当一个软件系统在运行时产生的对象数量太多，将导致运行代价过高，带来系统性能下降等问题。例如在一个文本字符串中存在很多重复的字符，如果每一个字符都用一个单独的对象来表示，将会占用较多的内存空间，那么我们如何去避免系统中出现大量相同或相似的对象，同时又不影响客户端程序通过面向对象的方式对这些对象进行操作？享元模式正为解决这一类问题而诞生。**享元模式**通过共享技术**实现相同或相似对象的重用**，在逻辑上每一个出现的字符都有一个对象与之对应，然而在物理上它们却共享同一个享元对象，这个对象可以出现在一个字符串的不同地方，相同的字符对象都指向同一个实例，在**享元模式中，存储这些共享实例对象的地方称为享元池**(Flyweight Pool)。我们可以针对每一个不同的字符创建一个享元对象，将其放在享元池中，需要时再从享元池取出。如图14-2所示：

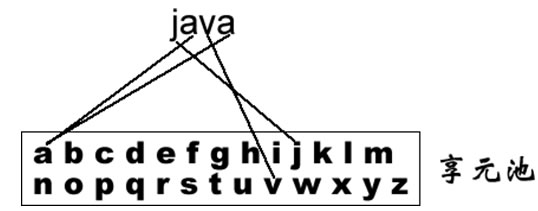


图14-2 字符享元对象示意图

享元模式以共享的方式高效地支持大量细粒度对象的重用，**享元对象能做到共享的关键是区分了内部状态(Intrinsic State)和外部状态(Extrinsic State)**。下面将对享元的内部状态(Intrinsic State)和外部状态(Extrinsic State)进行简单的介绍：

      (1) **内部状态是存储在享元对象内部并且不会随环境改变而改变的状态，内部状态可以共享**。**如字符的内容**，不会随外部环境的变化而变化，无论在任何环境下字符“a”始终是“a”，都不会变成“b”。

      (2) **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通常由客户端保存，并在享元对象被创建之后，需要使用的时候再传入到享元对象内部。一个外部状态与另一个外部状态之间是相互独立的。**如字符的颜色**，可以在不同的地方有不同的颜色，例如有的“a”是红色的，有的“a”是绿色的，**字符的大小**也是如此，有的“a”是五号字，有的“a”是四号字。而且字符的**颜色**和**大小**是两个独立的外部状态，它们可以独立变化，相互之间没有影响，客户端可以在使用时将外部状态注入享元对象中。

      正因为区分了内部状态和外部状态，我们可以将具有相同内部状态的对象存储在享元池中，享元池中的对象是可以实现共享的，需要的时候就将对象从享元池中取出，实现对象的复用。通过向取出的对象注入不同的外部状态，可以得到一系列相似的对象，而这些对象在内存中实际上只存储一份。

享元模式定义如下：

**享元模式(Flyweight Pattern)**：运用共享技术有效地支持大量细粒度对象的复用。系统只使用少量的对象，而这些对象都很相似，状态变化很小，可以实现对象的多次复用。由于享元模式要求能够共享的对象必须是细粒度对象，因此它又称为轻量级模式，它是一种对象结构型模式。

