

54 | 享元模式（上）：如何利用享元模式优化文本编辑器的内存占用？

2020-03-06 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 11:09 大小 10.22M



上一节课中，我们讲了组合模式。组合模式并不常用，主要用在数据能表示成树形结构、能通过树的遍历算法来解决的场景中。今天，我们再来学习一个不那么常用的模式，**享元模式**（Flyweight Design Pattern）。这也是我们要学习的最后一个结构型模式。

跟其他所有的设计模式类似，享元模式的原理和实现也非常简单。今天，我会通过棋牌游戏和文本编辑器两个实际的例子来讲解。除此之外，我还会讲到它跟单例、缓存、对象池的区别和联系。在下一节课中，我会带你剖析一下享元模式在 Java Integer、String 中的，☆
用。

话不多说，让我们正式开始今天的学习吧！

享元模式原理与实现


所谓“享元”，顾名思义就是被共享的单元。享元模式的意图是复用对象，节省内存，前提是享元对象是不可变对象。

具体来讲，当一个系统中存在大量重复对象的时候，如果这些重复的对象是不可变对象，我们就可以利用享元模式将对象设计成享元，在内存中只保留一份实例，供多处代码引用。这样可以减少内存中对象的数量，起到节省内存的目的。实际上，不仅仅相同对象可以设计成享元，对于相似对象，我们也可以将这些对象中相同的部分（字段）提取出来，设计成享元，让这些大量相似对象引用这些享元。

这里我稍微解释一下，定义中的“不可变对象”指的是，一旦通过构造函数初始化完成之后，它的状态（对象的成员变量或者属性）就不会再被修改了。所以，不可变对象不能暴露任何 `set()` 等修改内部状态的方法。之所以要求享元是不可变对象，那是因为它会被多处代码共享使用，避免一处代码对享元进行了修改，影响到其他使用它的代码。

接下来，我们通过一个简单的例子解释一下享元模式。

假设我们在开发一个棋牌游戏（比如象棋）。一个游戏厅中有成千上万个“房间”，每个房间对应一个棋局。棋局要保存每个棋子的数据，比如：棋子类型（将、相、士、炮等）、棋子颜色（红方、黑方）、棋子在棋局中的位置。利用这些数据，我们就能显示一个完整的棋盘给玩家。具体的代码如下所示。其中，`ChessPiece` 类表示棋子，`ChessBoard` 类表示一个棋局，里面保存了象棋中 30 个棋子的信息。

 复制代码

```
1 public class ChessPiece { // 棋子
2     private int id;
3     private String text;
4     private Color color;
5     private int positionX;
6     private int positionY;
7
8     public ChessPiece(int id, String text, Color color, int positionX, int positionY) {
9         this.id = id;
10        this.text = text;
11        this.color = color;
12        this.positionX = positionX;
13        this.positionY = positionY;
14    }
15 }
```

```

16     public static enum Color {
17         RED, BLACK
18     }
19
20     // ...省略其他属性和getter/setter方法...
21 }
22
23 public class ChessBoard { //棋局
24     private Map<Integer, ChessPiece> chessPieces = new HashMap<>();
25
26     public ChessBoard() {
27         init();
28     }
29
30     private void init() {
31         chessPieces.put(1, new ChessPiece(1, "車", ChessPiece.Color.BLACK, 0, 0));
32         chessPieces.put(2, new ChessPiece(2, "馬", ChessPiece.Color.BLACK, 0, 1));
33         //...省略摆放其他棋子的代码...
34     }
35
36     public void move(int chessPieceId, int toPositionX, int toPositionY) {
37         //...省略...
38     }
39 }

```


为了记录每个房间当前的棋局情况，我们需要给每个房间都创建一个 ChessBoard 棋局对象。因为游戏大厅中有成千上万的房间（实际上，百万人同时在线的游戏大厅也有很多），那保存这么多棋局对象就会消耗大量的内存。有没有什么办法来节省内存呢？

这个时候，享元模式就可以派上用场了。像刚刚的实现方式，在内存中会有大量的相似对象。这些相似对象的 id、text、color 都是相同的，唯独 positionX、positionY 不同。实际上，我们可以将棋子的 id、text、color 属性拆分出来，设计成独立的类，并且作为享元供多个棋盘复用。这样，棋盘只需要记录每个棋子的位置信息就可以了。具体的代码实现如下所示：

```

1 // 享元类
2 public class ChessPieceUnit {
3     private int id;
4     private String text;
5     private Color color;
6
7     public ChessPieceUnit(int id, String text, Color color) {
8         this.id = id;

