

Storage and Buffer Manager

SA23011086 熊鹏

1. 实验目的

实现一个缓冲区管理，由于框架已经给定，只需要把相应的操作函数进行补全即可

本实验采用 go 语言进行实现

2. 实现细节

2.1 buffer manager

buffer 由多个 frame 组成，frame 中可以装载 page，此实验中将 frame 的大小和 page 的大小为相同的值。


默认的 frame 大小为 4096 个 byte，默认的 buffer 中有 1024 个 frame。

buffer manager 的类结构如下图所示：

```
1 type BMgr struct {
2     freeFramesNum int // the num of free frame
3     ftop          [DEFBUFSIZE]int // frameID to pageID map
4     ptof          [DEFBUFSIZE]*BCB // use pageID % DEFBUFSIZE to calculate the index, if two pageID get the same index, use the linklist
5     cache         Cache // cache
6     dsmgr         DSMgr // data storage manager
7     Buf           [DEFBUFSIZE]BFrame // buffer
8     hitCount      int
9     readDiskIO    uint64
10    writeDiskIO   uint64
11 }
```

- freeFrameNum: 目前 buffer 中空闲的 frame 个数
- ftop: frameID 到 pageID 的映射数组
- ptof: 目前 pageID 到 BCB 的映射，如果存在哈希冲突，将冲突块加入链表尾部
- cache: Cache 接口 (因为需要测定不同 cache 策略的性能，因此构建了 Cache 接口，可以见 [Cache 接口](#) 该接口的讲解)
- dsmgr: 数据存储管理器
- Buf: buffer 中的 frame 数组
- hitCount: 统计 buffer 命中的次数
- readDiskIO: 读操作累计的 read byte 数
- writeDiskIO: 写操作累计的 write byte 数

2.1.1 frame



```
1  const FRAMESIZE = 4096
2  const DEFBUFSIZE = 1024
3
4  type BFrame struct {
5      Filed [FRAMESIZE]byte
6  }
7
```


- FRAMESIZE: 默认的 frame 大小, 4096
- DEFBUFSIZE: 默认的 buffer 大小, 1024
- BFrame: frame 类

2.1.2 Cache 接口

为了提高通用性, 采用了接口 (interface), 只需要构造了 Cache 接口中的函数就可作为参数传入 buffer manager 类中, 包含了 Cache 的一些基础操作: 插入、查询、替换、移除操作。

方便对不同的替换策略性能进行测试

Cache 接口如下所示



```
1  type Cache interface {
2      Insert(frameID int)
3      GetVictim() (frameID int)
4      Query(frameID int)
5      RemoveEle(frameID int)
6  }
```

- Insert(): 插入
- GetVictim(): 获取替换块

- Query(): 查询
- RemoveEle(): 剔除块

在本实验中，实现了替换策略为 LRU 以及 LRU-2 的 Cache，下面详细说明一下两者的实现细节。

2.1.2.1 LRU

LRU 替换策略是替换最近没有访问到的块

LRU 实现时采用的是双向链表 + 哈希表 (由于本实验中 frameID 不重复，因此不需要考虑哈希碰撞)

链表结点以及 LRU 结构定义如下

```
1  type linkNode struct {
2      frameID int
3      val     int
4      pre     *linkNode
5      post    *linkNode
6  }
7
8  type lruCache struct {
9      head      *linkNode
10     tail      *linkNode
11     MaxLength int
12     curLength int
13     hash      map[int]*linkNode
14 }
```

linkNode 链表结点

- frameID: frame 的编号
- val: frameID 的访问次数 (初始为 1) (方便后续使用 LRU-k 策略，如果只使用 LRU，可以不用该成员)
- pre: 指向该结点的前一个结点

- post: 指向该结点的后一个结点

lruCache 替换策略为 LRU 的 Cache

- head: 指向链表的头结点
- tail: 指向链表的尾结点
- MaxLength: Cache 的最大容量()
- curLength: Cache 目前已使用的容量
- hash: 哈希表 **frameID** → **linkNode**, 一一映射

lruCache 实例化

参数 maxLength

- 通常为前面定义的常量 DEFBUFSIZE 和 BufferSize 相同

```
1 func NewLRUCache(maxLength int) *lruCache {
2     // LRUCahe 的实例化
3     return &lruCache{
4         MaxLength: maxLength,
5         hash:      make(map[int]*linkNode),
6     }
7 }
```

