1 享元模式 (Flyweight)

享元模式的用意: 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的维护与管理. 享元对象能实现共享的关键是区分**内部状态**（Internal state）和**外部状态**（External State）.

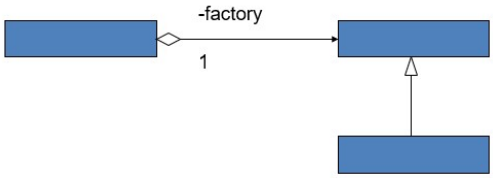
当以下所有条件都满足时，可以考虑使用享元模式:

1. 一个系统有大量的对象。
2. 这些对象消耗大量的内存。
3. 这些对象的状态中的大部分都可以外部化。
4. 软件系统不依赖于这些对象的身份，即这些对象可以是不可分辨的。

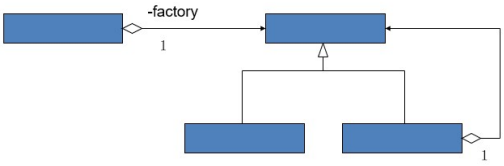
内部状态是存储在享元对象内部的并且是不会随环境改变而有所不同的. 因此，一个享元可以具有内部状态并可以共享.

外部状态是随环境改变而改变的、不可以共享的状态. 享元对象的外部状态必须由客户端保存，并在享元对象被创建之后，在需要使用的时候再传入到享元对象内部. 外部状态不可以影响享元对象的内部状态. 内外部状态是互相独立的.

享元模式的种类: 根据所涉及的享元对象的内部结构，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. 下图是单纯享元模式的结构示意图:



下图所示是复合享元模式的结构示意图:



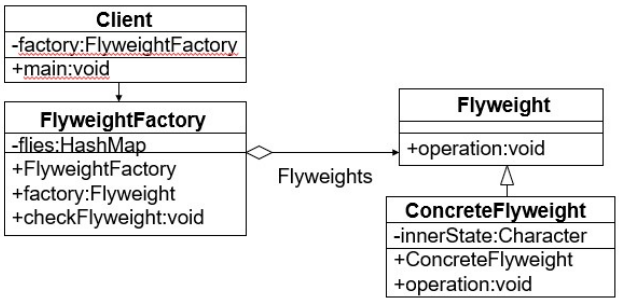
可见，在复合享元模式中，享元对象构成合成模式. 因此，**复合享元模式**实际上是**单纯享元模式**与**合成模式**的组合。

享元模式的应用:

- 享元模式在编辑器系统中大量使用. 一个文本编辑器往往会提供很多种字体，而通常的做法就是将每一个字母作成一个享元对象. 享元对象的内部状态就是这个字母的字符数据，而字母在文本中的位置和字体风格等其它信息则是外部状态.
- 在 Java 语言中，String 类型就使用了享元模式. String 对象是不变对象，一旦创建出来就不能改变.

1.1 单纯享元模式的结构

单纯享元模式中，所有享元对象都是可以共享的. 下图是一个单纯享元模式的简单实现：



单纯享元模式涉及的角色：

- 抽象享元角色：该角色是所有的具体享元类的超类，为这些类规定出需要实现的公共接口.
- 具体享元（ConcreteFlyweight）角色：实现抽象享元角色所规定的接口.
- 享元工厂（FlyweightFactory）角色：本角色负责创建和管理享元角色
- 客户端（Client）角色：本角色需要维护一个对所有享元对象的引用.

```
// 抽象享元角色
abstract public class Flyweight {
  //一个示意性的方法，参数state表示外部状态
  abstract public void operation(String state);
}
```

```
public class ConcreteFlyweight extends Flyweight {
  private Character innerState = null;
  // 构照函数，内部状态作为参数传入
  public ConcreteFlyweight(Character state) {
    this.innerState = state;
  }
  //外部状态作为参数传入方法中，改变方法的行为
  //但并不改变对象的内部状态
  public void operation(String state) {
    System.out.print( "\nIntrinsic State = " + innerState +
      ", Extrinsic State = " + state);
  }
}
```

