

## 《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)。

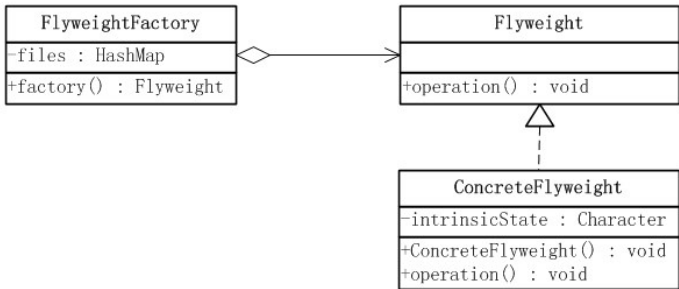
一个内蕴状态是存储在享元对象内部的，并且是不会随环境的改变而有所不同。因此，一个享元可以具有内蕴状态并可以共享。

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，在需要使用的时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内蕴状态，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
```

#### 导航

博客园  
首页  
新随笔  
联系  
订阅 [XML](#)  
管理

#### 统计

随笔 - 32  
文章 - 0  
评论 - 436  
引用 - 0

#### 公告

昵称: java\_my\_life  
园龄: 8年2个月  
粉丝: 2885  
关注: 0  
[+加关注](#)

#### 搜索

找找看

谷歌搜索

#### 常用链接

[我的随笔](#)  
[我的评论](#)  
[我的参与](#)  
[最新评论](#)  
[我的标签](#)

#### 我的标签

[设计模式\(1\)](#)

#### 随笔档案(32)

2017年6月 (3)  
2017年4月 (1)  
2012年8月 (3)  
2012年6月 (6)  
2012年5月 (8)  
2012年4月 (7)  
2012年3月 (4)

#### 积分与排名

积分 - 108239  
排名 - 3738

#### 最新评论

1. Re:《JAVA与模式》之调停者模式  
@人生24k我也是...

--Li\_Zhanj

2. Re:JAVA虚拟机体系结构  
大神啊，膜拜!!

--小猿972j

3. Re:《JAVA与模式》之调停者模式  
一边看着 大话设计模式，一边

结合着博主的文章看，

--人生24k

4. Re:《JAVA与模式》之备忘录模式  
我觉得多重检查点有问题，在恢

复指定检查点的备忘录对象的时候应该检查的是备忘录自身的index，而不是直接根据list索引来获取

--背锅

5. Re:《JAVA与模式》之适配器模式  
来的晚了，鲁智深还是挺厉害

的！

--小路爱学习

#### 阅读排行榜

1. 《JAVA与模式》之适配器模式 (122154)  
2. 《JAVA与模式》之观察者模式 (107131)  
3. 《JAVA与模式》之策略模式 (95664)  
4. 《JAVA与模式》之抽象工厂模式 (74704)  
5. 《JAVA与模式》之装饰模式 (66917)

#### 评论排行榜

```
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");

    }

}
```

1. 《JAVA与模式》之抽象工厂模式(32)
2. 《JAVA与模式》之适配器模式(28)
3. 《JAVA与模式》之装饰模式(24)
4. 《JAVA与模式》之策略模式(22)
5. JAVA虚拟机体系结构(20)

推荐排行榜

1. 《JAVA与模式》之策略模式(73)
2. 《JAVA与模式》之适配器模式(47)
3. 《JAVA与模式》之抽象工厂模式(44)
4. 《JAVA与模式》之装饰模式(41)
5. 《JAVA与模式》之单例模式(36)

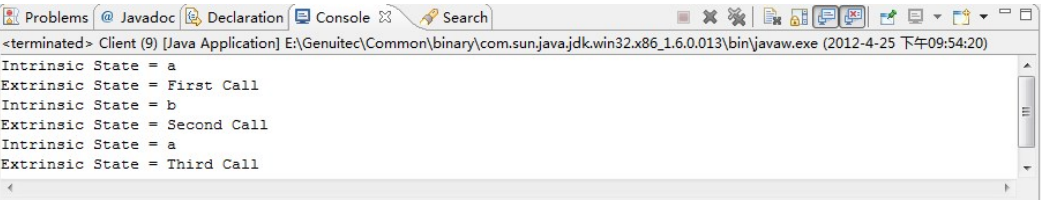
```
fly = factory.factory(new Character('a'));

fly.operation("Third Call");

