COA2021-programming08

Good luck and have fun!

1 实验要求

1.1 fetch

在Cache类中，实现**基于通用映射策略的**fetch方法，查询数据块在cache中的行号。如果目标数据块不在cache内，则将数据从内存读到Cache，并返回被更新的cache行的行号

```
private int fetch(String pAddr)
```

1.2 map

在Cache类中，实现**基于通用映射策略的**map方法，根据数据在内存中的块号进行映射，返回cache中所对应的行，-1表示未命中

```
private int map(int blockNO)
```

1.3 strategy

在cacheReplacementStrategy包中，分别实现**先进先出**替换策略、**最不经常使用**替换策略、**最近最少使用**替换策略。

1.4 write

在Cache类中，**基于写直达和写回两种策略**完善write方法

```
public void write(String pAddr, int len, char[] data)
```

2 实验攻略

2.1 实验概述

本次作业和上次作业组合为一个通用的cache系统。在这个系统中，不仅需要大家实现基本的读写功能，还需要大家实现各种各样的策略，具体有：

- 映射策略：直接映射、关联映射、组关联映射
- 替换策略：最近最少使用算法 (LRU)、先进先出算法 (FIFO)、最不经常使用算法 (LFU)
- 写策略：写直达(write through)、写回法(write back)

相信在上次作业中，大家已经对框架代码有了一个初步的理解，打下了良好的基础，那么下一步工作的开始将没有想象中的那么困难。接下来就是需要大家去动手实现完整功能的时候了！

请认真阅读PPT与课本，任何与具体实现相关的知识都已经体现在PPT之中了。

2.2 代码导读

2.2.1 代码结构

```
.
│  pom.xml      # Maven Config File
│  README.md    # This File
│
├─.idea
│
└─src
    ├─main
    │  └─java
    │      ├─cpu
    │      │
    │      ├─memory
    │      │  │  Memory.java     # 主存类，需要阅读，不需要修改
    │      │  │
    │      │  └─cache
    │      │      │  Cache.java  # Cache类，需要阅读且修改
    │      │      │
    │      │      └─cacheReplacementStrategy
    │      │              FIFOReplacement.java   # 先进先出替换策略类，需要修改
    │      │              LFUReplacement.java    # 最不经常使用替换策略类，需要修改
    │      │              LRUReplacement.java    # 最近最少使用替换策略类，需要修改
    │      │              ReplacementStrategy.java       # 替换策略接口，需要阅读，不需要修改
    │      │
    │      └─util
    │
    └─test
        └─java
            └─memory
                └─cache
                        AssociativeMappingTest.java       # 关联映射策略测试
                        DirectMappingTest.java    # 直接映射策略测试
                        SetAssociativeMappingTest.java    # 组关联映射策略测试
                        WriteBackTest.java        # 写回策略测试
                        WriteThroughTest.java     # 写直达策略测试
```

2.2.2 版本更新说明

在上次作业中，我们已经对Cache的存储结构模拟、映射策略模拟、数据读取模拟进行了讲解，在此不再赘述。接下来回答一下上次作业中大家的一些疑问。

Q：在上次作业中，read函数中貌似没有实现分行读取的功能?

A：确实，但是上次作业的测试用例并不涉及跨行的数据，所以在上次作业中没有发现这个bug。本次版本更新修复了这个小bug，修改了fetch方法的参数，大家可以自行查看并理解其中含义。当然这个bug只会影响本次作业，上次作业中不修改也可以顺利ac~

Q：内存大小是32MB，那对于直接映射，12位tag是不是用不满?

