享元模式——对象共享，避免创建多对象

1. 享元模式介绍

享元模式是对象池的一种实现，它的英文名称叫做Flyweight，代表轻量级的意思。享元模式用来尽可能减少内存使用量，他适合用于可能存在大量重复对象的场景，来缓存可共享的对象，达到对象共享、避免创建过多对象的效果，这样一来就可以提升性能、避免内存溢出等。

享元对象中的部分状态是可以共享，可以共享的状态成为内部状态，内部状态不会随着环境变化；不可共享得状态则称之为外部状态；他们会随着环境的改变而改变。在享元模式中会创建一个对象容器，在经典的享元模式中该容器为一个Map,它的键是享元对象的内部状态，它的值就是享元对象本身。客户端程序通过这个内部状态从享元工厂中获取享元对象，如果有缓存则使用缓存的对象，否则创建一个享元对象并且存入容器中，这样一来就避免了创建过多对象的问题。

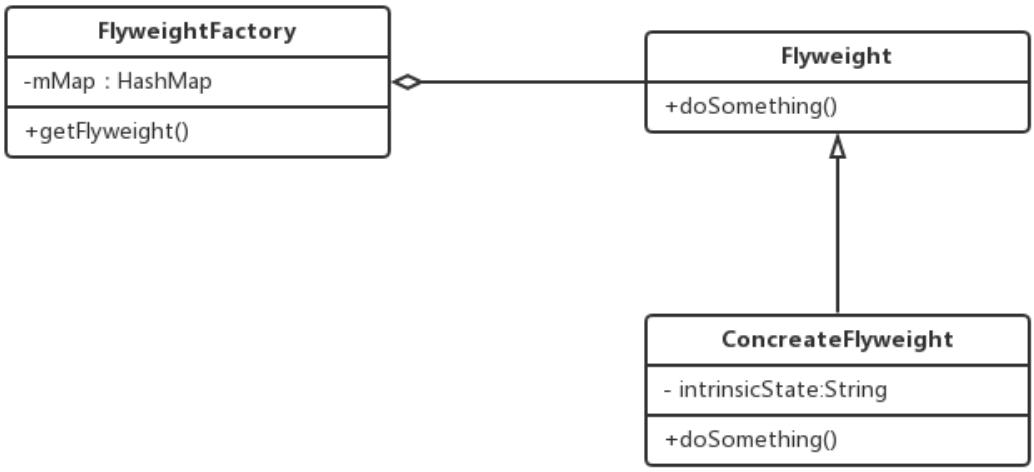
2.享元模式定义

使用共享对象可有效地支持大量的细粒度的对象。

3.享元模式的使用场景

1. 系统中存在大量的相似对象；
2. 细粒度的对象都具备较接近的外部状态，而且内部状态与环境无关，也就是说对象没有特定身份；
3. 需要缓冲池的场景。

4.享元模式的UML类图



角色介绍：

- Flyweight:享元对象抽象基类或接口；
- ConcreteFlyweight:具体的享元对象；
- FlyweightFactory:享元工厂，负责管理享元对象池和创建享元对象；

5.享元模式的简单示例

过年回家买过车票是一件很困难的事，无数人用刷票插件在想服务器端发出请求，对于每一个请求服务器都必须作出应答。在用户设置好出发地和目的地之后，每次请求都返回一个查询的车票结果。为了必然会造成大量重复对象的创建、销毁，使得GC任务繁重、内存占用高居不下。而这类问题通过享元模式就能够得到很好的改善，从A城市到B城市是有限的，车上的铺位也就是软卧、硬卧、坐票3种。我们将这些可以公用的对象缓存起来，在用户查询时优先使用缓存，如果没有缓存则重新创建。这样就将成千上万的对象变为可选择的有限数量。

首先我们创建一个Ticket接口，该接口定义展示车票信息的函数，具体代码如下：

```
1 public interface Ticket {
2         public void showTicketInfo(String bunk);
3 }
```

