70-备忘录模式：对于大对象的备份和恢复，如何优化内存和时间的消耗？

上两节课，我们学习了访问者模式。在23种设计模式中，访问者模式的原理和实现可以说是最难理解的了，特别是它的代码实现。其中，用Single Dispatch来模拟Double Dispatch的实现思路尤其不好理解。不知道你有没有将它拿下呢？如果还没有弄得很清楚，那就要多看几遍、多自己动脑经琢磨一下。

今天，我们学习另外一种行为型模式，备忘录模式。这个模式理解、掌握起来不难，代码实现比较灵活，应用场景也比较明确和有限，主要是用来防丢失、撤销、恢复等。所以，相对于上两节课，今天的内容学起来相对会比较轻松些。

话不多说，让我们正式开始今天的学习吧！

备忘录模式的原理与实现

备忘录模式，也叫快照（Snapshot）模式，英文翻译是Memento Design Pattern。在GoF的《设计模式》一书中，备忘录模式是这么定义的：

> Captures and externalizes an object’ s internal state so that it can be restored later, all without violating encapsulation.

翻译成中文就是：在不违背封装原则的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便之后恢复对象为先前的状态。

在我看来，这个模式的定义主要表达了两部分内容。一部分是，存储副本以便后期恢复。这一部分很好理解。另一部分是，要在不违背封装原则的前提下，进行对象的备份和恢复。这部分不太好理解。接下来，我就结合一个例子来解释一下，特别带你搞清楚这两个问题：

- 为什么存储和恢复副本会违背封装原则？
- 备忘录模式是如何做到不违背封装原则的？

假设有这样一道面试题，希望你编写一个小程序，可以接收命令行的输入。用户输入文本时，程序将其追加存储在内存文本中；用户输入“:list”，程序在命令行中输出内存文本的内容；用户输入“:undo”，程序会撤销上一次输入的文本，也就是从内存文本中将上次输入的文本删除掉。

我举了个小例子来解释一下这个需求，如下所示：

```
>hello
>:list
hello
>world
>:list
helloworld
>:undo
>:list
hello
```

怎么来编程实现呢？你可以打开IDE自己先试着编写一下，然后再看我下面的讲解。整体上来讲，这个小程序实现起来并不复杂。我写了一种实现思路，如下所示：

```
public class InputText {
  private StringBuilder text = new StringBuilder();

  public String getText() {
    return text.toString();
  }

  public void append(String input) {
    text.append(input);
  }

  public void setText(String text) {
    this.text.replace(0, this.text.length(), text);
  }
}

public class SnapshotHolder {
  private Stack<InputText> snapshots = new Stack<>();

  public InputText popSnapshot() {
    return snapshots.pop();
  }

  public void pushSnapshot(InputText inputText) {
    InputText deepClonedInputText = new InputText();
    deepClonedInputText.setText(inputText.getText());
    snapshots.push(deepClonedInputText);
  }
}

public class ApplicationMain {
  public static void main(String[] args) {
    InputText inputText = new InputText();
    SnapshotHolder snapshotsHolder = new SnapshotHolder();
    Scanner scanner = new Scanner(System.in);
    while (scanner.hasNext()) {
      String input = scanner.next();
      if (input.equals(":list")) {
        System.out.println(inputText.toString());
      } else if (input.equals(":undo")) {
        InputText snapshot = snapshotsHolder.popSnapshot();
        inputText.setText(snapshot.getText());
      } else {
        snapshotsHolder.pushSnapshot(inputText);
        inputText.append(input);
      }
    }
  }
}
```

实际上，备忘录模式的实现很灵活，也没有很固定的实现方式，在不同的业务需求、不同编程语言下，代码实现可能都不大一样。上面的代码基本上已经实现了最基本的备忘录的功能。但是，如果我们深究一下的话，还有一些问题要解决，那就是前面定义中提到的第二点：要在不违背封装原则的前提下，进行对象的备份和恢复。而上面的代码并不满足这一点，主要体现在下面两方面：

- 第一，为了能用快照恢复InputText对象，我们在InputText类中定义了setText()函数，但这个函数有可能会被其他业务使用，所以，暴露不应该暴露的函数违背了封装原则；
- 第二，快照本身是不可变的，理论上讲，不应该包含任何set()等修改内部状态的函数，但在上面的代码实现中，“快照“这个业务模型复用了InputText类的定义，而InputText类本身有一系列修改内部状态的函数，所以，用InputText类来表示快照违背了封装原则。