A：是的，32位地址应该对应4GB内存，但是由于seecoder平台上无法开辟这么大的内存，所以我们只能使用32MB内存，但为了通用系统的设计，我们还是采用了32位地址。

接下来我们来讲讲这次作业的新增内容。

2.2.3 替换策略模拟

在上次作业中，我们只需要大家实现直接映射策略。对于直接映射策略，每个数据块都只有唯一对应的行可以放置，因此不需要替换策略。而对于关联映射和组关联映射，在cache进行映射时，一旦cache行被占用，当新

的数据块装入cache中时，原先存放的数据块将会被替换掉，这时候就需要替换策略来决定替换哪一行了。

本次作业中，我们用策略模式来模拟替换策略，Cache类中相关代码如下

```
private ReplacementStrategy replacementStrategy;    // 替换策略

public void setReplacementStrategy(ReplacementStrategy replacementStrategy) {
        this.replacementStrategy = replacementStrategy;
    }
```

在ReplacementStrategy接口中，我们定义了两个抽象方法，方法的具体功能已在注释中标出，方法签名如下

```
/**
 * 结合具体的替换策略，进行命中后进行相关操作
 * @param rowNO 行号
 */
void hit(int rowNO);

/**
 * 结合具体的映射策略，在给定范围内对cache中的数据进行替换
 * @param start 起始行
 * @param end 结束行 闭区间
 * @param addrTag tag
 * @param input  数据
 */
int replace(int start, int end, char[] addrTag, char[] input);
```

然后，我们提供了实现了ReplacementStrategy接口的三个子类：FIFOReplacement类、LFUReplacement类和LRUReplacement类，这三个实体策略类需要根据具体的策略来实现不同的hit和replace方法。

根据策略模式的思想，Cache类只需要持有一个ReplacementStrategy接口的字段replacementStrategy，通过setReplacementStrategy方法设置replacementStrategy所引用的实体策略类。因此，Cache类并不需要分别持有三个实体策略类的引用，它也不需要知道replacementStrategy字段中存放的具体是哪个子类，它只需要在合适的地方调用ReplacementStrategy接口的hit方法和replace方法。依据Java的多态特性，在Cache类调用这两个方法时，会自动根据replacementStrategy字段中存放的实体策略类去选择具体的实现。

这样，通过策略模式，我们即可完成替换策略的模拟~

2.2.4 写策略模拟

本次作业中新增了cache的写功能，提供了write方法。在write方法中，我们已经实现好了向cache中写数据的功能，并且也处理好了数据跨行的情况。方法实现如下

```
/**
 * 向cache中写入[pAddr, pAddr + len)范围内的连续数据，可能包含多个数据块的内容
 *
 * @param pAddr 数据起始点(32位物理地址 = 22位块号 + 10位块内地址)
 * @param len   待写数据的字节数
 * @param data  待写数据
 */
public void write(String pAddr, int len, char[] data) {
    int addr = Integer.parseInt(transformer.binaryToInt("0" + pAddr));
    int upperBound = addr + len;
    int index = 0;
    while (addr < upperBound) {
        int nextSegLen = LINE_SIZE_B - (addr % LINE_SIZE_B);
        if (addr + nextSegLen >= upperBound) {
```

```
                nextSegLen = upperBound - addr;
            }
            int rowNO = fetch(transformer.intToBinary(String.valueOf(addr)));
            char[] cache_data = cache.get(rowNO).getData();
            int i = 0;
            while (i < nextSegLen) {
                cache_data[addr % LINE_SIZE_B + i] = data[index];
                index++;
                i++;
            }
                    // TODO
            addr += nextSegLen;
        }
    }
```

如果只是写数据到cache，那么本次作业就非常的简单。但是考虑了主存的内容后，一切都变的复杂了起来。由于Cache和Memory的数据之间存在一定的一致性，会不会出现Cache被修改了，主存也被修改了，但是两次修改的数据不一样呢？这个时候该以哪一份数据为准呢？这时候就需要写策略了。

由于写策略只分为写直达和写回两种策略，在本次作业中，我们简单地通过一个布尔类型的字段来模拟，如下所示

```
public static boolean isWriteBack;   // 写策略
```