它的具体的实现类是TrainTicket类，具体代码如下：

```
1
2 public class TrainTicket implements Ticket {
3
4         public String from;// 始发地
5         public String to;// 目的地
6         public String bunk;// 铺位
7         public int price;
8
```

```
        public TrainTicket(String from, String to) {
                this.from = from;
                this.to = to;
        }

        @Override
        public void showTicketInfo(String bunk) {
                price = new Random().nextInt(300);
                System.out.println("购买从" + from + "到" + to + "的" + bunk + "火车票"
                                + ",价格:" + price);
        }

}
```

数据库中表示火车票的信息有出发地、目的地、铺位、价格等字段，在购票用户每次查询时如果没有某种缓存模式，那么返回车票数据的接口实现如下：

```
public class TicketFactory{
   public static Ticket getTicket(String from,String to){
        return new TrainTicket(from,to);
   }
}
```

在TicketFactory的getTicket函数中每次会new一个TrainTicket对象，也就是说如果在短时间内有10000用户求购北京到青岛的车票，那么北京到青岛的车票对象就会被创建1000次，当数据返回之后这些对象变得无用了又会被虚拟机回收。此时就会造成大量的重复对象存在内存中，GC对这些对象的回收也会非常消耗资源。如果用户的请求量很大可能导致系统变得极其缓慢，甚至可能导致OOM。

正如上文所说，享元模式通过消息池的形式有效地减少重复对象的存在。它通过内部状态标识某个种类的对象，外部程序根据这个不会变化的内部状态从消息池中取出对象。使得同一类对象可以被复用，避免大量重复对象。

使用享元模式简单，只需要简单地改造一下TicketFactory,具体代码如下：

```
public class TicketFactory {
        static Map<String, Ticket> sTicketMap = new ConcurrentHashMap<String, Ticket>();

        public static Ticket getTicket(String from, String to) {
                String key = from + "-" + to;
                if (sTicketMap.containsKey(key)) {
                        System.out.println("使用缓存==>" + key);
                        return sTicketMap.get(key);
                } else {
                        System.out.println("创建对象==>" + key);
                        Ticket ticket = new TrainTicket(from, to);
                        sTicketMap.put(key, ticket);
                        return ticket;
                }

        }
}
```

我们在TicketFactory添加一个map容器，并且以出发地+“-”+目的为键、以车票对象作为值存储车票对象。这个map的键就是我们说的内部状态，如果没有缓存则创建一个对象，并且将这个对象缓存到map中，下次再有这类请求时则直接从缓存中获取。这样即使有10000个请求从北京到青岛的车票信息，那么出发地是北京、目的地是青岛的车票对象只有一个，这样就从这个对象从10000减到了1个，避免了大量的内存占用及频繁的GC操作。简单实现代码如下：

```
public class Test {
        public static void main(String[] args) {
                Ticket ticket01 = TicketFactory.getTicket("北京", "青岛");
                ticket01.showTicketInfo("上铺");
                Ticket ticket02 = TicketFactory.getTicket("北京", "青岛");
                ticket01.showTicketInfo("下铺");
                Ticket ticket03 = TicketFactory.getTicket("北京", "青岛");
                ticket01.showTicketInfo("坐票");
        }
}
```

运行结果：

```
创建对象==>北京-青岛
购买从北京到青岛的上铺火车票,价格:270
使用缓存==>北京-青岛
购买从北京到青岛的下铺火车票,价格:249
使用缓存==>北京-青岛
购买从北京到青岛的坐票火车票,价格:256
```

从输出结果可以看到，只有第一次查询车票时创建了一个对象，后续的查询都使用的是消息池中的对象。这其实就是相当于一个对象缓存，避免了对象的重复创建与回收。在这个例子中，内部状态就是出发地和目的地，内部状态不会变化；外部状态就是铺位和价格，价格会随着铺位的变化而变化。

总结

享元模式实现比较简单，但是它的作用在某些场景却是极其重要。它可以大大减少应用程序创建的对象，降低程序内存的占用，增强程序的性能，但它同时也提高了系统的复杂性，需要分离出外部状态和内部状态，而且外部状态具有固化特性，不应该随内部状态改变而改变，否则导致系统的逻辑混乱。

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。为了是对象可以共享，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。