```

 复制代码

```

9     this.text = text;
10    this.color = color;
11 }
12
13 public static enum Color {
14     RED, BLACK
15 }
16
17 // ...省略其他属性和getter方法...
18 }
19
20 public class ChessPieceUnitFactory {
21     private static final Map<Integer, ChessPieceUnit> pieces = new HashMap<>();
22
23     static {
24         pieces.put(1, new ChessPieceUnit(1, "車", ChessPieceUnit.Color.BLACK));
25         pieces.put(2, new ChessPieceUnit(2, "馬", ChessPieceUnit.Color.BLACK));
26         //...省略摆放其他棋子的代码...
27     }
28
29     public static ChessPieceUnit getChessPiece(int chessPieceId) {
30         return pieces.get(chessPieceId);
31     }
32 }
33
34 public class ChessPiece {
35     private ChessPieceUnit chessPieceUnit;
36     private int positionX;
37     private int positionY;
38
39     public ChessPiece(ChessPieceUnit unit, int positionX, int positionY) {
40         this.chessPieceUnit = unit;
41         this.positionX = positionX;
42         this.positionY = positionY;
43     }
44     // 省略getter、setter方法
45 }
46
47 public class ChessBoard {
48     private Map<Integer, ChessPiece> chessPieces = new HashMap<>();
49
50     public ChessBoard() {
51         init();
52     }
53
54     private void init() {
55         chessPieces.put(1, new ChessPiece(
56             ChessPieceUnitFactory.getChessPiece(1), 0,0));
57         chessPieces.put(1, new ChessPiece(
58             ChessPieceUnitFactory.getChessPiece(2), 1,0));
59         //...省略摆放其他棋子的代码...
60     }

```

```
61
62     public void move(int chessPieceId, int toPositionX, int toPositionY) {
63         //...省略...
64     }
65 }
```

在上面的代码实现中，我们利用工厂类来缓存 ChessPieceUnit 信息（也就是 id、text、color）。通过工厂类获取到的 ChessPieceUnit 就是享元。所有的 ChessBoard 对象共享这 30 个 ChessPieceUnit 对象（因为象棋中只有 30 个棋子）。在使用享元模式之前，记录 1 万个棋局，我们要创建 30 万（30*1 万）个棋子的 ChessPieceUnit 对象。利用享元模式，我们只需要创建 30 个享元对象供所有棋局共享使用即可，大大节省了内存。

那享元模式的原理讲完了，我们来总结一下它的代码结构。实际上，它的代码实现非常简单，主要是通过工厂模式，在工厂类中，通过一个 Map 来缓存已经创建过的享元对象，来达到复用的目的。


享元模式在文本编辑器中的应用

弄懂了享元模式的原理和实现之后，我们再来看另外一个例子，也就是文章标题中给出的：如何利用享元模式来优化文本编辑器的内存占用？

你可以把这里提到的文本编辑器想象成 Office 的 Word。不过，为了简化需求背景，我们假设这个文本编辑器只实现了文字编辑功能，不包含图片、表格等复杂的编辑功能。对于简化之后的文本编辑器，我们要在内存中表示一个文本文件，只需要记录文字和格式两部分信息就可以了，其中，格式又包括文字的字体、大小、颜色等信息。

尽管在实际的文档编写中，我们一般都是按照文本类型（标题、正文.....）来设置文字的格式，标题是一种格式，正文是另一种格式等等。但是，从理论上讲，我们可以给文本文件中的每个文字都设置不同的格式。为了实现如此灵活的格式设置，并且代码实现又不过于太复杂，我们把每个文字都当作一个独立的对象来看待，并且在其中包含它的格式信息。具体的代码示例如下所示：

```
1 public class Character { //文字
2     private char c;
3
4     private Font font;
```

 复制代码

```

5     private int size;
6     private int colorRGB;
7
8     public Character(char c, Font font, int size, int colorRGB) {
9         this.c = c;
10        this.font = font;
11        this.size = size;
12        this.colorRGB = colorRGB;
13    }
14 }
15
16 public class Editor {
17     private List<Character> chars = new ArrayList<>();
18
19     public void appendCharacter(char c, Font font, int size, int colorRGB) {
20         Character character = new Character(c, font, size, colorRGB);
21         chars.add(character);
22     }
23 }