针对以上问题，我们对代码做两点修改。其一，定义一个独立的类（Snapshot类）来表示快照，而不是复用InputText类。这个类只暴露get()方法，没有set()等任何修改内部状态的方法。其二，在InputText类中，我们把setText()方法重命名为restoreSnapshot()方法，用意更加明确，只用来恢复对象。

按照这个思路，我们对代码进行重构。重构之后的代码如下所示：

```
public class InputText {
  private StringBuilder text = new StringBuilder();

  public String getText() {
    return text.toString();
  }

  public void append(String input) {
    text.append(input);
  }

  public Snapshot createSnapshot() {
    return new Snapshot(text.toString());
  }

  public void restoreSnapshot(Snapshot snapshot) {
    this.text.replace(0, this.text.length(), snapshot.getText());
  }
}

public class Snapshot {
  private String text;

  public Snapshot(String text) {
    this.text = text;
  }

  public String getText() {
    return this.text;
  }
}

public class SnapshotHolder {
  private Stack<Snapshot> snapshots = new Stack<>();

  public Snapshot popSnapshot() {
    return snapshots.pop();
  }

  public void pushSnapshot(Snapshot snapshot) {
    snapshots.push(snapshot);
  }
}

public class ApplicationMain {
  public static void main(String[] args) {
```

```
    InputText inputText = new InputText();
    SnapshotHolder snapshotsHolder = new SnapshotHolder();
    Scanner scanner = new Scanner(System.in);
    while (scanner.hasNext()) {
      String input = scanner.next();
      if (input.equals(":list")) {
        System.out.println(inputText.toString());
      } else if (input.equals(":undo")) {
        Snapshot snapshot = snapshotsHolder.popSnapshot();
        inputText.restoreSnapshot(snapshot);
      } else {
        snapshotsHolder.pushSnapshot(inputText.createSnapshot());
        inputText.append(input);
      }
    }
  }
}
```

实际上，上面的代码实现就是典型的备忘录模式的代码实现，也是很多书籍（包括GoF的《设计模式》）中给出的实现方法。

除了备忘录模式，还有一个跟它很类似的概念，“备份”，它在我们平时的开发中更常听到。那备忘录模式跟“备份”有什么区别和联系呢？实际上，这两者的应用场景很类似，都应用在防丢失、恢复、撤销等场景中。它们的区别在于，备忘录模式更侧重于代码的设计和实现，备份更侧重架构设计或产品设计。这个不难理解，这里我就不多说了。

如何优化内存和时间消耗？

前面我们只是简单介绍了备忘录模式的原理和经典实现，现在我们再继续深挖一下。如果要备份的对象数据比较大，备份频率又比较高，那快照占用的内存会比较大，备份和恢复的耗时会比较长。这个问题该如何解决呢？

不同的应用场景下有不同的解决方法。比如，我们前面举的那个例子，应用场景是利用备忘录来实现撤销操作，而且仅仅支持顺序撤销，也就是说，每次操作只能撤销上一次的输入，不能跳过上次输入撤销之前的输入。在具有这样特点的应用场景下，为了节省内存，我们不需要在快照中存储完整的文本，只需要记录少许信息，比如在获取快照当下的文本长度，用这个值结合InputText类对象存储的文本来做撤销操作。

我们再举一个例子。假设每当有数据改动，我们都需要生成一个备份，以备之后恢复。如果需要备份的数据很大，这样高频率的备份，不管是对存储（内存或者硬盘）的消耗，还是对时间的消耗，都可能是无法接受的。想要解决这个问题，我们一般会采用“低频率全量备份”和“高频率增量备份”相结合的方法。

全量备份就不用讲了，它跟我们上面的例子类似，就是把所有的数据“拍个快照”保存下来。所谓“增量备份”，指的是记录每次操作或数据变动。

当我们需要恢复到某一时间点的备份的时候，如果这一时间点有做全量备份，我们直接拿来恢复就可以了。如果这一时间点没有对应的全量备份，我们就先找到最近的一次全量备份，然后用它来恢复，之后执行此次全量备份跟这一时间点之间的所有增量备份，也就是对应的操作或者数据变动。这样就能减少全量备份的数量和频率，减少对时间、内存的消耗。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

备忘录模式也叫快照模式，具体来说，就是在不违背封装原则的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便之后恢复对象为先前的状态。这个模式的定义表达了两部分内容：一部分是，存储副本以便后期恢复；另一部分是，要在不违背封装原则的前提下，进行对象的备份和恢复。

备忘录模式的应用场景也比较明确和有限，主要是用来防丢失、撤销、恢复等。它跟平时我们常说的“备份”很相似。两者的主要区别在于，备忘录模式更侧重于代码的设计和实现，备份更侧重架构设计或产品设计。

