Buffer Manager 设计报告

数字媒体技术 3180101044 沈吕可晟

数字媒体技术 3180101939 陆子仪

一、模块概述

Buffer Manager 用于管理缓冲区以及物理文件。主要功能如下:

1）将指定数据从缓冲区写回物理文件或者从物理文件中读入缓冲区;

2）建立新的 dbs 文件，并且将初始化信息写入文件

3）记录缓冲区各页的状态，实现替换算法，当缓冲区满的时候将适合的页进行替换。

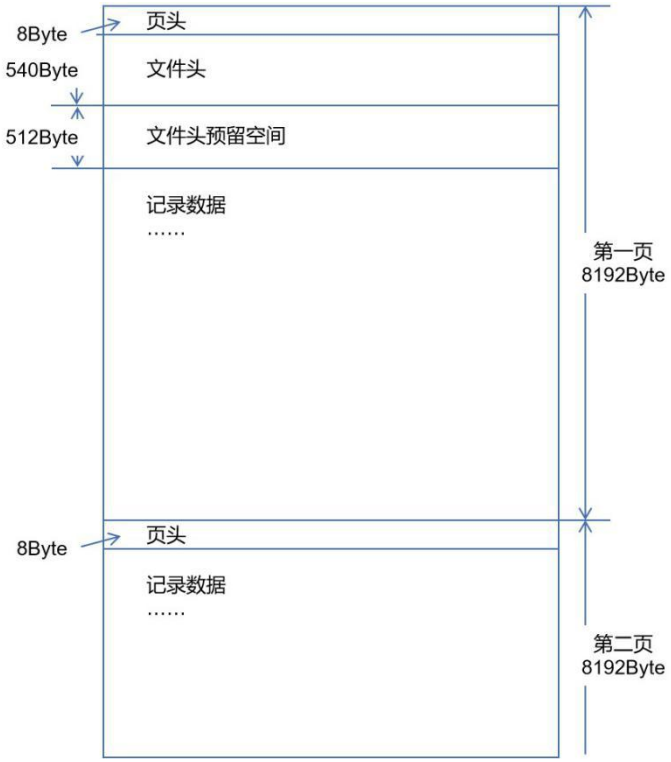
4）对底层的文件管理

二、设计思路

Buffer Manager 的设计围绕着缓冲区和物理文件展开。

缓冲区中的数据和物理文件中的记录以页（Page）为单位进行交互，为提高 I/O 的效率，每一页的大小被设为固定的 8K。读入的页的信息储存在 Page 类中，包括了页头指针、文件指针、文件页号、最近一次访问内存是否被使用、是否脏页、内存中实际保存物理文件数据的起始地址等信息。

在物理文件部分，每一个表中的数据被写入.dbs 文件，该文件的结构如下:



文件头记录与文件和表相关的信息，页头中记录与每一页相关的信息，然后在数据区记录一条条的元组数据。对于物理文件中的某一条数据（元组）由 FileAddress 记录，其结构为<物理文件内页编号，页内偏移量>。

三、数据结构以及类定义

1. 页头（PageHead）

记录了该页的页编号以及是否被锁定（用于替换算法）

```
class PageHead {
public:
    void SetPageId(unsigned long PageId);
    void SetFixed(bool IsFixed);
    void Initialize();
private:
    unsigned long PageId;
    bool IsFixed;  // 内存页是否被锁定，如果被锁定则不会被关闭
};
```

2. 页（Page）

记录了页的信息，并提供将内存中的页写回物理文件以及获取文件头指针的接口函数，并且可以设置该页是否被修改以及最后一次访问内存是否被使用。

```
class Page {
    friend class Buffer;
    friend class Clock;
    friend class File;
public:
    PageHead* Pagehead;     // 页头指针
    FILE* FilePointer;      // 文件指针，当其置为 NULL 时，该页被抛弃
    unsigned long FilePageID;     // 文件页号
    mutable bool IsLastUsed;      // 最近一次访问内存是否被使用，用于
Clock 算法
    mutable bool IsModified;      // 是否脏页
    void *PtrtoPageBeginAddress; // 内存中实际保存物理文件数据的起始地
址
    /* 成员函数 */
    void WriteMemToFile() const;  // 把内存中的页写回到文件中
    FileHead* GetFileHead();      // 获取文件头指针（当文件页为 0 的时
候调用此函数，来返回对应的文件头指针，即指向文件头开始的地址）
    Page();
    ~Page();
    void SetModified(bool State);
    void SetLastUsed(bool State);
};
```