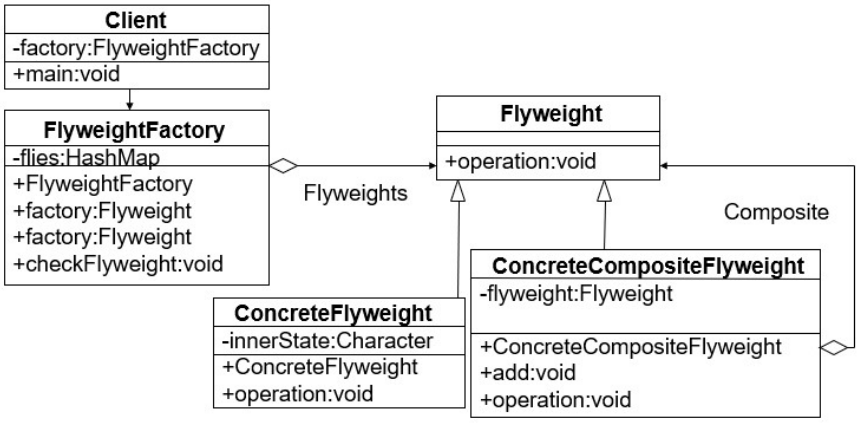
客户端不可以直接将具体享元类实例化，而必须通过一个工厂对象，利用一个 factory() 方法得到享元对象。一般而言，享元对象在整个系统中只有一个，因此可以使用单例模式。当客户端

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部状态。

```
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
public class FlyweightFactory {
  private HashMap flies = new HashMap();
  private Flyweight lnkFlyweight;
  public FlyweightFactory() { } //默认构造函数
  //构造函数,内部状态作为参数传入
  public synchronized Flyweight factory (Character state) {
    if (flies.containsKey(state)) {
      return (Flyweight) flies.get(state);
    } else {
      Flyweight fly = new ConcreteFlyweight(state);
      flies.put(state, fly);
      return fly;
    }
  }
  public void checkFlyweight() { //辅助方法
    Flyweight fly ;
    int i = 0;
    System.out.println("\n==========checkFlyweight()=============");
    for (Iterator it = flies.entrySet().iterator(); it.hasNext();) {
      Map.Entry e = (Map.Entry) it.next();
      System.out.println("Item " + (++i) + " : " + e.getKey());
    }
    System.out.println("==========checkFlyweight()=============");
  }
}
```

1.2 复合享元模式的结构

在上面的单纯享元模式中，所有的享元对象都是单纯享元对象，即都是可以直接共享的. 下面考虑一个较复杂的情况，将一些单纯享元使用合成模式加以组合，形成复合享元对象. 这样的复合本身不能共享，但是它们可以分解成单纯享元对象，而后者可以共享. 复合享元模式的类图如下:



复合享元模式涉及的角色:

- 抽象享元角色：此角色是所有的具体享元类的超类，为这些类规定出需要实现的公共接口.
- 具体享元（ConcreteFlyweight）角色：实现抽象享元角色所规定的接口.
- 复合享元（UnsharableFlyweight）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.
- 享元工厂（FlyweightFactory）角色：负责创建和管理享元角色. 必须保证享元对象可以被系统适当的地共享.
- 客户端（Client）角色：使用享元对象，但需要自行存储所有享元对象的外部状态.

```
abstract public class Flyweight {
  //外部状态作为参数传入到方法中
  abstract public void operation(String state);
}
```

具体享元角色的主要责任:

- 实现了抽象享元角色所声明的接口，也就是 operation() 方法。Operation() 方法接收外部状态作为参数。
- 为内部状态提供存储空间的，在本实现中就是 innerState 属性。享元模式本身对内部状态的存储类型并无要求，这里的内部状态就是 Character 类型，是为了给复合享元的内部状态选做 String 类提供方便。

```
public class ConcreteFlyweight extends Flyweight {
  private Character innerState = null;
  //构造函数,内部状态作为参数传入
  public ConcreteFlyweight(Character state) {
    this.innerState = state;
  }
  //外部状态作为参数传入到方法中
  public void operation(String state) {
```

```
    System.out.print("\nInternal State = " + innerState
      + " Extrinsic State = " + state);
  }
}
```

```
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
public class ConcreteCompositeFlyweight extends Flyweight {
  private HashMap flies = new HashMap(10);
  public ConcreteCompositeFlyweight() { }
  //增加一个新的单纯享元对象到聚集中
  public void add(Character key, Flyweight fly) { flies.put(key, fly); }
  //外部状态作为参数传入到方法中
  public void operation(String extrinsicState) {
    Flyweight fly = null;
    for (Iterator it = flies.entrySet().iterator(); it.hasNext();) {
      Map.Entry e = (Map.Entry) it.next();
      fly = (Flyweight) e.getValue();
      fly.operation(extrinsicState);
    }
  }
}
```