```


在文本编辑器中，我们每敲一个文字，都会调用 Editor 类中的 appendCharacter() 方法，创建一个新的 Character 对象，保存到 chars 数组中。如果一个文本文件中，有上万、十几万、几十万的文字，那我们就要在内存中存储这么多 Character 对象。那有没有办法可以节省一点内存呢？

实际上，在一个文本文件中，用到的字体格式不会太多，毕竟不大可能有人把每个文字都设置成不同的格式。所以，对于字体格式，我们可以将它设计成享元，让不同的文字共享使用。按照这个设计思路，我们对上面的代码进行重构。重构后的代码如下所示：

```

1 public class CharacterStyle {
2     private Font font;
3     private int size;
4     private int colorRGB;
5
6     public CharacterStyle(Font font, int size, int colorRGB) {
7         this.font = font;
8         this.size = size;
9         this.colorRGB = colorRGB;
10    }
11
12    @Override
13    public boolean equals(Object o) {
14        CharacterStyle otherStyle = (CharacterStyle) o;
15        return font.equals(otherStyle.font)

```

 复制代码

```

16         && size == otherStyle.size
17         && colorRGB == otherStyle.colorRGB;
18     }
19 }
20
21 public class CharacterStyleFactory {
22     private static final List<CharacterStyle> styles = new ArrayList<>();
23
24     public static CharacterStyle getStyle(Font font, int size, int colorRGB) {
25         CharacterStyle newStyle = new CharacterStyle(font, size, colorRGB);
26         for (CharacterStyle style : styles) {
27             if (style.equals(newStyle)) {
28                 return style;
29             }
30         }
31         styles.add(newStyle);
32         return newStyle;
33     }
34 }
35
36 public class Character {
37     private char c;
38     private CharacterStyle style;
39
40     public Character(char c, CharacterStyle style) {
41         this.c = c;
42         this.style = style;
43     }
44 }
45
46 public class Editor {
47     private List<Character> chars = new ArrayList<>();
48
49     public void appendCharacter(char c, Font font, int size, int colorRGB) {
50         Character character = new Character(c, CharacterStyleFactory.getStyle(font
51         chars.add(character);
52     }
53 }

```

享元模式 vs 单例、缓存、对象池

在上面的讲解中，我们多次提到“共享”“缓存”“复用”这些字眼，那它跟单例、缓存、对象池这些概念有什么区别呢？我们来简单对比一下。

我们先来看享元模式跟单例的区别。

在单例模式中，一个类只能创建一个对象，而在享元模式中，一个类可以创建多个对象，每个对象被多处代码引用共享。实际上，享元模式有点类似于之前讲到的单例的变体：多例。

我们前面也多次提到，区别两种设计模式，不能光看代码实现，而是要看设计意图，也就是要解决的问题。尽管从代码实现上来看，享元模式和多例有很多相似之处，但从设计意图上来看，它们是完全不同的。应用享元模式是为了对象复用，节省内存，而应用多例模式是为了限制对象的个数。

我们再来看享元模式跟缓存的区别。

在享元模式的实现中，我们通过工厂类来“缓存”已经创建好的对象。这里的“缓存”实际上是“存储”的意思，跟我们平时所说的“数据库缓存”“CPU 缓存”“MemCache 缓存”是两回事。我们平时所讲的缓存，主要是为了提高访问效率，而非复用。

最后我们来看享元模式跟对象池的区别。

对象池、连接池（比如数据库连接池）、线程池等也是为了复用，那它们跟享元模式有什么区别呢？

你可能对连接池、线程池比较熟悉，对对象池比较陌生，所以，这里我简单解释一下对象池。像 C++ 这样的编程语言，内存的管理是由程序员负责的。为了避免频繁地进行对象创建和释放导致内存碎片，我们可以预先申请一片连续的内存空间，也就是这里说的对象池。每次创建对象时，我们从对象池中直接取出一个空闲对象来使用，对象使用完成之后，再放回到对象池中以供后续复用，而非直接释放掉。

虽然对象池、连接池、线程池、享元模式都是为了复用，但是，如果我们再细致地抠一抠“复用”这个字眼的话，对象池、连接池、线程池等池化技术中的“复用”和享元模式中的“复用”实际上是不同的概念。