享元模式**结构较为复杂，一般结合工厂模式一起使用**，在它的结构图中包含了一个**享元工厂类**，其结构图如图14-3所示：

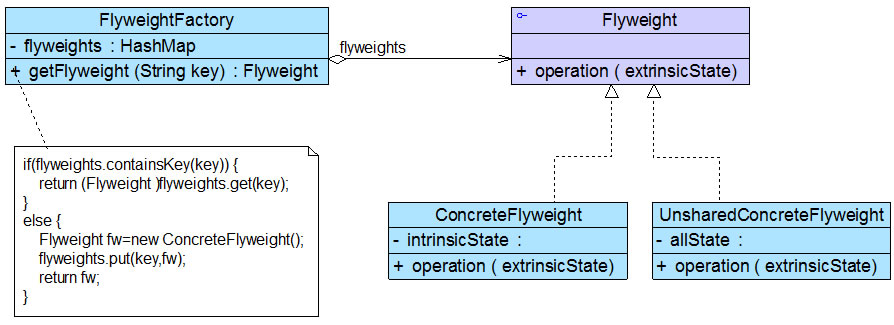


图14-3 享元模式结构图

在享元模式结构图中包含如下几个角色：

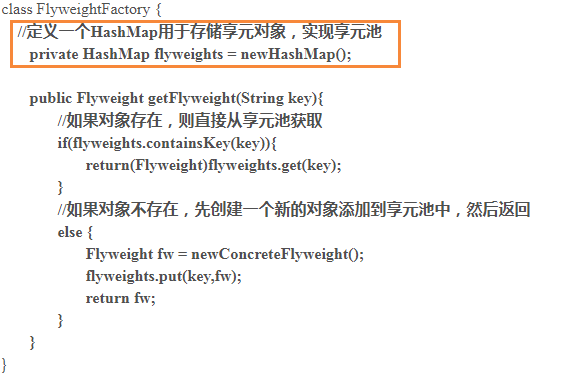
      ● **Flyweight（抽象享元类）**：通常是一个接口或抽象类，在抽象享元类中声明了具体享元类公共的方法，这些方法可以向外界提供享元对象的内部数据（内部状态），同时也可以通过这些方法来设置外部数据（外部状态）。

      ● **ConcreteFlyweight（具体享元类）**：它实现了抽象享元类，其实例称为享元对象；在具体享元类中为内部状态提供了存储空间。通常我们可以结合单例模式来设计具体享元类，为每一个具体享元类提供唯一的享元对象。

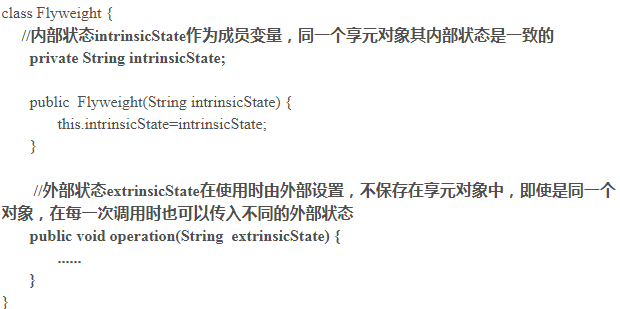
      ● **UnsharedConcreteFlyweight（非共享具体享元类）**：并不是所有的抽象享元类的子类都需要被共享，不能被共享的子类可设计为非共享具体享元类；当需要一个非共享具体享元类的对象时可以直接通过实例化创建。

      ● **FlyweightFactory（享元工厂类）**：享元工厂类用于创建并管理享元对象，它针对抽象享元类编程，将各种类型的具体享元对象存储在一个享元池中，**享元池一般设计为一个存储“键值对”的集合（也可以是其他类型的集合）**，可以结合工厂模式进行设计；当用户请求一个具体享元对象时，享元工厂提供一个存储在享元池中已创建的实例或者创建一个新的实例（如果不存在的话），返回新创建的实例并将其存储在享元池中。

在享元模式中引入了享元工厂类，享元工厂类的作用**在于提供一个用于存储享元对象的享元池**，当用户需要对象时，首先从享元池中获取，如果享元池中不存在，则创**建一个新的享元对象返回给用户，并在享元池中保存该新增对象**。典型的享元工厂类的代码如下：



享元类的设计是享元模式的要点之一，**在享元类中要将内部状态和外部状态分开处理**，通常将**内部状态作为享元类的成员变量**，而**外部状态通过注入的方式添加到享元类中**。典型的享元类代码如下所示：