当客户端需要复合享元对象的时候，需要调用享元工厂的 factory() 方法，并传入所需的复合享元对象的所有复合元素的内部状态。

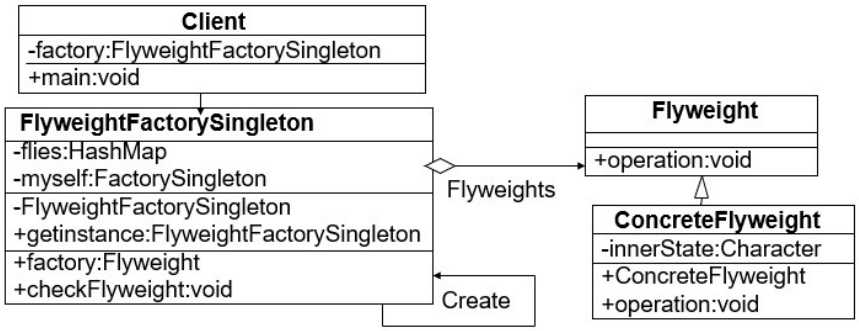
```
public class FlyweightFactory {
  private HashMap flies = new HashMap();
  public FlyweightFactory(){}
  //复合享元工厂方法，所需状态以参数形式传入
  //这个参数恰好可以使用String类型。
  public Flyweight factory(String compositeState) {
    ConcreteCompositeFlyweight compositeFly =
    new ConcreteCompositeFlyweight();
    int length = compositeState.length();
    Character state = null;
    for(int i = 0; i < length; i++) {
      state = new Character (compositeState.charAt(i) );
      compositeFly.add(state, this.factory(state));
    }
```

```
    return compositeFly;
  }
  //单纯享元工厂方法，所需状态以参数形式传入
  public Flyweight factory(Character state) {
    //检查具有此状态的享元是否已经存在
    if (flies.containsKey(state)) { //已存在，直接返回
      return (Flyweight) flies.get(state);
    } else { //不存在，创建新实例
      Flyweight fly = new ConcreteFlyweight(state);
      flies.put(state, fly); //将实例存储到聚集中
      return fly; //将实例返回
    }
  }
}
```

1.3 单例模式实现享元工厂角色

系统往往只需要一个享元工厂的实例，所以享元工厂可以设计成为单例模式.

单纯享元模式中的享元工厂角色:



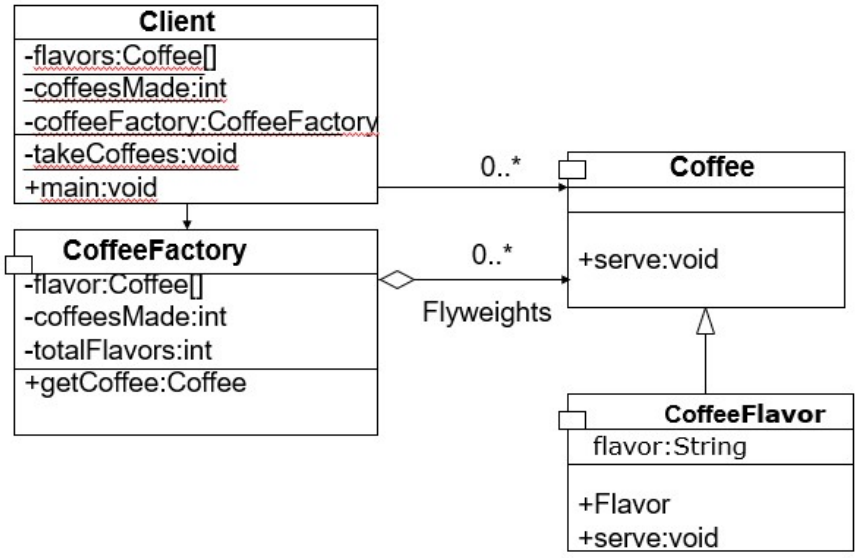
这是一个示意性客户端类的源代码，显示出调用单例工厂对象的 getInstance() 方法，以得到具体享元类.

```
public class ClientSingleton {
  private static FlyweightFactorySingleton factory;
  public static public void main(String[] args) {
    //创建享元工厂对象
    factory = FlyweightFactorySingleton.getInstance();
    //向享元工厂对象请求一个内部状态为'a'的对象
    Flyweight fly = factory.factory(new Character('a'));
    //以参数方式传入外部状态
    fly.operation("First Call");
```

```
10      //向工厂对象请求一个内部状态为’b’的对象
11      fly = factory.factory(new Character('b'));
12      //以参数方式传入外部状态
13      fly.operation("Second Call");
14      //向工厂对象请求一个内部为’a’的享元对象
15      fly = factory.factory(new Character('a'));
16      //以参数方式传入外部状态
17      fly.operation("Third Call");
18
19    }
20  }
```