池化技术中的“复用”可以理解为“重复使用”，主要目的是节省时间（比如从数据库池中取一个连接，不需要重新创建）。在任意时刻，每一个对象、连接、线程，并不会被多处使用，而是被一个使用者独占，当使用完成之后，放回到池中，再由其他使用者重复利用。享元模式中的“复用”可以理解为“共享使用”，在整个生命周期中，都是被所有使用者共享的，主要目的是节省空间。

重点回顾

好了，今天的内容到此就讲完了。我们来一块总结回顾一下，你需要重点掌握的内容。

1. 享元模式的原理

所谓“享元”，顾名思义就是被共享的单元。享元模式的意图是复用对象，节省内存，前提是享元对象是不可变对象。具体来讲，当一个系统中存在大量重复对象的时候，我们就可以利用享元模式，将对象设计成享元，在内存中只保留一份实例，供多处代码引用，这样可以减少内存中对象的数量，以起到节省内存的目的。实际上，不仅仅相同对象可以设计成享元，对于相似对象，我们也可以将这些对象中相同的部分（字段），提取出来设计成享元，让这些大量相似对象引用这些享元。

2. 享元模式的实现

享元模式的代码实现非常简单，主要是通过工厂模式，在工厂类中，通过一个 Map 或者 List 来缓存已经创建好的享元对象，以达到复用的目的。

3. 享元模式 VS 单例、缓存、对象池

我们前面也多次提到，区别两种设计模式，不能光看代码实现，而是要看设计意图，也就是要解决的问题。这里的区别也不例外。

我们可以用简单几句话来概括一下它们之间的区别。应用单例模式是为了保证对象全局唯一。应用享元模式是为了实现对象复用，节省内存。缓存是为了提高访问效率，而非复用。池化技术中的“复用”理解为“重复使用”，主要是为了节省时间。

课堂讨论

1. 在棋牌游戏的例子中，有没有必要把 ChessPiecePosition 设计成享元呢？
2. 在文本编辑器的例子中，调用 CharacterStyleFactory 类的 getStyle() 方法，需要在 styles 数组中遍历查找，而遍历查找比较耗时，是否可以优化一下呢？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

关注极客时间服务号 每日学习签到

月领 25+ 极客币

【点击】保存图片，打开【微信】扫码>>>



© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 53 | 组合模式：如何设计实现支持递归遍历的文件系统目录树结构？

下一篇 55 | 享元模式（下）：剖析享元模式在Java Integer、String中的应用

精选留言 (34)

写留言



Ken张云忠

2020-03-06

1.在棋牌游戏的例子中，有没有必要把 ChessPiecePosition 设计成享元呢？

没有必要,设计成享元模式主要是为了节省内存资源.

ChessPiece中的positionX和positionY共占用8个字节,而把ChessPiecePosition设计成享元模式,ChessPiecePosition的引用在ChessPiece中也是占用8个字节,反而还需要额外的内存空间来存放棋盘中各个位置的对象,最终就得不偿失了....

展开

3

16



Xion

2020-03-06

1. 没有必要，每局游戏的棋子位置不是完全相同的数据，这取决于用户的输入，随着时间的推移会不断地变化。而使用享元模式保存的数据，应当是那些不变的，会被大量对象复用的数据。

2.可以考虑使用哈希表保存文本格式，用多出来的一点点空间占用换取O（1）的查询效率。

展开 ∨



4



知非

2020-03-06

课后思考题:

1. position可以使用享元模式，但是对于位置信息而言，两个short类型的整数可以表示，大量的位置信息也不会占据太多的存储空间，使用享元模式一定程度上增加了代码实现的复杂度，造成move() 方法代码不够直观
2. 重写CharacterStyle 的hashCode()方法，使用map作为对象池，map的key就是hash...

展开 ∨



2



李湘河

2020-03-08

补充一下，对于第二个问题，用LinkedHashMap容器并开启它的LRU策略来装CharacterStyle更好，因为根据一个使用者的习惯，常用的字体风格就是自己最近使用的。



1



Wh1

2020-03-06

在避免创建CharacterStyle对象同时，以O(1)的时间复杂度判断CharacterStyle是否已经被创建，代码如下：

```
public class CharacterStyleFactory {  
    private static final Map<Integer, CharacterStyle> styles = new HashMap<>();  
    ...  
}
```

...

