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？

王争 2020-03-06

00:00 11:09
讲述：冯永吉 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 中的应用。

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

```java
public class ChessPiece {//棋子
  private int id;
  private String text;
  private Color color;
  private int positionX;
  private int positionY;

  public ChessPiece(int id, String text, Color color, int positionX, int posit
    this.id = id;
    this.text = text;
    this.color = color;
    this.positionX = positionX;
    this.positionY = positionX;
  }

  public static enum Color {
    RED, BLACK
  }

  // ...省略其他属性和getter/setter方法...
}

public class ChessBoard {//棋局
  private Map<Integer, ChessPiece> chessPieces = new HashMap<>();

  public ChessBoard() {
    init();
  }

  private void init() {
    chessPieces.put(1, new ChessPiece(1, "車", ChessPiece.Color.BLACK, 0, 0));
    chessPieces.put(2, new ChessPiece(2,"馬", ChessPiece.Color.BLACK, 0, 1));
    //...省略摆放其他棋子的代码...
  }

  public void move(int chessPieceId, int toPositionX, int toPositionY) {
    //...省略...
  }
}
```

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：

复制代码

```java
// 享元类
public class ChessPieceUnit {
  private int id;
  private String text;
  private Color color;

  public ChessPieceUnit(int id, String text, Color color) {
    this.id = id;
    this.text = text;
    this.color = color;
  }

  public static enum Color {
    RED, BLACK
  }

  // ...省略其他属性和getter方法...
}

public class ChessPieceUnitFactory {
  private static final Map<Integer, ChessPieceUnit> pieces = new HashMap<>();

  static {
    pieces.put(1, new ChessPieceUnit(1, "車", ChessPieceUnit.Color.BLACK));
    pieces.put(2, new ChessPieceUnit(2,"馬", ChessPieceUnit.Color.BLACK));
    //...省略摆放其他棋子的代码...
  }

  public static ChessPieceUnit getChessPiece(int chessPieceId) {
    return pieces.get(chessPieceId);
  }
}

public class ChessPiece {
  private ChessPieceUnit chessPieceUnit;
  private int positionX;
  private int positionY;

  public ChessPiece(ChessPieceUnit unit, int positionX, int positionY) {
    this.chessPieceUnit = unit;
    this.positionX = positionX;
    this.positionY = positionY;
  }
  // 省略getter、setter方法
}

public class ChessBoard {
  private Map<Integer, ChessPiece> chessPieces = new HashMap<>();

  public ChessBoard() {
    init();
  }

  private void init() {
    chessPieces.put(1, new ChessPiece(
            ChessPieceUnitFactory.getChessPiece(1), 0,0));
    chessPieces.put(1, new ChessPiece(
            ChessPieceUnitFactory.getChessPiece(2), 1,0));
    //...省略摆放其他棋子的代码...
  }

  public void move(int chessPieceId, int toPositionX, int toPositionY) {
    //...省略...
  }
}
```

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：

复制代码

```java
public class Character {//文字
  private char c;

  private Font font;
  private int size;
  private int colorRGB;

  public Character(char c, Font font, int size, int colorRGB) {
    this.c = c;
    this.font = font;
    this.size = size;
    this.colorRGB = colorRGB;
  }
}

public class Editor {
  private List<Character> chars = new ArrayList<>();

  public void appendCharacter(char c, Font font, int size, int colorRGB) {
    Character character = new Character(c, font, size, colorRGB);
    chars.add(character);
  }
}
```

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：

复制代码

```java
public class CharacterStyle {
  private Font font;
  private int size;
  private int colorRGB;

  public CharacterStyle(Font font, int size, int colorRGB) {
    this.font = font;
    this.size = size;
    this.colorRGB = colorRGB;
  }

  @Override
  public boolean equals(Object o) {
    CharacterStyle otherStyle = (CharacterStyle) o;
    return font.equals(otherStyle.font)
            && size == otherStyle.size
            && colorRGB == otherStyle.colorRGB;
  }
}

public class CharacterStyleFactory {
  private static final List<CharacterStyle> styles = new ArrayList<>();

  public static CharacterStyle getStyle(Font font, int size, int colorRGB) {
    CharacterStyle newStyle = new CharacterStyle(font, size, colorRGB);
    for (CharacterStyle style : styles) {
      if (style.equals(newStyle)) {
        return style;
      }
    }
    styles.add(newStyle);
    return newStyle;
  }
}

public class Character {
  private char c;
  private CharacterStyle style;

  public Character(char c, CharacterStyle style) {
    this.c = c;
    this.style = style;
  }
}

public class Editor {
  private List<Character> chars = new ArrayList<>();

  public void appendCharacter(char c, Font font, int size, int colorRGB) {
    Character character = new Character(c, CharacterStyleFactory.getStyle(font
    chars.add(character);
  }
}
```

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。