对于大对象的备份来说，备份占用的存储空间会比较大，备份和恢复的耗时会比较长。针对这个问题，不同的业务场景有不同的处理方式。比如，只备份必要的恢复信息，结合最新的数据来恢复；再比如，全量备份和增量备份相结合，低频全量备份，高频增量备份，两者结合来做恢复。

课堂讨论

今天我们讲到，备份在架构或产品设计中比较常见，比如，重启Chrome可以选择恢复之前打开的页面，你还能想到其他类似的应用场景吗？

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。

精选留言：

- DexterPoker 2020-04-13 08:21:01
  MySQL数据库低频全量备份，结合binlog增量备份，来恢复数据。 [4赞]

- jaryoung 2020-04-13 18:21:27
  游戏存档是不是很典型的例子？想当年，玩《勇者斗恶龙》的时候，打不过boss不断回到存档的地方，不断尝试。 [2赞]

- Jackey 2020-04-13 09:41:12
  想起了Redis主从同步的增量和全量模式 [2赞]

- Frank 2020-04-13 07:54:03
  打卡：今日学习备忘录模式，收获如下：
  备忘录模式是在不违背封装原则的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便之后恢复对象为先前的状态。个人觉得理解起来就是解决对象状态一致性问题，主要是在代码设计上不要违背封装原则，如果能打破封装原则，那么就有可能对象的状态不一致了，后面恢复后对象状态就不一致了。备忘录模式与备份相似，前者注重代码设计，后者注重架构和产品设计。在大对象备份过程中，需要考虑存储以及恢复的时间，可以使用一定的策略，如只备份恢复必要的信息，如全量备份和增量备份相结合。
  对于课堂讨论还有其他场景：数据库备份与恢复，Git版本管理，虚拟机生成快照与恢复等。 [1赞]

- Demon.Lee 2020-04-13 23:37:30
  System.out.println(inputText.toString()); ---> System.out.println(inputText.getText());

- Geek_54edc1 2020-04-13 17:43:29
  浏览器的后退、前进功能，也可以看作是一种“备份”

- hanazawakana 2020-04-13 13:20:31
  MySQL也会定期做全备份，然后还有binlog redolog这样的增量备份

- Heaven 2020-04-13 12:20:22
  1.从开发上将,在使用定时任务Quartz的时候,会进行对应的备份,方便我们在项目重启后从数据库中反序列化回来,利用了一个外部工具来进行了备份
  2.在整体架构中,MySQL就是使用全量备份和增量备份相结合的方式进行了备份,我们自己的项目也是一星期一次全量,配合binlog回滚
  3.在生活中,我记得XBox上的极限竞速游戏提供了回滚功能,就是使用的备份来方便撞车后直接回溯操作

- 守拙 2020-04-13 10:10:10
  备份的应用场景:
  1. 游戏有自动备份功能, 掉线/下线再次上线时, 可以继续之前的游戏;
  2. Android提供了Activity状态的备份/恢复, 当App处于后台, Activity被系统回收前会调用onSavedInstanceState()备份, App切回前台Activity会重建, 可以通过onRestoreInstanceState()恢复之前的状态.

- 三木子 2020-04-13 09:37:50
  每次更新服务打补丁都要备份一个当前版本了。

- 朱晋君 2020-04-13 09:36:55
  数据库的mvcc机制，现在一些分布式事务的实现方式也用到了这种思想

- Ken张云忠 2020-04-13 08:14:56
  mysql和redis的快照功能

- Ken张云忠 2020-04-13 08:13:05
  mysql数据库基于时间点的数据恢复也该是备忘录模式实现的

- Modern 2020-04-13 08:08:13
  将软件配置导出，转给他人，他人用此文件初始化软件，比如快捷键外观字体等的设置
  打开软件可以任意选择一个曾经工作的项目，可以打开之前的编辑状态

- 小晏子 2020-04-13 08:04:40
  想到的是redis中数据的备份和恢复，通过全量和增量备份结合来做恢复。

- 何用 2020-04-13 08:02:02
  老师用词太过主观了，灌输访问者模式难理解的潜意识。我倒觉得 Single Dispatch 和 Double Dispatch 容易理解，反倒是今天的备忘模式难理解了。。。

- 马以 2020-04-13 07:51:48
  高频增量也是数据库事务隔离用的方法

- eason2017 2020-04-13 07:25:30
  电脑待机后启动恢复到待机前最后一个状态。

- 张三 2020-04-13 01:46:36
  IDE每次退出重新打开都有之前打开的代码窗口，git呢?