完整解决方案

为了节约存储空间，提高系统性能，Sunny公司开发人员使用享元模式来设计围棋软件中的棋子，其基本结构如图14-4所示：

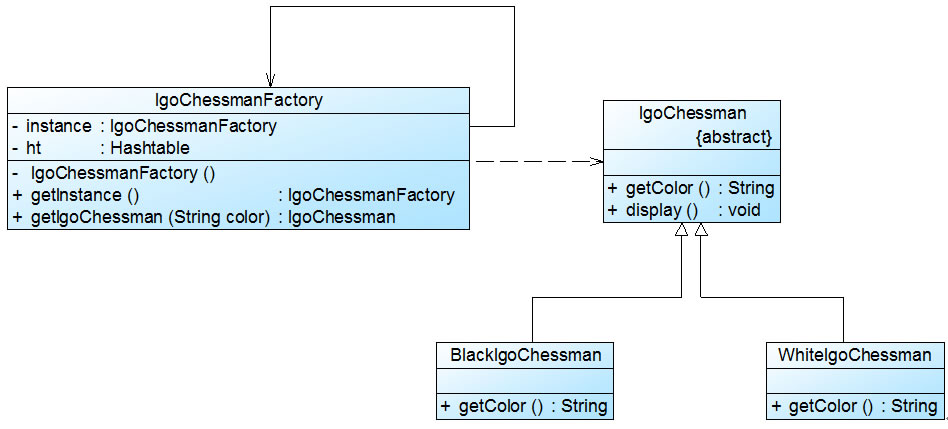
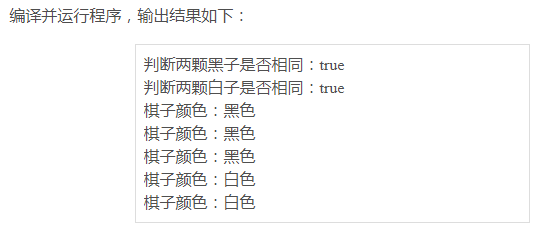


图14-4 围棋棋子结构图

FlyWeightFactory（享元工厂类）：IgoChessmanFactory

FlyWeight（抽象享元类）：IgoChessman

ConcreteFlyWeight（具体享元类）：BlackIgoChessman、WhiteIgoChessman



从输出结果可以看出，虽然我们获取了三个黑子对象和两个白子对象，但是它们的内存地址相同，也就是说，它们实际上是同一个对象。在实现享元工厂类时我们使用了**单例模式（饿汉模式）**和**简单工厂模式**，确保了享元工厂对象的唯一性，并提供工厂方法来向客户端返回享元对象。

带外部状态的解决方案

Sunny软件公司开发人员通过对围棋棋子进行进一步分析，发现虽然黑色棋子和白色棋子可以共享，但是它们将显示在棋盘的不同位置，如何让相同的黑子或者白子能够多次重复显示且位于一个棋盘的不同地方？解决方法就是将棋子的**位置定义为棋子的一个外部状态**，在需要时再进行设置。因此，我们在图14-4中增加了一个新的类Coordinates（坐标类），用于存储每一个棋子的位置，修改之后的结构图如图14-5所示：

