COA2023-programming05

> 融会贯通

0 前情提要

在前面四次作业中，同学们已经实现了Transformer、ALU、FPU等模块的对应功能，在本次作业中，同学们将会对Cache的一些功能进行模拟

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, byte[] data)
```

2 实验攻略

2.1 实验概述

我们将在本次作业中实现一个模拟的Cache系统，在这个系统中，大家不仅需要实现基本的读写功能，还需要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)

请认真阅读PPT与课本，救赎之道，就在其中。

接下来，我们将对Cache类的源码进行一个全面的解读。

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 内存结构模拟

我们模拟了一个32KB大小的Cache，规定每一行大小为64B，则cache一共含有512行，如下所示。

```
public static final int CACHE_SIZE_B = 32 * 1024; // 32 KB 总大小

public static final int LINE_SIZE_B = 64; // 64 B 行大小
```

如何模拟Cache的内存结构呢？我们在Cache类中定义了一个私有内部类：CacheLine。CacheLine表示Cache中的一行。这个内部类的数据结构比较简单，大家可以自行阅读源码，在此不再赘述。

2.2.3 映射方法模拟

在Cache类中，我们定义了两个变量，表示组数和每组行数，并且提供了相应的setter方法，如下所示

```
private int SETS;   // 组数
private int setSize;    // 每组行数
public void setSETS(int SETS) {
    this.SETS = SETS;
}
public void setSetSize(int setSize) {
    this.setSize = setSize;
}
```

由于直接映射和关联映射可以视为特殊的组关联映射，因此，**通过设置不同的SETS和setSize**，即可实现不同映射方法模拟。

同时，不同的映射方法会导致Cache行的tag位数不同。以本次实验为例，若采用直接映射，则tag的位数应该为17位，而如果采用关联映射，则tag位数应该为26位（怎么算出来？）。为了实现通用的Cache系统，我们在CacheLine类里面统一使用26位的char数组来模拟tag，tag数组中以低位为准，高位补0，有效长度取决于具体的映射方法。以直接映射方法为例，当tag为1时，数组中后17位应该为“00000000000000001”，前9位补0即可。

2.2.4 数据读取模拟

在成功模拟Cache的内存结构和映射策略后，接下来就是要模拟Cache的数据读取功能。我们提供了一个read方法。请同学们编码之前仔细阅读这段代码，这会帮助你理解整个Cache运行的流程，也会帮助你进行编码。

```
/**
* 读取[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 = 26位块号 + 6位块内地址)
* @param len   待读数据的字节数
* @return 读取出的数据，以char数组的形式返回
```

```
*/
public byte[] read(String pAddr, int len) {
    byte[] data = new byte[len];
    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)));
        byte[] cache_data = cache[rowNO].getData();
        int i = 0;
        while (i < nextSegLen) {
            data[index] = cache_data[addr % LINE_SIZE_B + i];
            index++;
            i++;
        }
        addr += nextSegLen;
    }
    return data;
}
```

由于在Cache中读数据可能会有跨行的问题，即要读取的数据在内存中跨过了数据块的边界，我们在read方法内已经帮大家处理好了这种情况，大家不需要再考虑数据跨行的问题。

2.2.5 替换策略模拟

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。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，没有选择的机会，因此不需要替换策略。而对于关联映射和组关联映射，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，我们定义了两个方法，方法的具体功能已在注释中标出，方法签名如下

```
void hit(int rowNO);