3. 物理文件地址（FileAddress）

物理文件地址表示了某条记录在物理文件中的具体位置，通过页号和偏移量进行标识

```
class FileAddress {
    friend class File;
public:
    // 设置 FileAddress 的变量值
    void SetFileAddress(unsigned long FilePageID, unsigned int  OffSet);
    bool operator==(const FileAddress &rhs) const;
    bool operator!=(const FileAddress &rhs) const;
    bool operator<(const FileAddress &rhs)const;
```

```
    unsigned long FilePageID;      // 文件页编号
    unsigned int  OffSet;          // 页内偏移量
};
```

4. 文件头（FileHead)

文件头类，该信息被写在文件的第一页开头

```
class FileHead {
public:
    void Initialize();       // 初始化新的文件的文件头信息
    FileAddress DelFirst;    // 第一条被删除记录地址
    FileAddress DelLast;     // 最后一条被删除记录地址
    FileAddress NewInsert;   // 文件末尾可插入新数据的地址

    unsigned long TotalPage;                // 目前文件中共有页数
    char Reserve[FILECOND_RESERVE_SPACE];  // 预留空间
};
```

5. 文件类（File)

记录一个具体的物理文件 (.dbs) 的类，提供主要的接口函数供数据的读取、添加、更新和删除，除去添加都需要提供具体的 FileAddress。

```
class File {
    friend class Buffer;
    friend class Page;
public:
    char* GetFileName();
    char FileName[MAX_FILENAME_LEN];
    FILE* FilePointer;         // 文件指针
    unsigned long TotalPage;  // 目前文件中共有页数

    /* 函数 */
    File(const char *FileName, FILE* FilePointer);
    Page* AddExtraPage();      // 当前文件添加一页空间
    Page* GetFileFirstPage(); // 得到文件首页

    // 读取内存文件,返回读取位置指针
    void* ReadMem(FileAddress *ReadAddress);
```

```
    // 在可写入地址写入数据
    FileAddress WriteMem(const void* Source, unsigned int Length);
    FileAddress WriteMem(const void* Source, unsigned int Length,
FileAddress* Dest);


    /* 对外接口函数 */
    // 读取某条记录,返回记录指针(包括记录地址数据)
    const void* ReadRecord(FileAddress *RecordAddress)const;
   // 添加一条记录,返回记录所添加的位置
    FileAddress AddRecord(const void* RecordSource, unsigned int
SizeofRecord);
    // 删除一条记录,返回删除的位置
    FileAddress DeleteRecord(FileAddress *DeleteAddress, unsigned int
SizeofRecord);
    // 更新一条记录,返回成功与否
    bool UpdateRecord(FileAddress *DeleteAddress, void
*UpdateRecordData, unsigned int SizeofRecord);
};
```

6. 内存管理类（Clock）

进行缓冲区的内存管理，设定为缓存区一共可以处理 4k 张页，当缓存区满时，替换掉合适的页.

```
class Clock {  //内存页管理类
    friend class File;
    friend class Buffer;
public:
    Clock();
    ~Clock();
    Page* MemPages[MemPageCount + 1];  // 内存页对象数组  MemPageCount
是内存页数量


    /* 函数 */
    // 获得物理文件页（某一物理文件页用文件指针和页编号表示）在内存中
的地址
    Page* GetMemAddr(FILE* FilePointer, unsigned long FilePageID);
    // 返回一个可替换的内存页索引，将原页面内容该写回先写回
    unsigned int GetUseablePage();
    // 创建新页，适用于创建新文件或者添加新页的情况下
    Page* CreateNewPage(FILE* FilePointer, unsigned long FilePageId);
```

```
};
```

7. Buffer 类

读写文件使用的最底层的类。

```
class Buffer {
    friend class Clock;


public:
    ~Buffer();
    File* operator[](const char *fileName);  // 打开文件，打开失败返回
nullptr
    void CreateFile(const char *FileName);
    File* GetFile(const char *FileName);

    void CloseFile(const char *FileName);
    void CloseAllFile();
private:
    std::vector<File*> MemFiles;  // 保存已经打开的文件列表,存的是文件
指针
};
```

