第二次数据库实验报告

因为第一次报告中设计为整体框架，对于内存管理部分，已经实现了此次的大部分要求，第二次数据库实验只是在第一次基础上添加了两种页面置换算法的实现。

一、内存区域划分

在设计的时候，我使用`BufferPoolBuilder`来制定缓冲池的大小，并且提供了`build()`方法来创建，因此要是为`sql访问计划`、`数据缓存`以及`日志缓存`提供相应的`Buffer`的话，直接创建即可。

而对应的数据字典、表结构等等数据，在设计时打开对应表的时候会创建对应的`TableHandler`，里面就对第二页数据进行了解析并且得到了表结构等信息。

```
#[test]
    fn test_bufferPool(){
        let log_buffer_pool_builder = BufferPoolBuilder::new().with_size(10);
        let log_buffer_pool = log_buffer_pool_builder.build();
        let sql_buffer_pool_builder = BufferPoolBuilder::new().with_size(20);
        let sql_buffer_pool_builder = sql_buffer_pool_builder.build();
        let buffer_pool_builder = BufferPoolBuilder::new().with_size(5);
        let buffer_pool = buffer_pool_builder.build();
    }
```

二、缓冲载入

当缓冲池中有空闲页面可以载入时，直接放入缓冲池并且加标志即可。

如果缓冲池中有页面，直接读取，并且如果是进行插入的话，会调用`make_dirty`方法将页面标记为脏页。

测试代码如下：

```
#[test]
    fn test_page_get(){
        let mut buffer_pool = BufferPool::default();
        let buffer_pool_rc = Rc::new(RefCell::new(buffer_pool));
        let table_manager = TableManager { buffer_pool: buffer_pool_rc.clone() };
        let table_handler =  table_manager.open_table("user");
        let mut page_handler = table_handler.page_handler;
        let vec = page_handler.get_items_by_page(0);

        let t = table_manager.open_table("user");
        for v in vec.iter() {
            t.parse_item(v);
        }

        let vec3 = page_handler.get_items_by_page(0);
        for v in vec.iter() {
            t.parse_item(v);
        }
    }
```

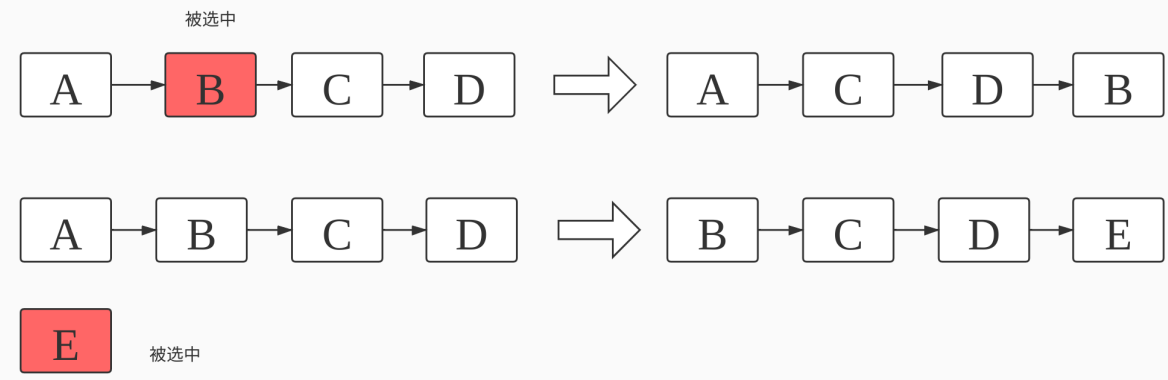
```
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is user,the page_id is 1
finish lru
属性名为:userID
22
属性名为:userName
小李
属性名为:userID
22
属性名为:userName
小李
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is user,the page_id is 1
finish lru
属性名为:userID
22
属性名为:userName
小李
属性名为:userID
22
属性名为:userName
小李
```