int replace(int start, int end, char[] addrTag, byte[] input);
```

并且，我们提供了实现ReplacementStrategy接口的三个子类：FIFOReplacement类、LFUReplacement类和LRUReplacement类。

根据策略模式的思想，Cache类不需要知道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接口的这两个方法。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字段中存放的具体策略去选择具体的实现方法。这样，通过策略模式，我们即可完成替换策略的模拟~

2.2.6 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，我们已经帮你实现好了向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 = 26位块号 + 6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, byte[] 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)));
        byte[] cache_data = cache[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 故事的开始——通用映射策略的实现

在你充分阅读源代码后，你会发现read方法中需要调用fetch方法，故事就从这个fetch方法开始。请大家从fetch方法开始着手，结合Cache的工作流程，一步步实现好Cache基本的数据读取功能。

在fetch方法中，你需要计算出数据所在的块号，然后调用map方法查询数据块在Cache中的行号。如果目标数据块不在Cache内，则需要将数据从内存读到Cache（需要阅读Memory类源码），并返回被更新的Cache行的行号。

在map方法中，你需要计算出当前块号所对应的Cache行号，然后将当前块号与该Cache行的信息进行比较（需要比较什么信息？），来判断是否命中。命中则返回行号，未命中则返回-1。

在本次作业中，请确保fetch和map方法依照注释正确实现，因为框架代码会对这两个方法进行直接的调用。fetch和map方法均已用TODO标出。

2.3.2 替换策略的实现

第二步，你需要实现好三个策略类：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。注意：

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

至于三个策略的hit方法具体需要干什么事情，这就留给聪明的你自己思考啦。而replace函数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并进行替换并返回行号。

注意，在这一步中，你可能会发现在三个Replacement类中无法访问CacheLinePool，因为他被设置为了Cache类的私有字段。这时候，你可能会想把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编写get/set函数，你可能需要用到的get/set函数方法签名如下，大家可以自行使用并实现

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

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

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

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

```
// 用于LRU算法，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)
```

注意，我们在框架代码中留下了一个update方法。**如果你懂得如何抽象出这个方法，那么在三个Replacement类中就可以直接调用了。**

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。可以发现，涉及到Cache往主存写数据的只有两个地方：

- write函数直接向Cache写数据时
- replace函数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函数和replace函数的相应地方对isWriteBack字段判断，然后根据具体策略来做不同的事情。具体来说，写直达策略就是在write函数完成写Cache后直接修改主存；写回策略就是在write函数完成写Cache后**设置好脏位**，并在replace函数将要替换掉该行的时候将该行写回内存。

提示一下，在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行号，并不知道内存的物理地址，这时候就**需要在Cache类中编写一个根据行号计算物理地址的方法**，具体怎么实现留给聪明的你啦。

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**，大家可以在测试时发现其中的奥妙。

2.3.4 其他方法说明

在Cache类中，我们还友好地提供了计算块号的方法，大家可以自行调用，方法签名如下

```
private int getBlockNO(String pAddr)
```

此外，我们还提供了几个仅用于测试的方法，在这些方法上的注释都已经进行了相关的说明。请大家不要修改这些方法，否则会影响到测试。

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. 正确实现好写回策略，然后你应该可以通过WriteBackTest的4个测试用例。
6. 正确实现好写直达策略，然后你应该可以通过WriteThroughTest的4个测试用例。

至此，你已经完成了全部工作(・ω・)ノ

为了降低实验难度，本次实验我们提供了全部的测试用例用例给大家进行调试。在测试中，我们不仅会检查访问特定行是否读出了正确的数据，还会访问特定的行检查它的Tag以及有效位。具体细节都已经在测试用例中给出，大家可以自行阅读。

3 相关资料

> 下面将会是你们在软工二中会接触到的内容，是软件工程领域非常常用的思想和方法。

3.1 设计模式——单例模式

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。单例模式的要点如下：

1. 单例类只能有一个实例。
2. 单例类必须自己创建自己的唯一实例。
3. 单例类必须给所有其他对象提供这一实例。

为什么要使用单例模式？因为我们使用了OO编程语言，但是在真实的计算机系统当中，很多部件都是唯一的。所以我们需要通过单例模式来保证主存、CPU、Cache等的唯一性，在本次作业中体现为保证Memory类和Cache类分别只有一个实例，并分别提供一个访问它们的全局访问点。

如何实现单例模式？我们只需要将类的构造方法私有化，再添加一个该类类型的私有静态字段，然后提供一个该类的get方法。对于使用该类的使用者们来说，他们只能通过get方法得到该类的实例，那么它们看到的永远是相同的对象。实现代码如下

```
public A{
private A() {}
private static A a = new A();
public static A getA() { return a; }
        }
```

本次作业中，Cache类和Memory类都采用了单例模式，大家可以自行阅读相关代码。

3.2 设计模式——策略模式

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，比如你在购物结算的时候需要选择使用微信支付、支付宝支付或其他支付方式，但他们可以抽象称为同一种功能，这时候我们就需要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，比如排序，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。

为了解决这个问题，策略模式就应运而生了。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。它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。

策略模式的具体实现可以参考：https://www.runoob.com/design-pattern/strategy-pattern.html