Insert()

关键就是判定 cache 是否为空，如果是空，就把 head 和 tail 同时指向新的结点

否则，采用头插法把新结点插入到链表头部

最后，更新 head



```
1 func (l *lruCache) Insert(frameID int) {
2     var newNode *linkNode
3     if l.IsEmpty() { // cache 为空, 则将 head 和 tail 都设置为当前插入的节点
4         newNode = &linkNode{frameID: frameID, val: 1, pre: nil, post: nil}
5         l.tail = newNode
6     } else { // 否则, 将插入到头部
7         newNode = &linkNode{frameID: frameID, val: 1, pre: nil, post: l.head}
8         l.head.pre = newNode
9     }
10    l.head = newNode
11    // 存储在哈希表中, 保证 O(1) 时间消耗
12    l.hash[newNode.frameID] = newNode
13    if !l.IsFull() {
14        l.curLength += 1
15    }
16 }
```

GetVictim()

把链表尾部结点对应的 frameID 返回即可



```
1 func (l *lruCache) GetVictim() (frameID int) {
2     frameID = l.tail.frameID
3     return
4 }
```

RemoveEle()

在链表中删除对应为 frameID 的结点

该函数通常是调用 GetVictim() 函数之后调用, 将替换块从 cache 中删除

需要考虑以下情况

1. cache 中只有这一个元素 (既是头结点又是尾结点)
2. 该结点为头结点
3. 该结点为尾结点
4. 该结点为中间结点

不同情况下, 链表的操作不同



```
1 func (l *lruCache) RemoveEle(frameID int) {
2     node := l.hash[frameID]
3     if node == l.head && node == l.tail {
4         l.head, l.tail = nil, nil
5     } else if node == l.tail {
6         l.tail = node.pre
7         l.tail.post = nil
8     } else if node == l.head {
9         l.head = node.post
10        l.head.pre = nil
11    } else {
12        node.pre.post = node.post
13        node.post.pre = node.pre
14    }
15    delete(l.hash, frameID)
16    l.curLength -= 1
17 }
```

Query()

cache 的访问操作，将访问的结点移至链表的头部

同样考虑四种情况

1. cache 中只有这一个元素 (既是头结点又是尾结点) → 不做任何操作
2. 该结点为头结点 → 不做任何操作
3. 该结点为尾结点
4. 该结点为中间结点



```
1 func (l *lruCache) Query(frameID int) {
2     if l.isInLRU(frameID) {
3         queryNode := l.hash[frameID]
4         queryNode.val += 1
5         if queryNode == l.tail && queryNode == l.head {
6             return
7         } else if queryNode == l.head {
8             return
9         } else if queryNode == l.tail {
10            queryNode.pre.post = nil
11            l.tail = queryNode.pre
12        } else { // 访问的节点是头部和尾部之间的节点
13            queryNode.pre.post = queryNode.post
14            queryNode.post.pre = queryNode.pre
15        }
16        // 修改 cache 中的 head
17        l.head.pre = queryNode
18        queryNode.post = l.head
19        queryNode.pre = nil
20        l.head = queryNode
21    }
22 }
```

2.1.2.2 LRU-k

LRU-k 策略是维护两个 LRU cache，分别为 historyCache 和 bufferCache。

当 historyCache 中的结点访问次数达到 k 时(链表结点中的 val 成员)，从 historyCache 中移到 bufferCache 中。寻找替换块时，historyCache 不为空则到 history 中寻找，否则在 bufferCache 中寻找。尽量将访问次数达到 k 的结点存储在 cache 中。