- 当isWriteBack为true时，表示此时Cache使用写回策略。
- 当isWriteBack为false时，表示此时Cache使用写直达策略。

这样，我们就简单地完成了写策略的模拟。

在你已经结合源代码充分理解上述内容后，接下来就可以开始快乐地编码啦！

2.3 实现指导

2.3.1 通用映射策略的实现

第一步，你需要完善自己的fetch和map方法，以实现对不同SET和SetSize通用的映射策略。

上次作业中，我们只要求大家实现直接映射策略，测试用例中也只会设置SETS为1和SetSize为1024。而本次作业中，我们要求实现通用的映射策略，也就是说，我们会在测试用例中设置不同的SET和SetSize。

上次作业中，大家已经针对直接映射实现了fetch和map方法，可以直接复制粘贴过来。但是，在上次作业的map方法中，你可能会利用如下公式来计算cache行号。

```
int rowNO = blockNO % 1024;
```

这个公式仅对于直接映射生效，对于其他映射策略则无法生效。因此，你可能需要修改自己的map方法中计算行号的方法，编写一个对于各种不同的SET和SetSize均能正确映射的算法，来实现通用的映射策略。

同时，由于SET和SetSize的改变，tag的位数也会发生变化。因此，涉及到计算tag的地方（可能不止一处）都可能需要修改，最好的处理方式是抽象出一个calculateTag方法。关于tag位数的说明可以参考上次作业的手册2.2.3部分。

如果你在上次作业已经实现好了通用的fetch和map方法，那么这次作业就不需要修改啦~

2.3.2 替换策略的实现

第二步，你需要实现好三个实体策略类：FIFOReplacement类、LFUReplacement类和LRUReplacement类，并在Cache类中合适的地方对hit和replace方法进行调用。

我们给出了三个策略的参考实现方式如下，大家可以参考，也可以自行设计算法来实现。

- 对于FIFO策略，你可能需要用到CacheLine中的timeStamp字段，记录每一行进入Cache的时间。
- 对于LFU策略，你可能需要用到CacheLine中的visited字段，记录每一行被使用的次数。
- 对于LRU策略，你可能需要用到CacheLine中的timeStamp字段，记录每一行最后被访问时间。

hit方法要做的事情就是在该行命中后，进行具体替换策略相关的操作。replace方法要干的事情也很简单，就是在给定范围内，根据具体的策略，寻找出需要被替换的那一行进行替换并返回行号。至于hit方法和replace具体需要摆放在什么位置，就留给聪明的你自己思考啦。

注意，在这一步中，你会发现在三个实体策略类中无法访问熟悉的CacheLinePool对象cache。这是因为它被设置为了Cache类的私有字段，同时CacheLinePool类也是cache的私有内部类。这时候，你可能想把private统统改成public来解决问题，但这并不符合面向对象的数据封装思想。更好的解决方法应该是在Cache类内编写公有的getter/setter方法。你可能需要用到的getter/setter方法签名如下，大家可以自行使用并实现之。

```
// 获取有效位
public boolean isValid(int rowNO)

// 获取脏位
public boolean isDirty(int rowNO)

// 增加访问次数
public void addVisited(int rowNO)

// 获取访问次数
public int getVisited(int rowNO)

// 重置时间戳
public void setTimeStamp(int rowNO)

// 获取时间戳
public long getTimeStamp(int rowNO)

// 获取该行数据
public char[] getData(int rowNO)
```

还记得上次作业中我们留下的update方法吗？如果你在上次作业里成功抽象出了这个方法，那么在三个实体策略类中就可以直接调用这个方法了哦。

2.3.3 写策略的实现

在顺利完成上面两步之后，你就可以进入最后一步——写策略啦。可以发现，涉及到cache往主存里写数据的只有两个地方：

- write方法直接向cache写数据时
- replace方法需要替换掉一行数据时

写策略其实是代码量最小的部分，你只需要在write方法和replace方法中合适的地方添加代码，对isWriteBack字段判断，然后根据具体策略来做不同的事情。