}

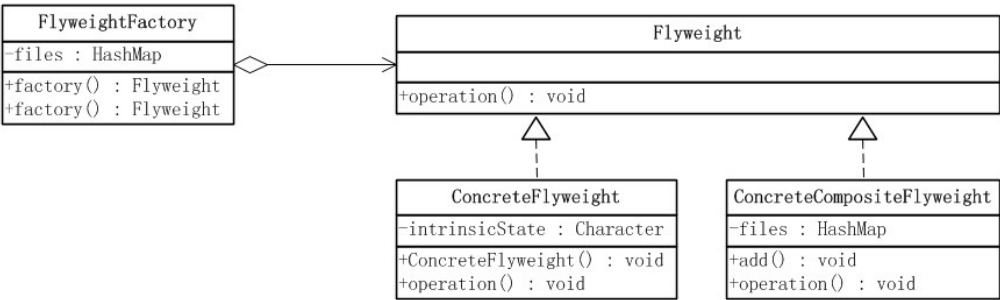
}
```

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:



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>();

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

        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");

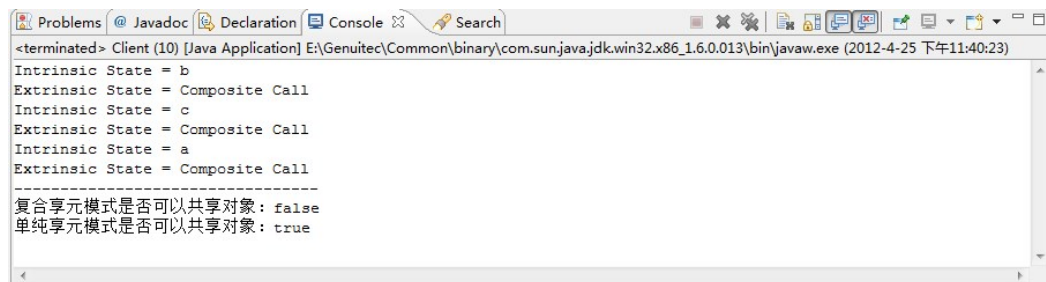
        System.out.println("-----");
        System.out.println("复合享元模式是否可以共享对象:" + (compositeFly1 == compositeFly2));

        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：



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。

好文要顶

关注我

收藏该文



java\_my\_life  
关注 - 0  
粉丝 - 2885

+加关注

15

推荐

0

反对

« 上一篇: {JAVA与模式}之代理模式

» 下一篇: {JAVA与模式}之门面模式

posted on 2012-04-26 13:00 java\_my\_life 阅读(31352) 评论(13) 编辑 收藏

## 评论

#1楼 2012-04-26 13:42 God Is Coder

不错 写的很细  
最好举一个现有的例子，比如哪个开源框架那部分代码使用了这个模式

支持(1) 反对(0)

#2楼[楼主] 2012-04-26 14:15 java\_my\_life

@ 爱公司的程序员  
因为享元模式使得系统会更加复杂，所以实际应用也不是很多。再加上本人水平有限，如果我知道哪个项目或开源框架有使用到这个模式的话一定会写出来的。  
支持(5) 反对(0)

#3楼 2012-06-26 15:51 Hello\_Java\_World

享元模式=单例模式+工厂模式+合成模式  
这种理解 对否  
支持(0) 反对(0)

#4楼 2012-06-26 15:57 Hello\_Java\_World

@ 爱公司的程序员  
享元模式在一般的项目开发中并不常用，而是常常应用于系统底层的开发，以便解决系统的性能问题。

比如：Java中的String类型就是使用了享元模式。

到底系统需要满足什么样的条件才能使用享元模式。对于这个问题，总结出以下几点：

- 1、一个系统中存在着大量的细粒度对象；
- 2、这些细粒度对象耗费了大量的内存。
- 3、这些细粒度对象的状态中的大部分都可以外部化；
- 4、这些细粒度对象可以按照内蕴状态分成很多的组，当把外蕴对象从对象中剔除时，每一个组都可以仅用一个对象代替。
- 5、软件系统不依赖于这些对象的身份，换言之，这些对象可以是不可分辨的。

满足以上的这些条件的系统可以使用享元对象。最后，使用享元模式需要维护一个记录了系统已有的所有享元的哈希表，也称之为对象池，而这也需要耗费一定的资源。因此，应当在有足够多的享元实例可供共享时才值得使用享元模式。

支持(2) 反对(0)

#5楼 2014-05-12 16:46 scansi

请教一个问题，我觉得  
private Map<Character,Flyweight> files = new HashMap<Character,Flyweight>();  
这里的files应该是静态的，否则无法保证在多线程下同样对象只得到一个。  
一般情况下使用享元模式如下，每个工厂类都是在单线程内全新创建的，而不是例子里那种在一个方法内复用工厂类，如果工厂类是单态又失去了享元的意义，测试代码如下：  
new Thread("Thread" + i) {  
public void run() {  
FlyweightFactory factory = new FlyweightFactory();  
Flyweight fly = factory.factory(new Character('a'));  
fly.operation("Third Call");  
}  
}.start();  
vm中看确实是3个对象，而不是2个。

支持(0) 反对(0)

#6楼 2014-05-12 16:53 scansi

又看了下，觉得例子里这种情况，单态可以解决，因为享元单元不是类变量，是方法内的局部变量，所以单态基本等于使用了静态。  
可实际生产情况下一般享元单元如果是局部变量则毫无意义。  
另外在集群部署情况下，享元模式的价值大打折扣，因为不同服务器的jvm不同，最少也是每种类型的对象在每台服务器上有一个。

支持(0) 反对(1)

#7楼 2014-06-12 10:21 牛奶毒咖啡

// Copyright(C) 2000-2003 Yoshinori Oota All rights reserved.

import java.util.\*;