展开 ∨



1



Frank

2020-03-06

打卡 今日学习享元设计模式，收获如下：

当某个需求中有大量的对象是相似的（或者对象中的某些属性是类似的），且是不可变的，此时可以使用享元设计模式将其进行缓存起来以达到共享使用，节省内存。

个人觉得享元模式体现了DRY原则，DRY原则是说不要写重复的代码，应用到对象存储方面，可以理解为不要存储相同的数据。

展开 ∨



1



Jackey

2020-03-06

前面看的时候就在想感觉有点像连接池，当看到一个“共享使用”，一个“重复使用”时真是有种恍然大悟的感觉

展开 ∨



1



rayjun

2020-03-06

棋牌中的位置也可以设置为享元，因为棋盘上位置个数有限，使用享元也可以节省内存



1



Jeff.Smile

2020-03-06

要点总结

1 代码实现主要是通过工厂模式，在工厂类中，通过一个 Map 或者 List 来缓存已经创建好的享元对象，以达到复用的目的。

2 应用单例模式是为了保证对象全局唯一。应用享元模式是为了实现对象复用，节省内存。缓存是为了提高访问效率，而非复用。池化技术中的“复用”理解为“重复使用”...

展开 ∨



1



zx

2020-03-10

对于百万在线的系统，棋盘的位置数量是有限的，可以设计为享元模式



不似旧日

2020-03-09

spring的对象注入是复用对象它是用了享元模式么？感觉文中示例代码中的工厂模式也不是必须的；如果不用工厂模式直接把ChessPiece注入到ChessBoard那么还是享元模式么？



jaryoung

2020-03-09

问题1：没有必要，因为他们不属于大量重复或者相似对象。

问题2：map存储最近几种常用的字体（LRU），如果寻找不到，再进行遍历去找。





2020-03-09

课后题1:没必要,原因是:使用享元模式的目的是节省内存,然而在当前场景下优势并不明显,以下为个人测试所得

使用工具类:版本为4.0.0的lucene-core包中的RamUsageEstimator,使用方法为shallowSizeOf()

测试环境:64位oracle JDK 1.8.0_181 ,默认开启压缩指针(可以在测试类中打断点,然后用j...
展开 ▾



李湘河

2020-03-07

问题1: 没必要, 位置信息一直在变, 不符合使用享元模式的特征;

问题2: 由于常用的字体格式就是那么几种, 这里可以使用LRU队列存储;



岁月

2020-03-07

课堂讨论题

第一题, 我的回答是:没必要, 这是因为一个指针在64位机器上的大小一般是8个字节, 而棋盘的位置信息, 我估计可以用一个短整形来表示, 一个短整形占用字节数大概就是2个字节到4个字节, 所以一个坐标占用的大小都没超过一个指针的内存大小。

这道题总的来说, 要不要用享元模式, 主要是看能节约多少内存, 如果节约不了或者节...
展开 ▾



Jemmy

2020-03-07

最近看Redis源码, 有个这个全局对象, 感觉用的就是享元模式:

```
// server.c
```

```
struct sharedObjectsStruct shared;
```



辣么大

2020-03-07

享元模式: 复用不变对象节省空间。

何时使用: 系统中存在大量重复对象, 考虑使用享元模式。(对象, 相同的字段等等)

如何使用: Java中使用List或者Map缓存创建好的对象。...

展开 ∨



webmin

2020-03-07

1. 在棋牌游戏的例子中，有没有必要把 ChessPiecePosition 设计成享元呢？

没有必要设计为享元，但是如果从节省存储空间的角度考虑，可以进行压缩，棋盘的大小是有限且固定的（围棋19*19应该是最大的棋盘），可以把x,y二维表示法，变为一维表示，这样有压缩效果，具体如下：以围棋为例 x表示第几行，y表示第几列的话 $pos = (x * 19) + y$ ，反推 $x = pos / 19, y = pos \% 19$

展开 ∨



Fstar

2020-03-07

1. 没有必要。棋盘上位置的点集是有限的，是可以设计成享元的。但我们只需要存两个很小的整数，用上享元代码就会变复杂，另外指针也要存储空间。设计成享元可以，但没有必要。

2. 用哈希表提高查询速度：将 font, size, style 连接为字符串（比如 'yahei-12-123456'...

展开 ∨



小刀

2020-03-07

position的位置相对来说，是有限的，也可以设计为享元