四、关键函数及代码

1. 替换算法

算法先查找有没有没分配的内存页，如果有直接拿来用就行，同时在此次遍历中确认被抛弃的内存页，方便若无未分配的内存页可以直接返回被抛弃的内存页。若两者皆无，则采用 LRU 算法找到合适替换的内存页进行替换。

```
unsigned int Clock::GetUseablePage()  //找一个可以用来记录的页
{
    // 先查找有没有没分配的内存页，如果有直接拿来用就行了
    for (int i = 1; i <= MemPageCount; i++)
    {
        if (MemPages[i] == NULL)
        {
            MemPages[i] = new Page();
            return i;
        }
        if (MemPages[i]->FilePointer == NULL) {
```

```
            flag = i;          //此处暂时记录一个被抛弃的内存页，如果找完所有没有空的内存页的话，就使用被抛弃的内存页，不用再查找一遍了
        }
    }
    //执行到此处即没有未分配的内存页了，判断一下之前的循环中有没有找到被抛弃的内存页
    if (flag == 0 && MemPages[0]->FilePointer != NULL) {   //若 flag 没变，并且内存页中第 0 页未被抛弃则说明，没有已经被抛弃的内存页,要使用置换算法了
        unsigned int i = 1;
        static unsigned long index = 1;
        if (MemPages[index] != nullptr) {
            throw Error("Error!");
        }
        while (MemPages[index]->IsLastUsed)     // 最近被使用过
        {
            MemPages[index]->IsLastUsed = 0;
            index = (index + 1) % MemPageCount;
            if (index == 0)index++;
        }
        auto res = index;
        MemPages[index]->IsLastUsed = 1;
        index = (index + 1) % MemPageCount;
        if (index == 0)index++;
        i = res;
        if (i == 0)i++;  //如果刚好凑巧碰到了第 0 页的话是不行的，往后移 1 页。
        MemPages[i]->WriteMemToFile();  //把这个下标对应的内存页写回文件中
        return i;
    }
    else {  //有被抛弃的内存页
        return flag;     //将被抛弃的内存页的数组坐标数返回
    }
}
```

2、新页创建函数

用于创建新页，适用于创建新文件或者添加新页的情况下。

```
Page* Clock::CreateNewPage(FILE* FilePointer, unsigned long FilePageId)
{
    // 初始化新的内存页对象
    int index = GetUseablePage();   //找到可用页
    memset(MemPages[index]->PtrtoPageBeginAddress, 0, PageSize);  //设置可用页的数据存储区域数据为 0
    MemPages[index]->FilePointer = FilePointer;
    MemPages[index]->FilePageID = FilePageId;
    MemPages[index]->SetModified(true);  // 新页设置为脏页，需要写回
```

```
    // 初始化新页的页头信息
    MemPages[index]->Pagehead->SetPageId(FilePageId); //设置页头中 页的 id 信息
    if (FilePageId != 0)
    {
        MemPages[index]->Pagehead->SetFixed(false);
    }
    else
    {
        MemPages[index]->Pagehead->SetFixed(true);      //如果新的页是一个文件的首页的
话，会把他锁定在内存中，不会被释放写回
        MemPages[index]->GetFileHead()->Initialize();  //再把这个文件页的文件头的信息
也进行初始化
    }
    return MemPages[index];
}
```

3、创建文件函数

在系统中创建相关文件的函数

```
void Buffer::CreateFile(const char *FileName)
{
    // 文件存在 创建失败
    if (fopen(FileName, "rb") != NULL)
    {
        throw Error("File Is Existed");
    }
    //创建文件
    FILE* NewFile = fopen(FileName, "wb+"); // 新建文件(打开文件)

    void *ptr = malloc(PageSize);    //PageSize 为 1 页文件页所规定的大小
    memset(ptr, 0, PageSize);
    PageHead* PageHeadPointer = (PageHead *)(ptr);
    FileHead* FileHeadPointer = (FileHead *)((char*)ptr + PageHeadSize);
    PageHeadPointer->Initialize();
    FileHeadPointer->Initialize();
    // 写回
    fwrite(ptr, PageSize, 1, NewFile);  /*size_t fwrite(const void *ptr, size_t size,
size_t nmemb, FILE *stream)
                ptr-- 这是指向要被写入的元素数组的指针。
                size-- 这是要被写入的每个元素的大小，以字节为单位。
                 nmemb-- 这是元素的个数，每个元素的大小为 size 字节。
                stream-- 这是指向 FILE 对象的指针，该 FILE 对象指定了一个输出流。*/
    fclose(NewFile);
    delete ptr;
    return;
```

```
}
```