新结点会插入到 historyCache 中，初始的 val 设定为 1。

LRU-k 的结构如下



```
1  type lru2Cache struct {  
2      k          int  
3      historyList *lruCache  
4      bufferList  *lruCache  
5  }
```

lru-k 实例化



```
1  func NewLRU2Cache(k int, maxLength int) *lru2Cache {  
2      return &lru2Cache{  
3          k:          k,  
4          historyList: NewLRUCache(maxLength),  
5          bufferList:  NewLRUCache(maxLength),  
6      }  
7  }  
8
```

需要注意的是，这里设定的两个 LRU Cache 的最大容量都是 `maxLength`，因为 Cache 是否满是通过上层的 `freeFrameNum` 决定的，当 buffer 已满，这两个 Cache 中结点数之和才是 `maxLength`。

当出现退化情况时，`historyCache` 为空，而 `bufferCache` 已使用的容量达到 `maxLength`，此时退化为 LRU

Insert()



```
1 func (l *lru2Cache) Insert(frameID int) {  
2     l.historyList.Insert(frameID)  
3 }
```

GetVictim()



```
1 func (l *lru2Cache) GetVictim() (frameID int) {  
2     if l.historyList.IsEmpty() {  
3         frameID = l.bufferList.GetVictim()  
4     } else {  
5         frameID = l.historyList.GetVictim()  
6     }  
7     return  
8 }
```

RemoveEle()



```
1 func (l *lru2Cache) RemoveEle(frameID int) {  
2     if l.historyList.isInLRU(frameID) {  
3         l.historyList.RemoveEle(frameID)  
4     } else {  
5         l.bufferList.RemoveEle(frameID)  
6     }  
7 }
```

Query()

如果需要访问的结点在 historyCache 中，判定其 val 是否达到 k，如果达到则将该结点从 historyCache 移至 bufferCache 中



```
1 func (l *lru2Cache) Query(frameID int) {  
2     if l.historyList.isInLRU(frameID) {  
3         l.historyList.Query(frameID)  
4         historyListHead := l.historyList.head  
5         if historyListHead.val >= l.k {  
6             l.bufferList.Insert(historyListHead.frameID)  
7             l.historyList.RemoveEle(historyListHead.frameID)  
8         }  
9     } else {  
10        l.bufferList.Query(frameID)  
11    }  
12 }
```

2.1.3 buffer manager 实例化

我这里只实现了两者策略

1. LRU
2. LRU-k

Parameters:

- isLRU: 是否采用 LRU 作为替换策略
- k: LRU-k 中的 k 值

```
1 func NewBMgr(isLRU bool, k int) *BMgr {
2     if isLRU {
3         return &BMgr{
4             freeFramesNum: DEFBUFSIZE,
5             cache:         NewLRUCache(DEFBUFSIZE),
6             dsmgr:         *newDSMgr(),
7         }
8     } else {
9         return &BMgr{
10            freeFramesNum: DEFBUFSIZE,
11            cache:         NewLRU2Cache(k, DEFBUFSIZE),
12            dsmgr:         *newDSMgr(),
13        }
14    }
15 }
```

2.1.4 BCB

由于没有实现高并发，把相应的锁和 count 成员注释

```
1 type BCB struct {
2     pageID int
3     frameID int
4     // latch int
5     // count int
6     dirty int
7     next *BCB
8 }
```

2.1.5 buffer manager 相关的接口

由于该实验中并没有实现并发操作，因此部分接口函数没有实现，下面只展示实现完成的接口

2.1.5.1 fixPage()