1.4 例子: 户外咖啡摊

系统的要求: 在这个咖啡摊 (Coffee Stall) 所使用的系统里，有一系列具有不同 “口味 (Flavor)” 的咖啡。客人到摊位上购买咖啡，所有的咖啡均放在台子上，客人自己买到咖啡后就离开摊位。咖啡有内部状态，也就是咖啡的口味；咖啡没有外部状态. 如果系统为每一杯咖啡都创建一个独立的对象那么就需要创建很多小对象，所以只要把咖啡按照种类（即口味）划分，每一种口味的咖啡只创建一个对象，并实行共享。



在这里享元模式所涉及的角色如下:

- 抽象享元（抽象咖啡）角色: 此角色由 Coffee 类扮演，它是所有的具体享元类的超类，为这些类规定出需要实现的公共接口。
- 具体享元（不同 “口味” 的咖啡）角色: 这个角色有 CoffeeFlavor 类扮演，它的实例是可以共享的。每一个不同的咖啡都对应于独立而且唯一的享元对象。

- 享元工厂（咖啡工厂）角色：这个角色由 CoffeeFactory 类扮演，它负责创建所有“口味”的咖啡对象。

客户端并不直接创建任何“口味”的咖啡对象，而是向咖啡工厂提出请求，由咖啡工厂提高一个相应的对象。下面是抽象享元角色的源代码：

```
public abstract class Coffee {
  public abstract void serve(); //将咖啡卖给客人
  public abstract String getFlavor(); //返回咖啡的名字
}
```

“口味”角色的源代码实现了抽象 Order 角色所声明的接口：

```
public class CoffeeFlavor extends Coffee {
  private String flavor;
  public CoffeeFlavor(String flavor) { //内部状态以参数方式传入
    this.flavor = flavor;
  }
  public String getFlavor() { //返回咖啡名字
    return this.flavor;
  }
  public void serve() { //将咖啡卖给客人
    System.out.println("Serving flavor " + flavor );
  }
}
```

所有不同“口味”的咖啡对象都应当由咖啡工厂提供，而不应当由客户端直接创建。咖啡工厂的源代码如下：

```
public class CoffeeFactory {
  private Coffee[] coffees = new Coffee[20];
  private int coffeesMade = 0;
  private int totalFlavors = 0 ;
  //工厂方法，根据所需的口味提供咖啡
  public Coffee getCoffee(String flavorToGet) {
    if (totalFlavors > 0) {
      for (int i = 0; i < totalFlavors ; i++) {
        if (flavorToGet.equals((coffees[i]).getFlavor())) {
          coffeesMade++;
          return coffees[i];
        }
      }
    }
```

```
    coffees[totalFlavors] = new CoffeeFlavor(flavorToGet);
    coffeesMade++;
    return coffees[totalFlavors++];
  }
  //辅助方法，返回创建过的口味对象的个数
  public int getTotalFlavorsMade() { return totalFlavors; }
}
```