三、页面置换算法

1. LRU 页面置换算法

`LRU`页面置换算法就是选择最近最久未使用的页面予以淘汰，基本的实现方法就是维护一个链表，如果有访问到缓冲中存在的页面的话，就将其放置到最后面，如果没有的话，就选取链表的头写入对应页，并放到链表尾部。

具体的流程如图：



但是，因为`rust`特有的所有权机制的问题，要写一个可以支持并发的链表不是很容易，所以我在实现的时候使用的`Vec`数组进行模拟，即每次如果要找的页在缓冲中，就将其后面的所有页往前移动一格，并将其放到末尾。若在缓冲中不存在，就将所有页往前移动一格并且将其放到末尾。

具体流程为：先进行一遍扫描，如果扫描得到了对应的页的话，就将该页后面的所有页往前移动一格，并且将该页放置到数组尾部，并返回该页数据。如果扫描没有得到对应的页的话，就调用磁盘管理器`DiskManager`获取对应的页，获取后将除第一页外所有页往前移动一位，将获取的新页加到尾部并返回。

具体实现代码为：

```
pub fn get_page_lru(&mut self, file_name:&str, page_id:u32) ->BufferReference{
        // 如果需要的页在缓存中存在的话，就直接返回
        let mut mut_buffer = self.buffers.clone();
        let k = self.buffers.clone();
        let mut buffer_ref:Option<BufferReference> = None ;
        // 遍历查看页是否已经在缓冲中
        for (index,buffer) in self.buffers.as_slice().iter().enumerate(){
            let d:&RefCell<Buffer> = (*buffer).borrow();
            let e = d.borrow();
            if e.file_name == file_name && e.page_id == page_id {
                for i in index..(self.pool_size-1) as usize {
                    if let Some(v_i) = mut_buffer.get_mut(i){
                        *v_i = k[i+1].clone();
                    }
                }
                if let Some(v_i) = mut_buffer.get_mut((self.pool_size-1)as usize)
{
                    *v_i = buffer.clone();
                }
                buffer_ref = Some(BufferReference{
                    buffer: buffer.clone()
                });
            }
        };
        // 如果有的话 就更新缓冲池并且返回
        if buffer_ref.is_some() {
            self.buffers = mut_buffer;
            for buffer in self.buffers.as_slice().iter() {
                let d:&RefCell<Buffer> = (*buffer).borrow();
                let e = d.borrow();
                println!("the file_name is {},the page_id is
{}",e.file_name,e.page_id);
            }
            println!("finish lru");
            return buffer_ref.unwrap();
        }
        //如果没有的话，要从磁盘中读取
        let disk_handler = DiskManager::get_file(file_name);
        let page = disk_handler.get_page(page_id);

        for i in 0..(self.pool_size - 1) as usize {
            if let Some(v_i) = self.buffers.get_mut(i){
                *v_i = k[i+1].clone();
            }
        }
```

```
        let s = Buffer{
            file_name: String::from(file_name),
            is_dirty: false,
            page_id,
            buffer: page,
            is_used: false
        };
        let return_buffer = Rc::new(RefCell::new(s));
        if let Some(v_i) = self.buffers.get_mut((self.pool_size -1) as usize){
            *v_i = return_buffer.clone();
        }

        let r = BufferReference{
            buffer: return_buffer.clone()
        };
        for buffer in self.buffers.as_slice().iter() {
            let d:&RefCell<Buffer> = (*buffer).borrow();
            let e = d.borrow();
            println!("the file_name is {},the page_id is
{}",e.file_name,e.page_id);
        }
        println!("finish lru");
        r
    }
```

对应的测试代码为:

```
#[test]
    fn test_lru(){
        let mut buffer_pool = BufferPool::default();
        let file_name = "user";
        buffer_pool.get_page_lru(file_name,1);
        buffer_pool.get_page_lru(file_name,3);
        buffer_pool.get_page_lru(file_name,1);
        buffer_pool.get_page_lru(file_name,4);
        buffer_pool.get_page_lru(file_name,5);
        buffer_pool.get_page_lru(file_name,6);
        buffer_pool.get_page_lru(file_name,2);
    }
```