请求的 pageID 在缓冲区

记录其对应的 frameID，执行 cache 中的 Query 操作

请求的 pageID 不在缓冲区

缓冲区未满

将数据从磁盘中读取，buffer 提供一个空闲的 frameID，将其插入 cache 中，并 add 对应的 BCB

缓冲区已满

从 cache 中获取替换 frameID，将其对应的 BCB 以及 cache 结点删除

读操作则在 Cache 中执行 Query 操作，写操作则需要额外将对应的 bcb 块的 dirty 置为 1

```
1 func (b *BMgr) FixPage(pageID, prot int) (frameID int) {
2     bcb := b.getBCB(pageID)
3     if bcb != nil { // 如果请求的页在缓冲区中
4         b.hitCount += 1
5         frameID = bcb.frameID
6     } else { // 不在缓冲区中，将页从磁盘中取出并加入到 frame 中
7         if b.freeFramesNum != 0 { // 缓冲区没满
8             frameID = DEFBUFSIZE - b.freeFramesNum
9             b.freeFramesNum--
10        } else { // 缓冲区满了
11            frameID = b.selectVictim()
12            b.removeLRUEle(frameID)
13            b.removeBCB(b.ftop[frameID])
14        }
15        b.addBCB(pageID, frameID)
16        b.ftop[frameID] = pageID
17        b.Buf[frameID] = b.dsmgr.readPage(pageID)
18        b.incReadIO()
19        b.cache.Insert(frameID)
20    }
21    switch prot {
22    case 0: // 表示读数据
23        b.cache.Query(frameID)
24    default: // 表示写
25        // TODO: 修改 buf 中对应 frameID 的数据
26        b.cache.Query(frameID)
27        b.setDirty(frameID)
28    }
29    return
30 }
```

2.1.5.2 fixNewPage()

向堆文件中写入一个 page 页大小的数据，数据是随机生成的；初始时，需要向 data.dbf 中写入 50000 页的数据。

初始化时，我没有选择将部分页存储 buffer 中，也可以选择初始存入 buffer。

```
1 func (b *BMgr) FixNewPage() int {
2     b.dsmgr.incNumPages()
3     pageID := b.dsmgr.GetNumPages()
4     bytes := make([]byte, FRAMESIZE)
5     for i := 0; i < FRAMESIZE; i++ {
6         bytes[i] = byte(rand.Intn(256))
7     }
8     b.dsmgr.writePage(pageID, BFrame{Filed: [4096]byte(bytes)})
9     b.dsmgr.setUse(pageID-1, 1)
10    return pageID
11 }
```

2.1.5.3 NumFreeFrames()

```
1 func (b *BMgr) NumFreeFrames() int {
2     return b.freeFramesNum
3 }
```

2.1.5.4 selectVictim()

选取替换块，底层调用的是 cache 中的 GetVictim 函数

```
1 func (b *BMgr) selectVictim() (frameID int) {
2     frameID = b.cache.GetVictim()
3     return
4 }
```

2.1.5.5 hash()

hash 函数, pageID \rightarrow ptof 数组的下标

如果出现哈希冲突, 则将 bcb 块在链表尾部进行插入



```
1 func (b *BMgr) hash(pageID int) (frameIndex int) {
2     frameIndex = pageID % DEFBUFSIZE
3     return
4 }
```

2.1.5.6 removeBCB()


如果删除的块 dirty 位为 1, 则将其写回磁盘



```
1 func (b *BMgr) removeBCB(pageID int) {
2     var pre *BCB
3     var p *BCB
4     frameIndex := b.hash(pageID)
5     head := b.ptof[frameIndex]
6     for p = head; p != nil; p = p.next { // 遍历链表, 找到对应的 BCB 节点, 将其删除
7         if p.pageID == pageID {
8             break
9         }
10        pre = p
11    }
12    if pre != nil {
13        pre.next = p.next
14    } else {
15        b.ptof[frameIndex] = p.next
16    }
17    if p.dirty == 1 {
18        b.dsmgr.writePage(p.pageID, b.Buf[p.frameID])
19        b.incWriteIO()
20    }
21 }
```

2.1.5.7 removeLRUEle()


调用 cache 的 removeEle 函数



```
1 func (b *BMgr) removeLRUEle(frameID int) {
2     b.cache.RemoveEle(frameID)
3 }
```

2.1.5.8 setDirty()


获取对应的 bcb 块，将其 dirty 置为 1



```
1 func (b *BMgr) setDirty(frameID int) {
2     pageID := b.ftop[frameID]
3     bcb := b.getBCB(pageID)
4     bcb.dirty = 1
5 }
```

2.1.5.9 writeDirtys()