注意，在这一步中，当你需要往主存写数据的时候，你可能会发现你手上只有一个cache行号，并不知道需要写入的主存物理地址。这时候，解决方法可以是在Cache类中编写一个calculatePAddr方法，结合具体映射策略，根据行号计算该行首地址对应的物理地址。

需要注意的是，对于写回策略，脏位和有效位之间是存在一定的约束的，大家可以在测试时体会其中的奥妙。

2.4 总结

所有需要大家完成的部分都已经用TODO标出。为了减轻大家的负担，我们归纳了本次作业中你需要完成的小任务以及步骤

1. 正确实现好通用的映射策略，然后你应该可以通过DirectMappingTest（与上次作业相同）的4个用例，以及AssociativeMappingTest和SetAssociativeMappingTest的两个test01。
2. 正确实现好FIFO替换策略，然后你应该可以通过AssociativeMappingTest和SetAssociativeMappingTest的两个test02。
3. 正确实现好LFU替换策略，然后你应该可以通过AssociativeMappingTest和SetAssociativeMappingTest的两个test03。
4. 正确实现好LRU替换策略，然后你应该可以通过AssociativeMappingTest和SetAssociativeMappingTest的两个test04。
5. 正确实现好写直达策略，然后你应该可以通过WriteThroughTest的4个用例。
6. 正确实现好写回策略，然后你应该可以通过WriteBackTest的4个用例。

至此，你已经完成了全部工作(・ω・)ノ

2.5 测试相关

如果你在测试的时候，发现如果运行整个测试文件就会导致test01到test04全都报错，但是如果单独运行test02却可以通过的时候，不要惊慌，不是见鬼了。这是由于每个测试用例的最后一行都会运行cache.clear()函数，该函数的作用是清空cache的全部缓存。如果你的test01未能通过，那么test01将会提早assert，这会导致test01无法运行到cache.clear()那一行。那么，当test01处理结束，继续运行到test02的时候，cache内将会带有上一次保存的数据，这就是导致你的test02也无法通过的原因。

因此，请确保你的每个测试用例都能单独运行通过，再去运行整个测试文件，否则运行结果可能会对你撒谎哦。

3 相关资料

3.1 设计模式——策略模式

3.1.1 背景

在软件开发中我们常常会遇到这种情况，实现某一个功能有多种策略，我们需要根据环境或者条件的不同选择不同的策略来完成该功能。

一种常用的方法是硬编码(Hard Coding)在一个类中。举个例子，比如你需要提供多种排序算法，可以将这些算法都写到一个类中，在该类中提供多个方法，每一个方法对应一个具体的排序算法。当然，我们也可以将这些排序算法封装在一个统一的方法中，通过if-else或者switch-case等条件判断语句来进行选择。这两种实现方法我们都称之为硬编码。

上述硬编码的做法有一个严重的缺点：在这个算法类中封装了多个排序算法，该类代码将较复杂，维护较为困难。如果需要增加一种新的排序算法，或者更换排序算法，都需要修改封装算法类的源代码。