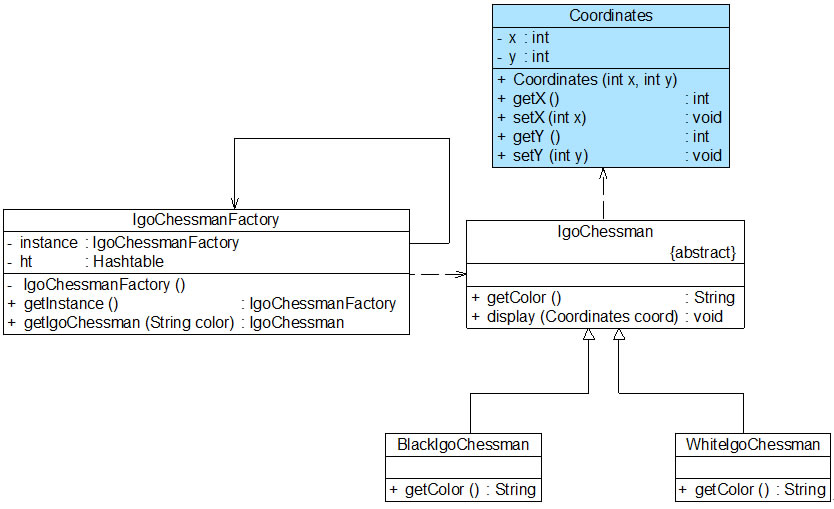
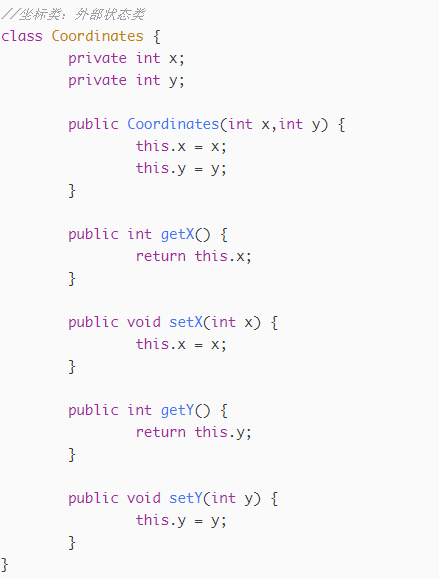
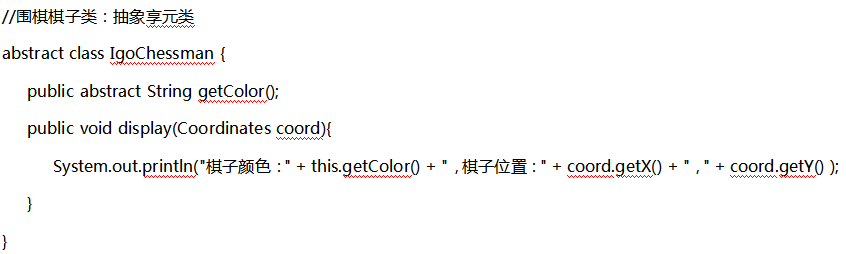
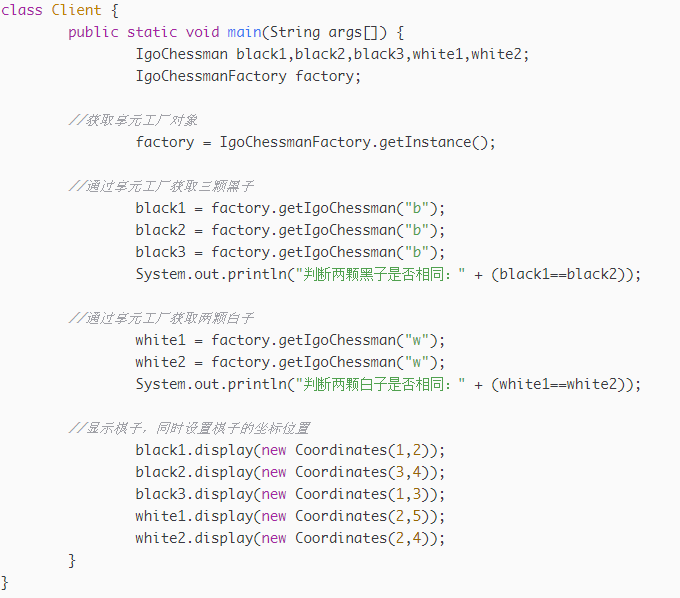


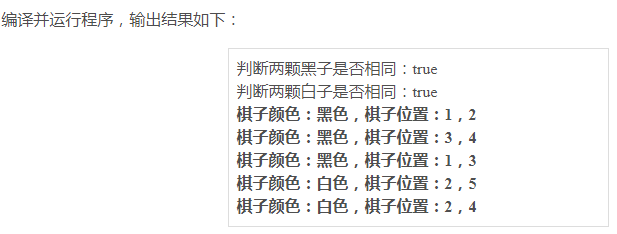
图14-5 引入外部状态之后的围棋棋子结构图

在图14-5中，除了增加一个坐标类Coordinates以外，**抽象享元类**IgoChessman中的**display()方法**也将对应**增加一个Coordinates类型的参数**，用于在显示棋子时指定其坐标，Coordinates类和修改之后的IgoChessman类的代码如下所示：