结束前将所有的脏块写回磁盘中，遍历所有的 bcb 块即可；如果 dirty 位为 1，则将其写回磁盘中。



```

1 func (b *BMgr) writeDirtyys() {
2     for _, bcb := range b.ptof {
3         for p := bcb; p != nil; p = p.next {
4             if p.dirty != 0 {
5                 b.dsmgr.writePage(p.frameID, b.Buf[p.frameID])
6                 b.incWriteIO()
7             }
8         }
9     }
10 }

```

2.1.5.10 getBCB()

获取页号对应的 bcb 块，可以用于判定 buffer 中是否存在该页号

Parameters


pageID int

查询的页号

Returns

*BCB

遍历其对应的链表，如果存在某结点的 pageID 成员和输入的参数相等，则将该结点返回，否则返回 nil



```

1 func (b *BMgr) getBCB(pageID int) *BCB {
2     frameIndex := b.hash(pageID)
3     for p := b.ptof[frameIndex]; p != nil; p = p.next {
4         if p.pageID == pageID {
5             return p
6         }
7     }
8     return nil
9 }

```


2.1.5.11 addBCB()

添加对应 BCB 块，通常是在 fixPage 函数中插入新块在 buffer 中调用，在对应的链表中添加新结点

Parameters

pageID int

页号


frameID int

对应的帧号

```
1 func (b *BMgr) addBCB(pageID, frameID int) {
2     frameIndex := b.hash(pageID)
3     head := b.ptof[frameIndex]
4     if head == nil {
5         b.ptof[frameIndex] = &BCB{
6             pageID: pageID,
7             frameID: frameID,
8         }
9         return
10    }
11    for p := head; ; p = p.next {
12        if p.next == nil {
13            p.next = &BCB{
14                pageID: pageID,
15                frameID: frameID,
16            }
17            return
18        }
19    }
20 }
```

2.2 data storage manager


数据存储管理器结构如下



```
1  const MAXPAGES = 50000
2
3  type DSMgr struct {
4      currFile *os.File
5      numPages int
6      pages    [MAXPAGES]int
7  }
```

- MAXPAGES 最大的页数
- currFile: 当前的文件指针
- numPages: page 的数量
- pages: page 是否可用的标志位数组

2.2.1 openFile()



```
1  func (d *DSMgr) openFile(fileName string) int {
2      f, err := os.OpenFile(fileName, os.O_CREATE|os.O_RDWR, 0644)
3      if err != nil {
4          return 0
5      }
6      d.currFile = f
7      return 1
8  }
```

2.2.2 closeFile()



```
1 func (d *DSMgr) closeFile() int {
2     err := d.currFile.Close()
3     if err != nil {
4         return 0
5     } else {
6         return 1
7     }
8 }
```

2.2.3 seek()

将文件指针进行偏移，从 pos 位置开始偏移 offset 字节，offset 通常是 (pageID - 1) * FRAMESIZE



```
1 func (d *DSMgr) seek(offset int64, pos int) int {
2     _, err := d.currFile.Seek(offset, pos)
3     if err != nil {
4         return 0
5     } else {
6         return 1
7     }
8 }
```

2.2.4 readPage()