```
public class FlyweightSample {
    static public void main(String[] args) {
        FlyweightFactory factory = new FlyweightFactory();
```

```
        Flyweight shared = factory.GetFlyweight('O');
        shared.Operation();
        shared = factory.GetFlyweight('o');
        shared.Operation();
        shared = factory.GetFlyweight('f');
        shared.Operation();
        shared = factory.GetFlyweight('a');
        shared.Operation();
        shared = factory.GetFlyweight(' ');
        shared.Operation();
```

```
// extrinsicなFlyweightを生成してみよう！
Flyweight unshared = factory.CreateFlyweight("Yoshinori");
unshared.Operation();
    }
}
```

```
class FlyweightFactory {
    private static final int ARRAY_SIZE = 96;
    private ArrayList _intrinsic = new ArrayList(ARRAY_SIZE);
    public FlyweightFactory() {
        for (char i = ' '; i <= '.'; ++i) {
            _intrinsic.add(new SharedFlyweight(i));
        }
        _intrinsic.add(null);
    }
    public Flyweight GetFlyweight(char code) {
        char c = (char)(code - ' ');
        return (Flyweight)_intrinsic.get(c);
    }
    public Flyweight CreateFlyweight(String context) {
        UnsharedFlyweight unshared = new UnsharedFlyweight();
        int length = context.length();
        for (int i = 0; i < length; ++i) {
            char c = (char)(context.charAt(i) - ' ');
            unshared.add((Flyweight)_intrinsic.get(c));
        }
        return unshared;
    }
}
```

```
abstract class Flyweight {
    abstract public void Operation();
}
```

```
class SharedFlyweight extends Flyweight {
    private Character _ascii = null;
    public SharedFlyweight(char code) {
        _ascii = new Character(code);
    }
    public void Operation() {
        System.out.print(_ascii);
    }
}

class UnsharedFlyweight extends Flyweight {
    private LinkedList _flyweights = new LinkedList();
    public void add(Flyweight flyweight) {
        _flyweights.add(flyweight);
    }
    public void Operation() {
        ListIterator it = _flyweights.listIterator();
        while (it.hasNext()) {
            Flyweight flyweight = (Flyweight)it.next();
            flyweight.Operation();
        }
        System.out.println();
    }
}
```

大家能帮我看看这个享元模式的代码吗？对于UnsharedFlyweight的operation()方法到底实现的是什麼，还有对于输出结果是Oota Yoshinori 怎么解释啊

支持(0) 反对(0)

#8楼 2014-06-12 20:28 chenfei0801

```
我觉得单纯享元的那个例子不好，不能说明享元。
因为在 public void operation(String state) {
// TODO Auto-generated method stub
System.out.println("Intrinsic State = " + this.intrinsicState);
System.out.println("Extrinsic State = " + state);
}
```

中，通过内涵状态来说明只创建了一个对象不合理。这地方可以通过在客户端“＝”或者干脆在operation这个方法中大约对象的内存地址来判断对象是否是同一个

支持(0) 反对(0)

#9楼 2015-02-05 09:08 Ydoing

写的不错,看懂了

支持(0) 反对(0)

#10楼 2015-07-19 16:41 九人雅。

复合享元是 ConcreteCompositeFlyweight compositeFly = new ConcreteCompositeFlyweight();得来的，所以＝比较的时候不相等，那如果有一个map维护复合享元，那复合享元是不是就可以共享了？

支持(0) 反对(0)

#11楼 2015-07-21 16:37 月夜归醉

通俗易懂的文章，让我看了之后至少明白享元模式到底是什么回事^\_^

支持(0) 反对(0)

#12楼 2017-02-10 11:07 sck\_nbu

@ scansi  
享元里可以有多个状态，单态无法有多个状态吧？

支持(0) 反对(0)

#13楼 2017-02-10 11:21 sck\_nbu

@ 九人雅。  
我觉得不是可不可以共享的问题，而是想不想共享的问题。

支持(1) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#), [访问](#)网站首页。

相关博文：

- [java设计模式之享元模式](#)
- [Java设计模式之享元模式](#)
- [java 之 享元模式\(大话设计模式\)](#)
- [设计模式之享元模式](#)
- [Java\\_设计模式之享元模式](#)

热门推荐：

- [SQL Server 2012-2016-2017 简体中文版下载和序列号](#)
- [如果看了此文你还不懂傅里叶变换，那就过来掐死我吧【完整版】](#)
- [让任正非愤怒的到底是华为财管团队还是流程本身？](#)
- [HTML中<input>和<textarea>的区别](#)
- [<input type="text">和<textarea>的区别](#)