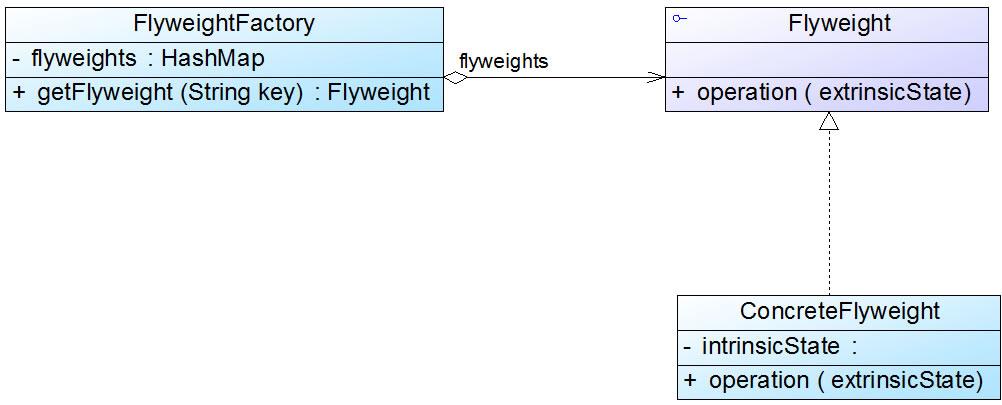


单纯享元模式和复合享元模式

标准的享元模式结构图中既包含可以共享的具体享元类，也包含不可以共享的非共享具体享元类。但是在实际使用过程中，我们有时候会用到两种特殊的享元模式：**单纯享元模式**和**复合享元模式**，下面将对这两种特殊的享元模式进行简单的介绍：

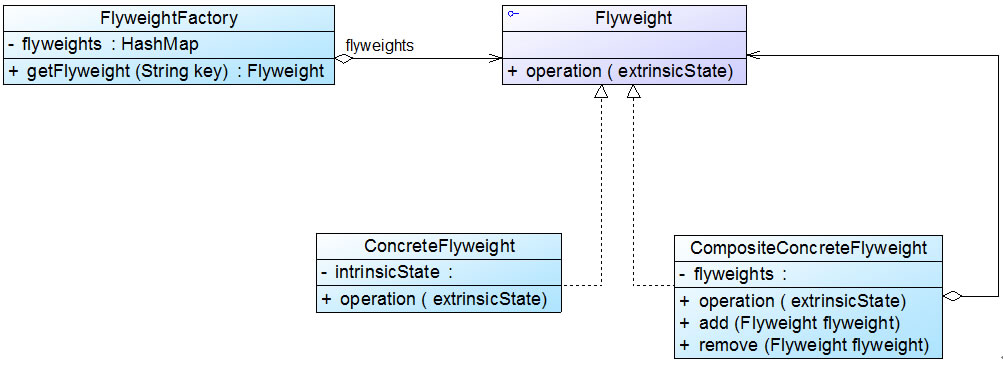
1.单纯享元模式

       在单纯享元模式中，所有的具体享元类都是可以共享的，**不存在非共享具体享元类**。单纯享元模式的结构如图14-6所示：



2.复合享元模式

       将一些**单纯享元对象使用组合模式加以组合，还可以形成复合享元对象**，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。复合享元模式的结构如图14-7所示：



通过复合享元模式，可以确保**复合享元类**CompositeConcreteFlyweight中所**包含的每个单纯享元类**ConcreteFlyweight都具有相同的外部状态，而这些单纯享元的内部状态往往可以不同。如果希望为多个内部状态不同的享元对象设置相同的外部状态，可以考虑使用复合享元模式。