先将文件指针偏移 to 该页的起始位置，再从磁盘中读取一个页的数据

Parameters

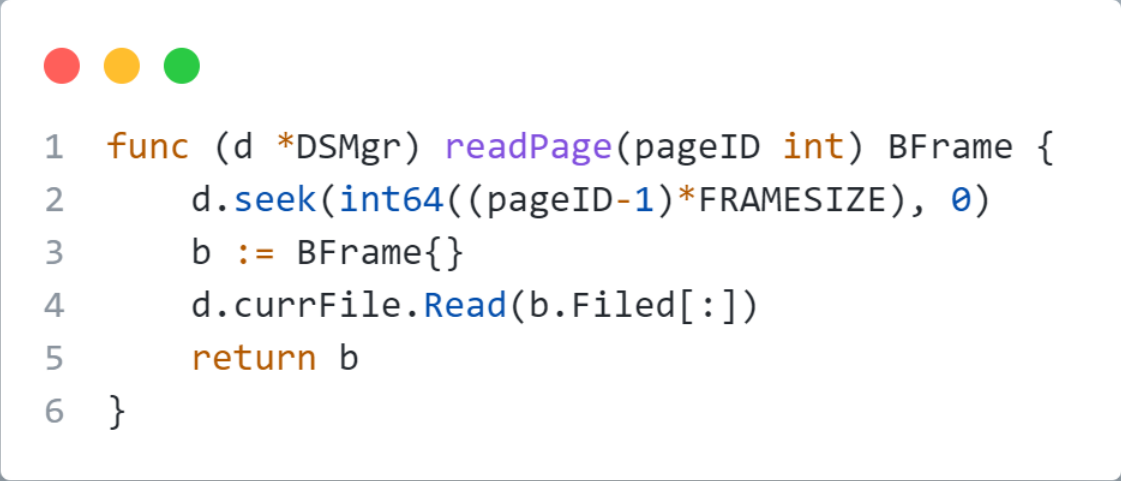
pageID int

页号

Returns:

BFrame

读取的数据



```
1 func (d *DSMgr) readPage(pageID int) BFrame {
2     d.seek(int64((pageID-1)*FRAMESIZE), 0)
3     b := BFrame{}
4     d.currFile.Read(b.Filed[:])
5     return b
6 }
```

2.2.5 writePage()

类似地，也是将文件指针先偏移 to 该页的起始位置，然后向文件写入一个页的数据

Parameters

pageID int

页号


frm BFrame

写入的数据

Returns

n int


写入成功的字节数



```
1 func (d *DSMgr) writePage(pageID int, frm BFrame) int {
2     d.seek(int64((pageID-1)*FRAMESIZE), 0)
3     n, _ := d.currFile.Write(frm.Filed[:])
4     return n
5 }
```


2.2.6 incNumPages()

增加对应的 numPages



```
1 func (d *DSMgr) incNumPages() {
2     d.numPages += 1
3 }
```

2.2.7 GetNumPages()



```
1 func (d *DSMgr) GetNumPages() int {
2     return d.numPages
3 }
```

2.2.8 GetFile()

获取当前的文件指针




```
1 // 返回当前的文件指针
2 func (d *DSMgr) GetFile() (file *os.File) {
3     return d.currFile
4 }
```

2.2.9 setUse() 和 GetUse()

将 pages 数组对应下标设置为 use_bit

返回对应下标的 use_bit



```
1 func (d *DSMgr) setUse(index, use_bit int) {
2     d.pages[index] = use_bit
3 }
4
5 func (d *DSMgr) GetUse(index int) int {
6     return d.pages[index]
7 }
```

3. 实验结果

3.1 LRU 结果

```
lru:
Hit Count is 169565
runtime is 4.358896s
ReadBytes: 1353461760 bytes
WriteBytes: 706093056 bytes
```

- hit Count 命中次数
- runtime 运行时间
- ReadBytes 读取的字节数
- WriteBytes 写入的字节数

3.2 LRU-k 结果

下面测试了 k = 2、3、4、5 的结果

3.2.1 k = 2

```
lru2:
Hit Count is 169565
runtime is 4.470704s
ReadBytes: 1353461760 bytes
WriteBytes: 706093056 bytes
```

3.2.2 k = 3

```
lru3:
Hit Count is 217859
runtime is 3.269918s
ReadBytes: 1155649536 bytes
WriteBytes: 563982336 bytes
```

3.2.3 k = 4

```
lru4:
Hit Count is 213537
runtime is 4.191304s
ReadBytes: 1173352448 bytes
WriteBytes: 579604480 bytes
```

3.2.4 k = 5

```
lru5:
Hit Count is 202061
runtime is 4.071334s
ReadBytes: 1220358144 bytes
WriteBytes: 609939456 bytes
```

4. 文件结构

buffer manager

— Storage and Buffer Manager.pdf	实验报告
— buffer	buffer package
— BCB.go	BCB 数据结构
— BufferManager.go	buffer manager
— Cache.go	cache
— DataStorageManager.go	data storage manager
— Frame.go	frame 数据结构以及常量的定义
— go.mod	
— go.sum	
— go.work	
— main.go	主文件入口

5. 总结与收获

更加深入理解了 buffer manager 的底层工作原理。