4、获取文件信息

```
File* Buffer::GetFile(const char *FileName) {
    //先查看该文件是否已经打开
    for (int i = 0; i < this->MemFiles.size(); i++) {
        if ((strcmp(MemFiles[i]->GetFileName(), FileName) == 0))
            return MemFiles[i];
    }
    //若循环结束还未 return，则说明此文件尚未打开，则打开该文件
    FILE* fp = fopen(FileName, "rb+");
    if (fp == NULL)
    {
        return NULL;
    }
    else {
        File* NewFile = new File(FileName, fp);
        //????读取过程较为繁杂
        this->MemFiles.push_back(NewFile);
        return NewFile;
    }
}
```

5、获取文件在内存中的映射指针

先从文件 clock 中查找该页是否已经在内存中，如果在则直接获取，不在则从磁盘读入

```
Page* Clock::GetMemAddr(FILE* FilePointer, unsigned long FilePageID) {
    Page* MemPage = NULL;
    //遍历所有的内存中已经有的文件页,查找文件页是否已经在内存中载入了
    for (int i = 1; i <= MemPageCount; i++)
    {
        if (MemPages[i] && MemPages[i]->FilePointer == FilePointer &&
MemPages[i]->FilePageID == FilePageID) {
            MemPage = MemPages[i];
            break;
        }
    }
    if (MemPage != NULL)
        return MemPage;

    // 否则，从磁盘载入
    unsigned int NewPage = GetUseablePage();   //从 Clock 中找一个可以替换的页出来
    //给该页进行对应的初始化信息记录
    MemPages[NewPage]->FilePointer = FilePointer;
    MemPages[NewPage]->FilePageID = FilePageID;
```

```
    MemPages[NewPage]->IsModified = false;
    MemPages[NewPage]->IsLastUsed = true;

    int temp = fseek(FilePointer, FilePageID*PageSize, SEEK_SET);
     // 定位到将要取出的文件页的首地址
    long byte_count = fread(MemPages[NewPage]->PtrtoPageBeginAddress, PageSize, 1,
FilePointer);          // 读到内存中
    /*注意：fread 返回成功读取的对象个数，若出现错误或到达文件末尾，则可能小于 count。
        若 size 或 count 为零，则 fread 返回零且不进行其他动作。
        fread 不区分文件尾和错误，因此调用者必须用 feof 和 ferror 才能判断发生了什么。*/
        /*  if (byte_count == 0) {
            printf("fseek Error");
            return NULL;
        }*/
    return MemPages[NewPage];
}
```

6、写回在内存中的信息至文件中

```
void Page::WriteMemToFile() const {       // 把内存中的页写回到文件中
    // 只有脏页需要写回，而不是脏页的页是不需要写回的
    if (this->IsModified && this->FilePointer != NULL)
    {
        int temp = 0;
        temp = fseek(this->FilePointer, this->FilePageID*PageSize, SEEK_SET);  //从这个
文件开始，向后移文件页 * 文件页大小，就到了当前需要写回的文件页开始
        temp = fwrite(this->PtrtoPageBeginAddress, PageSize, 1, this->FilePointer); // 写
回文件
        this->IsModified = false;
        this->IsLastUsed = true;
    }
}
```

7、增加一个新页

```
Page* File::AddExtraPage()
{
    Clock *MemClock = GetGlobalClock();
    //获取文件首页,更新相关信息
    Page* FirstPage = this->GetFileFirstPage();
    this->TotalPage = FirstPage->GetFileHead()->TotalPage + 1;  //获取第一页中页头部分的
该文件的总页数+1，即为新文件的总页数
    FirstPage->GetFileHead()->TotalPage += 1; //页头中的信息也需要更新
```

```
    FirstPage->SetModified(true);
    FirstPage->SetLastUsed(true);
    //创建新内存页并且返回
    Page * NewPage = MemClock->CreateNewPage(this->FilePointer,
FirstPage->GetFileHead()->TotalPage - 1);
    NewPage->SetModified(true);
    NewPage->SetLastUsed(true);
    return NewPage;
}
```

8、记录读取函数

从相应的 FileAddress 中，读取相对应的内存中信息的地址

```
const void* File::ReadRecord(FileAddress *RecordAddress)const {
    Page* MemPage = GetGlobalClock()->GetMemAddr(this->FilePointer,
RecordAddress->FilePageID);    /
/在内存中通过文件指针和页号，找到要查询的记录的内存页的指针地址
    return (char*)(MemPage->PtrtoPageBeginAddress) + RecordAddress->OffSet;
//然后用此内存页中所记录的存数据的地址 + 页中数据偏移量  ==  某条记录的地址
}
```