测试结果为:

```
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is user,the page_id is 1
finish lru
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is user,the page_id is 1
the file_name is user,the page_id is 3
finish lru
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
```

```
the file_name is ,the page_id is 0
the file_name is user,the page_id is 3
the file_name is user,the page_id is 1
finish lru
the file_name is ,the page_id is 0
the file_name is ,the page_id is 0
the file_name is user,the page_id is 3
the file_name is user,the page_id is 1
the file_name is user,the page_id is 4
finish lru
the file_name is ,the page_id is 0
the file_name is user,the page_id is 3
the file_name is user,the page_id is 1
the file_name is user,the page_id is 4
the file_name is user,the page_id is 5
finish lru
the file_name is user,the page_id is 3
the file_name is user,the page_id is 1
the file_name is user,the page_id is 4
the file_name is user,the page_id is 5
the file_name is user,the page_id is 6
finish lru
the file_name is user,the page_id is 1
the file_name is user,the page_id is 4
the file_name is user,the page_id is 5
the file_name is user,the page_id is 6
the file_name is user,the page_id is 2
finish lru
```

可以看到，当缓冲池大小为5的时候，访问顺序为`1 3 1 4 5 6 2`的时候，最后访问2时去掉的是最久时间没有被访问的`3`。

2. Clock页面置换算法

时钟页面置换算法考虑了页面有没有被修改的情况，在进行访问的时候进行多次循环，先循环一遍看是否页在缓冲中，第二次再遍历得到没有被使用的页面进行替换，第三次遍历得到被使用但是没有被修改的页面进行替换，最后一次遍历找一个被使用且是脏页的页面进行替换。

在此也是一样，通过数组来实现时钟算法，同时，因为那几次遍历实际上都是条件不同，其他的循环方式等都是一样的，因此直接将循环方法进行封装，得到更简洁的代码：

```
fn change_page_clock(&mut self,file_name:&str, page_id:u32,page:&
[u8;SIZE],is_used_n:bool,is_dirty_n:bool)  ->Option<BufferReference>{
        let mut mut_buffer = self.buffers.clone();
        let mut buffer_ref:Option<BufferReference> = None ;
        for (index,buffer) in self.buffers.as_slice().iter().enumerate(){
            let d:&RefCell<Buffer> = (*buffer).borrow();
            let e = d.borrow();
            if e.is_used == is_used_n && e.is_dirty == is_dirty_n {
                let s = Buffer{
                    file_name: String::from(file_name),
                    is_dirty: false,
                    page_id,
                    buffer: *page,
```

```
                    is_used: true
                };
                let return_buffer = Rc::new(RefCell::new(s));
                if let Some(v_i) = mut_buffer.get_mut(index){
                    *v_i = return_buffer.clone();
                }
                buffer_ref = Some(BufferReference{
                    buffer: buffer.clone()
                });
                break;
            }
        };

        if buffer_ref.is_some() {
            self.buffers = mut_buffer;
            for buffer in self.buffers.as_slice().iter() {
                let d:&RefCell<Buffer> = (*buffer).borrow();
                let e = d.borrow();
                println!("the file_name is {},the page_id is {},the is_dirty is
{},the is_used is {}",
                         e.file_name,e.page_id,e.is_dirty,e.is_used);
            }
            println!("finish clock");
            return buffer_ref;
        }else {
            return None;
        }
    }

    pub fn get_page_clock(&mut self, file_name:&str, page_id:u32)-
>BufferReference{
        //先遍历一遍，找有没有存在在缓冲里
        let mut mut_buffer = self.buffers.clone();
        let mut buffer_ref:Option<BufferReference> = None ;
        for (index,buffer) in self.buffers.as_slice().iter().enumerate(){
            let d:&RefCell<Buffer> = (*buffer).borrow();
            let e = d.borrow();
            if e.file_name == file_name && e.page_id == page_id {
                buffer_ref = Some(BufferReference{
                    buffer: buffer.clone()
                });
                break
            }
        };
        if buffer_ref.is_some() {
            self.buffers = mut_buffer;
            for buffer in self.buffers.as_slice().iter() {
                let d:&RefCell<Buffer> = (*buffer).borrow();
                let e = d.borrow();
                println!("the file_name is {},the page_id is {},the is_dirty is
{},the is_used is {}",
                         e.file_name,e.page_id,e.is_dirty,e.is_used);
            }
            println!("finish clock");
            return buffer_ref.unwrap();
        }
        // 要是找不到的话，要从磁盘中读
        let disk_handler = DiskManager::get_file(file_name);
```

```
        let page = disk_handler.get_page(page_id);
        //先遍历一遍 找没有使用的
        let buffer_unused =
self.change_page_clock(file_name,page_id,&page,false,false);
        if buffer_unused.is_some() {
            return buffer_unused.unwrap();
        }
        //再遍历一遍，找使用但是没有被修改的
        let buffer_used =
self.change_page_clock(file_name,page_id,&page,true,false);
        if buffer_used.is_some() {
            return buffer_used.unwrap();
        };
        //最后遍历一遍，直接替换一个被使用的
        let buffer_final =
self.change_page_clock(file_name,page_id,&page,true,true);
        buffer_final.unwrap()
    }
```