下面是系统的客户端角色的源代码，该角色可代表咖啡摊的工作人员。

```
//客户端角色，代表咖啡摊的侍者
public class Client {
  //记录卖出咖啡的总数目
  private static Coffee[] coffees = new CoffeeFlavor[20];
  private static int ordersMade = 0;
  private static CoffeeFactory coffeeFactory;
  //静态方法，提供一杯咖啡
  private static void takeCoffees(String aFlavor) {
    coffees[ordersMade++]=coffeeFactory.
    getOrder(aFlavor);
  }
  public static void main(String[] args) {
    //创建口味工厂
    coffeeFactory = new CoffeeFactory();
    //创建一个个咖啡对象
    takeCoffees("Black Coffee");
    takeCoffees("Capucino");
    takeCoffees("Espresso");
    takeCoffees("Espresso");
    takeCoffees("Capucino");
    takeCoffees("Capucino");
    takeCoffees("Black Coffee");
    takeCoffees("Espresso");
    takeCoffees("Capucino");
    takeCoffees("Black Coffee");
    takeCoffees("Espresso");
    //将所创建的咖啡对象卖给客人
    for (int i = 0; i < coffeesMade; i++) {
      coffees[i].serve();
    }
    //打印出卖出的咖啡总数
```

```
32      System.out.println("\nTotal teaFlavor objects
33        made: "+coffeeFactory.getTotalFlavorsMade());
34    }
35  }
```

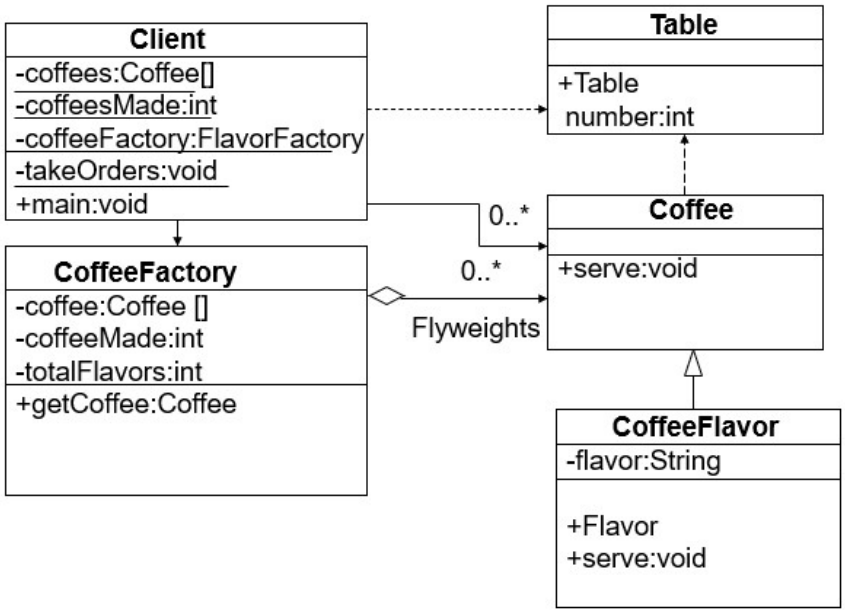
可以看出，这个咖啡摊可以为客人准备最多 20 种不同口味的咖啡。虽然上面的客户端对象叫了 11 杯咖啡，但是所有咖啡的口味却只有三种，即 Capucino、Espresso 和 Black Coffee。

1.5 例子: 咖啡屋

系统的要求: 在前面的咖啡摊项目里，由于没有供客人坐的桌子，所有的咖啡对象均没有考虑环境的影响。咖啡仅有内部状态，也就是咖啡的种类，而没有外部状态。下面考虑一个规模稍稍大一点的咖啡屋 (Coffee Shop) 项目。屋子里有很多桌子供客人坐，系统除了需要提供不同“口味”的咖啡之外，还需要跟踪咖啡被送到哪一个桌位上，因此，咖啡就有了桌子作为外部状态。

系统的设计: 由于外部状态的存在，没有外部状态的单纯享元模式不再符合要求。系统的设计可以利用有外部状态的单纯享元模式。在这里享元模式所涉及的角色与咖啡摊项目的角色都是一样的，除了 Table 角色之外:

- 环境角色：由 Table 类扮演，这就是所有的享元角色所涉及的外部状态。
- 抽象享元角色：此角色由 Coffee 类扮演，它是所有的具体享元类的超类，为这些类规定出需要实现的公共接口。
- 具体享元（咖啡“口味”）角色: 该角色由 CoffeeFlavor 类扮演，它的实例是可以共享的。每一杯不同的咖啡都对应一个独立而且唯一的享元对象。一个咖啡对象在创建出来之后，其内部状态就不再改变。
- 享元工厂角色：该角色由 FlavorFactory 类扮演，它负责创建一个对应的咖啡对象。在客户端提出请求后，咖啡工厂就会检查是否已经有一个对应的对象存在，如已有，直接返回这个已有的对象；反之，创建一个新的对象，并提供给客户端。
- 客户端（Client）角色：客户端并不直接创建任何“口味”的咖啡对象，而是向咖啡工厂提出请求，由咖啡工厂提供一个相应的对象。本角色代表咖啡屋的负责招待客人的侍者，侍者代替客人向调制咖啡的工作人员请求咖啡，后者将调制好的咖啡提供给侍者。



抽象享元角色的源代码如下：

```
1  public abstract class Coffee {
2    //将咖啡卖给客人
3    public abstract void serve(Table table);
4    //返回咖啡名字
5    public abstract String getFlavor();
6  }
```

“口味”角色的源代码如下：

```
1  public class CoffeeFlavor extends Coffee {
2    private String flavor;
3    //构造函数，内部状态以参数方式传入
4    public Flavor(String flavor) { this.flavor = flavor; }
5    //普通方法，返回咖啡名字
6    public String getFlavor() { return this.flavor; }
7    //将咖啡卖给客人
8    public void serve(Table table) {
9      System.out.println("Serving table " + table
10       .getNumber() + " with flavor " + flavor );
11   }
12 }
```

“口味”工厂的源代码：

```
1  public class CoffeeFactory {
```

```
  private Coffee[] coffees = new Coffee[20];
  private int coffeesMade = 0;
  private int totalFlavors = 0 ;
  //工厂方法，根据所需的口味提供咖啡
  public Coffee getCoffee(String flavorToGet) {
    if (totalFlavors > 0) {
      for (int i = 0; i < totalFlavors; i++) {
        if (flavorToGet.equals((coffees[i]).getFlavor())) {
          totalFlavors++;
          return coffees[i];
        }
      }
    }
    coffees[totalFlavors] = new Coffee (flavorToGet);
    coffeesMade++;
    return flavors[totalFlavors++];
  }
  //辅助方法，返回创建过的口味种类的个数
  public int getTotalFlavorsMade() {
    return totalFlavors;
  }
}
```

系统环境角色 Table 类的代码：

```
package com.javapatterns.flyweight.coffeeshop;
public class Table {
  private int number; //桌子号码
  public Table(int number) { //构造函数
    this.number = number;
  }
  public void setNumber(int number) {
    this.number = number;
  }
  public int getNumber() { return number; }
}
```

系统客户端角色代码如下：

```
public class Client {
  //卖出的咖啡总数
  private static Coffee[] coffee s=new Coffee[100];
```

```
4   private static int coffeesMade = 0;
5   private static CoffeeFactory coffeeFactory;
6   //静态方法，提供一杯咖啡
7   private static void takeCoffees(String aFlavor) {
8     coffees[coffeesMade++] = flavorFactory.getCoffee(aFlavor);
9   }
10  public static void main(String[] args) {
11    //创建咖啡工厂对象
12    coffeeFactory = new CoffeeFactory();
13    //创建一个个咖啡对象
14    takeCoffees("Black Coffee");
15    takeCoffees("Capucino");
16    takeCoffees("Espresso");
17    takeCoffees("Espresso");
18    takeCoffees("Capucino");
19    takeCoffees("Capucino");
20    takeCoffees("Black Coffee");
21    takeCoffees("Espresso");
22    takeCoffees("Capucino");
23    takeCoffees("Black Coffee");
24    takeCoffees("Espresso");
25    //将所创建的咖啡对象卖给客人
26    for (int i = 0; i < coffeesMade; i++) {
27      coffees[i].serve(new Table(i));
28    }
29    //打印卖出的咖啡总数
30    System.out.println("\nTotal teaFlavor objects made:"+ flavorFactory.
31      getTotalFlavorsMade());
32  }
33 }
```