# COA2021-programming07

## 1 实验要求

在Cache类中，使用直接映射的方法，实现cache基本的读取数据功能。

### 1.1 fetch

在Cache类中，实现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)
```

## 2 实验攻略

### 2.1 实验概述

本次作业为下次作业的预热作业，两次作业合起来为一个通用的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：

- 映射方法：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)

为了降低实验难度，我们将这个系统分成两次作业。本次作业主要的任务是**理解框架代码**，并使用最简单的**直接映射方法实现数据读取功能**。其他映射策略、替换策略以及写的功能我们留到下次作业。

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。

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

```
.
├── README.md    # This File
├── pom.xml      # Maven Config File
└── src
    └── main
        └── java
            ├── cpu
            │
            ├── memory
```

```
│   │  Memory.java  # 主存，需要阅读，不需要修改
│   │
│   └── cache
│           │       Cache.java      # Cache代码，需要阅读且修改
│           │
│           ├── cacheReplacementStrategy
│           │               ReplacementStrategy.java        # 替换策略，不需要阅读和修改
│           │
│           └── cacheWriteStrategy
│                           WriteStrategy.java      # 写策略，不需要阅读和修改
│
└── util
```

#### 2.2.2 内存结构模拟

我们模拟了一个1MB大小的cache，规定每一行大小为1KB，则cache一共含有1024行，如下所示。

```
public static final int CACHE_SIZE_B = 1024 * 1024; // 1 MB 总大小

public static final int LINE_SIZE_B = 1024; // 1 KB 行大小
```

如何模拟cache的内存结构呢？我们在Cache类中定义了两个私有内部类：CacheLine和CacheLinePool。CacheLine表示Cache中的一行，CacheLinePool就是行的集合。这两个内部类的数据结构都比较简单，大家可以自行阅读源码，在此不再赘述。

对于CacheLinePool类，重点是它持有了一个CacheLine数组，即cache行的集合。因此Cache类只需要持有一个CacheLinePool类的对象，即可模拟cache的内存。在Cache类中初始化如下

```
private final CacheLinePool cache = new CacheLinePool(CACHE_SIZE_B / LINE_SIZE_B);
```

（以下这段可能比较难懂，如果读不懂可以跳过，并不影响代码实现）

那么问题来了，为什么我们需要一个CacheLinePool类，而不是让Cache类直接持有一个CacheLine数组呢？答案藏在CacheLinePool类的get方法中。如果让Cache类直接持有一个CacheLine数组，那么该数组在初始化时，数组内部实际上全都是空指针，在直接对数组进行如clPool[0]的访问操作时，由于没有经过严格的检查，非常有可能出现空指针异常。而如果为了避免空指针异常，在CacheLine数组创建时就让数组的每一个元素分别指向一个CacheLine对象，这又是对内存资源的极大浪费。因此，通过使用CacheLinePool类来管理CacheLine数组的方法，做到了防止空指针与节省内存资源的完美权衡。

#### 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12位，而如果采用关联映射，则tag位数应该为22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22位的char数组来模拟tag，tag数组中以高位为准，低位补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，数组中前12位应该为“000000000001”，后10位补0至22位即可。

在本次作业中，由于只需要大家实现直接映射方法，所以在测试代码中我们只会设置SETS为1024，SetSize为1。但为了代码的可扩展性~~（为了下次作业着想）~~，大家可以在本次作业中尽量实现对各种映射方法通用的数据读取功能。

#### 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 = 22位块号 + 10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 char[] read(String pAddr, int len) {
    char[] data = new char[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(pAddr);
        char[] cache_data = cache.get(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。

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

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 关于TODO

大家也许会注意到，我们还有一个方法使用了TODO标出，方法签名如下

```
public void update(int rowNO, char[] tag, char[] input)
```

这个方法的功能已经在注释里详细标出。注意，这个方法完成与否，对完成本次作业没有任何影响。大家如果觉得使用这个方法签名不方便，完全可以自己另起炉灶实现相关功能，最终只需要确保实现好fetch以及map方法即可。

那为什么还需要用TODO标注出这个方法呢，因为我们建议在这一步抽象出这个功能接口，可以方便下一次作业哦~

#### 2.3.3 其他方法

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 测试相关

为了降低实验难度，本次实验我们提供了一半的测试用例用例给大家进行调试，而隐藏测试用例检查的各种情况跟本地测试用例的各种情况是一致的。也就是说，只要你不是面向用例编程，在通过本地测试用例后，隐藏测试用例就可以全部通过~

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。

### 2.5 下回预告😼

也许有同学已经注意到，在Cache类中，还有一些我们尚未讲解的字段，如下所示

```
private ReplacementStrategy replacementStrategy;    // 替换策略
private WriteStrategy writeStrategy;    // 写策略

public void setStrategy(ReplacementStrategy replacementStrategy, WriteStrategy writeStrategy) {
    this.replacementStrategy = replacementStrategy;
    this.replacementStrategy.setWriteStrategy(writeStrategy);
    this.writeStrategy = writeStrategy;
}
```

本次作业中，大家不需要用到这些字段。同时，我们也给出了两个抽象类：ReplacementStrategy类和WriteStrategy类，这两个类的源码也不需要大家阅读。但是相信大家都已经猜到了，这些都是为下一次作业做准备。是的，在下一次作业中，我们将使用策略模式来实现各种替换策略和写策略。感兴趣的同学可以预先了解策略模式的实现机制。策略模式的讲解我们也将放到下次作业的手册中。

# 3 相关资料

## 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。

Good luck and have fun!