9、添加记录函数

向对应的内存地址中添加一条记录数据，并且返回对应的映射文件中某一位置的地址 FileAddress，并返回新添加记录的地址

```
FileAddress File::AddRecord(const void* const Source, unsigned int SizeofRecord)
{
    Page* MemPage = GetGlobalClock()->GetMemAddr(this->FilePointer, 0);
    FileHead* Filehead = MemPage->GetFileHead();
    FileAddress fd; // AddRecord 写入的位置
    void *Temp;
    if (Filehead->DelFirst.OffSet == 0 && Filehead->DelLast.OffSet == 0)
    {
        // 没有被删除过的空余空间，直接在文件尾插入数据
        // 将添加的新地址作为记录数据的一部分写入
        Temp = malloc(sizeof(FileAddress) + SizeofRecord);
        memcpy(Temp, &Filehead->NewInsert, sizeof(FileAddress));
    //Temp == (记录着文件末尾可插入新数据的 FileAddress)的地址 + 记录的数据
        memcpy((char*)Temp + sizeof(FileAddress), Source, SizeofRecord);
        FileAddress Pos = WriteMem(Temp, SizeofRecord + sizeof(FileAddress));
    //将记录信息写入该内存文件中空余的部分,并且返回写入的物理文件中的地址信息
        WriteMem(&Pos, sizeof(FileAddress), &Pos);
        fd = Pos;                //即返回了
    }
    else if (Filehead->DelFirst == Filehead->DelLast)
    {
```

```
        // 在第一个被删除的数据处，填加新数据
        Temp = malloc(SizeofRecord + sizeof(FileAddress));
        memcpy(Temp, &Filehead->DelFirst, sizeof(FileAddress));
        memcpy((char*)Temp + sizeof(FileAddress), Source, SizeofRecord);
        WriteMem(Temp, SizeofRecord + sizeof(FileAddress), &Filehead->DelFirst);
        fd = Filehead->DelFirst;
        Filehead->DelFirst.OffSet = 0;
        Filehead->DelLast.OffSet = 0;
    }
    else
    {
        FileAddress Pos = Filehead->DelFirst;
        fd = Filehead->DelFirst;
        Filehead->DelFirst = *(FileAddress*)ReadMem(&Filehead->DelFirst);

        Temp = malloc(SizeofRecord + sizeof(FileAddress));
        memcpy(Temp, &Pos, sizeof(FileAddress));
        memcpy((char*)Temp + sizeof(FileAddress), Source, SizeofRecord);
        WriteMem(Temp, SizeofRecord + sizeof(FileAddress), &Pos);
     //将数据写入最近删除的记录处
    }
    delete Temp;
    MemPage->SetModified(true);
    return fd;
}
```

10、删除记录函数

在对应的内存地址中删除一条记录数据，并且返回对应的映射文件中某一位置的地址FileAddress，返回删除完记录后的地址。

```
FileAddress File::DeleteRecord(FileAddress *DeleteAddress, unsigned int SizeofRecord)
{
    Page* MemPage = GetGlobalClock()->GetMemAddr(this->FilePointer, 0);
    FileHead* Filehead = MemPage->GetFileHead();

    // 如果待删除数据地址的地址标识和本身地址不等，则是已经删除过的数据
    //返回的 fd 本来应该是返回的一个内存的地址
    FileAddress fd = *(FileAddress*)ReadMem(DeleteAddress);
    if (fd != *DeleteAddress)
    {
        FileAddress tmp;
        tmp.SetFileAddress(0, 0);
        return tmp;
    }
    else if (Filehead->DelFirst.OffSet == 0 && Filehead->DelLast.OffSet == 0)  // 之前没
```

```
有删除过记录
    {
        Filehead->DelFirst = Filehead->DelLast = *DeleteAddress;
        FileAddress tmp;
        tmp.SetFileAddress(0, 0);
        WriteMem(&tmp, sizeof(FileAddress), &Filehead->DelLast);
    }
    else
    {
        // 删除记录
        WriteMem(DeleteAddress, sizeof(FileAddress), &Filehead->DelLast);
        Filehead->DelLast = *DeleteAddress;
        FileAddress tmp;
        tmp.SetFileAddress(0, 0);
        WriteMem(&tmp, sizeof(FileAddress), &Filehead->DelLast);
    }

    MemPage->SetModified(true);
    return *DeleteAddress;
}
```