3.1.2 简介

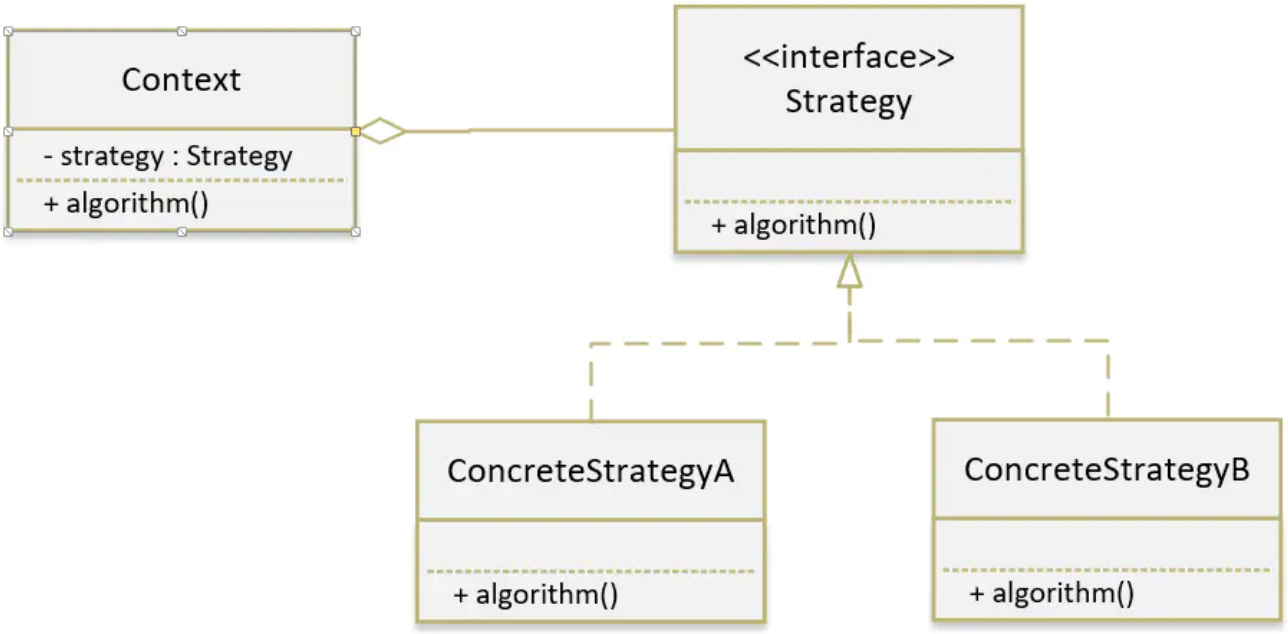
为了解决上述问题，策略模式就应运而生了。

- 意图：策略模式通过定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。

- 机制：策略模式使用的是面向对象的继承和多态机制，从而实现同一行为在不同场景下具备不同实现。
- 主要解决：策略模式主要解决了在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护。
- 优点：1、算法可以自由切换。 2、避免使用多重条件判断。3、扩展性良好。

3.1.3 角色

策略模式的通用类图如下。



从类图中我们可以看到，策略模式主要包含三种角色：

- 上下文角色（Context）：用来操作策略的上下文环境，屏蔽高层模块（客户端）对策略的直接访问，封装可能存在的变化。
- 抽象策略角色（Strategy）：规定策略或算法的行为。
- 具体策略角色（ConcreteStrategy）：具体的策略或算法实现。

3.1.4 实现

策略模式的通用实现方式如下。

```
class Client {

    //抽象策略类 Strategy
    interface IStrategy {
        void algorithm();
    }

    //具体策略类 ConcreteStrategy
    static class ConcreteStrategyA implements IStrategy {
        @Override
        public void algorithm() {
            System.out.println("Strategy A");
        }
    }

    //具体策略类 ConcreteStrategy
    static class ConcreteStrategyB implements IStrategy {
        @Override
        public void algorithm() {
            System.out.println("Strategy B");
```

```
        }
    }

    //上下文环境
    static class Context {
        private IStrategy mStrategy;

        public Context(IStrategy strategy) {
            this.mStrategy = strategy;
        }

        public void algorithm() {
            this.mStrategy.algorithm();
        }
    }

    public static void main(String[] args) {
        //选择一个具体策略
        IStrategy strategy = new ConcreteStrategyA();
        //创建一个上下文环境
        Context context = new Context(strategy);
        //客户端直接让上下文环境执行算法
        context.algorithm();
    }
}
```

3.2 彩蛋

在力扣上，我们不难找到跟这次作业内容相关的题目，不过力扣上的题目对算法的要求更高，本次作业简化了这部分内容，感兴趣的同学可以更深入地了解一下LRU和LFU算法的实现。

题目指路：

https://leetcode-cn.com/problems/lru-cache/

https://leetcode-cn.com/problems/lfu-cache/