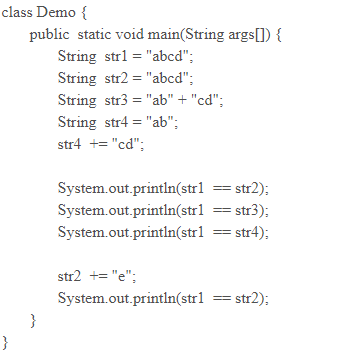
1.享元模式通常需要和其他模式一起联用，几种常见的联用方式如下：

(1)在享元模式的享元工厂类中通常提供一个静态的工厂方法用于返回享元对象，使用**简单工厂模式**来生成享元对象。

       (2)在一个系统中，通常只有唯一一个享元工厂，因此可以使用单例模式进行享元工厂类的设计。

       (3)享元模式可以结合组合模式形成复合享元模式，统一对多个享元对象设置外部状态。

2.享元模式与String类，JDK类库中的String类使用了享元模式，我们通过如下代码来加以说明：



在Java语言中，如果每次执行类似String str1="abcd"的操作时都创建一个新的字符串对象将导致内存开销很大，因此如果第一次创建了内容为"abcd"的字符串对象str1，下一次再创建内容相同的字符串对象str2时会将它的引用指向"abcd"，不会重新分配内存空间，从而实现了"abcd"在内存中的共享。上述代码输出结果如下：

true、true、false、false

       可以看出，前两个输出语句均为true，说明str1、str2、str3在内存中引用了相同的对象；如果有一个字符串str4，其初值为"ab"，**再对它进行操作str4 += "cd"，此时虽然str4的内容与str1相同，但是由于str4的初始值不同，在创建str4时重新分配了内存，所以第三个输出语句结果为false**；最后一个输出语句结果也为false，说明当对str2进行修改时将创建一个新的对象，修改工作在新对象上完成，而原来引用的对象并没有发生任何改变，str1仍然引用原有对象，而str2引用新对象，str1与str2引用了两个完全不同的对象。

**享元模式总结**

当系统中存在大量相同或者相似的对象时，享元模式是一种较好的解决方案，它通过共享技术实现相同或相似的细粒度对象的复用，从而节约了内存空间，提高了系统性能。相比其他结构型设计模式，享元模式的使用频率并不算太高，但是作为一种以“节约内存，提高性能”为出发点的设计模式，它在软件开发中还是得到了一定程度的应用。

1.主要优点

(1) 可以极大减少内存中对象的数量，使得相同或相似对象在内存中只保存一份，从而可以节约系统资源，提高系统性能。

      (2) 享元模式的外部状态相对独立，而且不会影响其内部状态，从而使得享元对象可以在不同的环境中被共享。

2.主要缺点

  (1) 享元模式使得系统变得复杂，需要分离出内部状态和外部状态，这使得程序的逻辑复杂化。

     (2) 为了使对象可以共享，享元模式需要将享元对象的部分状态外部化，而读取外部状态将使得运行时间变长。

3.适用场景

(1) 一个系统有大量相同或者相似的对象，造成内存的大量耗费。

     (2) 对象的大部分状态都可以外部化，可以将这些外部状态传入对象中。

     (3) 在使用享元模式时需要维护一个存储享元对象的享元池，而这需要耗费一定的系统资源，因此，应当在需要多次重复使用享元对象时才值得使用享元模式。

[享元模式](https://blog.csdn.net/LoveLion/article/details/17517213)

[实现对象的复用——享元模式（一）：围棋棋子的设计，享元模式概述（上）](https://blog.csdn.net/lovelion/article/details/7667781)

[实现对象的复用——享元模式（二）：享元模式概述（下）](https://blog.csdn.net/lovelion/article/details/7667810)

[实现对象的复用——享元模式（三）：围棋棋子的享元模式解决方案](https://blog.csdn.net/lovelion/article/details/7667839)

[实现对象的复用——享元模式（四）：带外部状态的围棋棋子解决方案](https://blog.csdn.net/lovelion/article/details/7667860)

[实现对象的复用——享元模式（五）：单纯享元模式和复合享元模式，关于享元模式的几点补充，享元模式总结](https://blog.csdn.net/lovelion/article/details/7667901)