对应的测试代码为:

```
#[test]
    fn test_clock(){
        let mut buffer_pool = BufferPool::default();
        // let buffer_pool_rc = Rc::new(RefCell::new(buffer_pool));
        let file_name = "user";
        buffer_pool.get_page_clock(file_name,1);
        buffer_pool.get_page_clock(file_name,2);
        buffer_pool.get_page_clock(file_name,3);
        buffer_pool.get_page_clock(file_name,4);
        buffer_pool.get_page_clock(file_name,5);
        buffer_pool.make_dirty(file_name,1);
        buffer_pool.get_page_clock(file_name,6);
    }
```

对应的测试结果为:

```
the file_name is user,the page_id is 1,the is_dirty is false,the is_used is true
the file_name is ,the page_id is 0,the is_dirty is false,the is_used is false
the file_name is ,the page_id is 0,the is_dirty is false,the is_used is false
the file_name is ,the page_id is 0,the is_dirty is false,the is_used is false
the file_name is ,the page_id is 0,the is_dirty is false,the is_used is false
finish clock
the file_name is user,the page_id is 1,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 2,the is_dirty is false,the is_used is true
the file_name is ,the page_id is 0,the is_dirty is false,the is_used is false
the file_name is ,the page_id is 0,the is_dirty is false,the is_used is false
the file_name is ,the page_id is 0,the is_dirty is false,the is_used is false
finish clock
the file_name is user,the page_id is 1,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 2,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 3,the is_dirty is false,the is_used is true
the file_name is ,the page_id is 0,the is_dirty is false,the is_used is false
the file_name is ,the page_id is 0,the is_dirty is false,the is_used is false
finish clock
the file_name is user,the page_id is 1,the is_dirty is false,the is_used is true
```

```
the file_name is user,the page_id is 2,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 3,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 4,the is_dirty is false,the is_used is true
the file_name is ,the page_id is 0,the is_dirty is false,the is_used is false
finish clock
the file_name is user,the page_id is 1,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 2,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 3,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 4,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 5,the is_dirty is false,the is_used is true
finish clock
the file_name is user,the page_id is 1,the is_dirty is true,the is_used is true
the file_name is user,the page_id is 6,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 3,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 4,the is_dirty is false,the is_used is true
the file_name is user,the page_id is 5,the is_dirty is false,the is_used is true
finish clock

Process finished with exit code 0
```

在这里，还是安排了缓冲池大小为`5`,并且在运行时改变了`1`的状态，使其变成脏页，最后得到的结果可以看到，虽然1在前面，但是却因为是脏页所以没有被替换。

四、总结

本次实验中遇到的问题主要有：

- 对`rust`不是很熟悉，因为其所有权机制的存在，链表的实现比较困难，所以对于两个页面置换算法就放弃了用链表实现的想法，使用数组来进行实现。