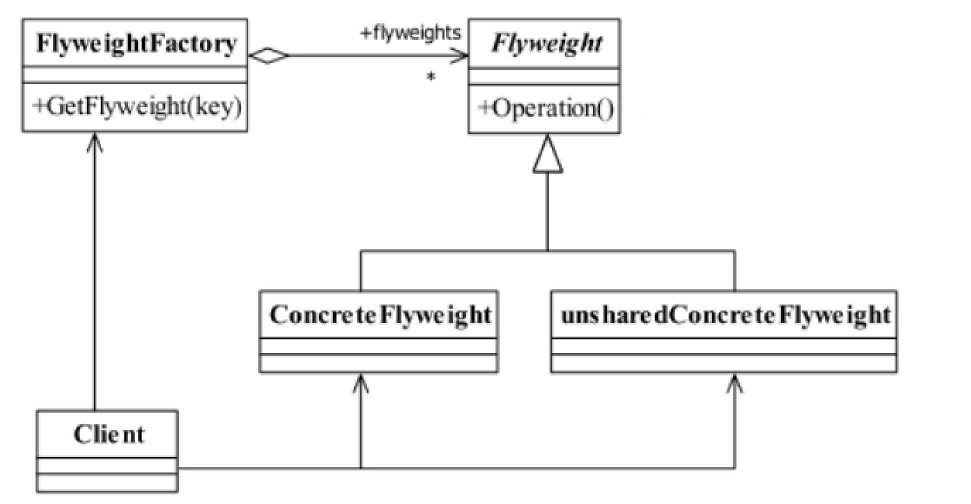
Flyweight pattern

- Usesharing to support large numbers of fine-grained objects efficiently.
  - 细粒度
  - 共享
- 对象拆分
  - 内部状态 不会变 可共享
  - 外部状态 会改变 不可共享
- 优缺点
  - 优点
    - 减少运行实例个数，节省内存
    - 将许多“虚拟”对象的状态集中管理
  - 使用
    - 一个类有很多实例，而这些实例能被同一个方法控制
  - 缺点
    - 单个的逻辑实例，

```
public class FlyweightFactory{
   //定义一个池容器
   private static HashMap<String,Flyweight> pool = new HashMap<String,Flyweight>();
   // 享元工厂
   public static Flyweight getFlyweight(String Extrinsic){
      //需要返回的对象
      Flyweight flyweight=null;
      //在池中没有改对象
      if(pool.containsKey(Extrinsic)){
         flyweight=pool.get(Extrinsic);
      }else{
         //根据外部状态创建享元对象
         flyweight=new ConcreteFlyweight1(Extrinsic);
         //放置到池中
         pool.put(Extrinsic,flyweight);
      }
      return flyweight;
   }
}
```

```
public abstract class Flyweight{
   //内部状态
   private String intrinsic;
   //外部状态
   protected final String Extrinsic;
   //要求享元角色必须接受外部状态
   public Flyweight(String _Extrinsic){
      this.Extrinsic = _Extrinsic;
   }
   //定义业务操作
   public abstract void operate();
   //内部状态的getter/setter
   public String getIntrinsic(){
      return intrinsic;
   }
   public void setIntrinsic(String intrinsic){
      this.intrinsic=intrinsic;
   }
}
```



```
public class ConcreteFlyweight1 extends Flyweight{
   //接受外部状态
   public ConcreteFlyweight1(String _Extrinsic){
      super(_Extrinsic);
   }
   //根据外部状态进行逻辑处理
   public void operate(){
      //业务逻辑
   }
}

public class ConcreteFlyweight2 extends Flyweight{
   //接受外部状态
   public ConcreteFlyweight2(String _Extrinsic){
      super(_Extrinsic);
   }
   //根据外部状态进行逻辑处理
   public void operate(){
      //业务逻辑
   }
}
```