第五组程序说明

程序功能

基础功能

1. 登录注册
2. 查找书籍（支持多条件查询）
   a) 根据 id（唯一）
   b) 根据 ISBN
   c) 根据作者
   d) 根据出版社
3. 用户操作
   a) 借书
   b) 还书
   c) 查看自身借阅记录（曾经借还记录，未还记录）
4. 管理员操作
   a) 添加书籍
   b) 删除书籍
   c) 修改某 ISBN 书籍信息
   d) 查看在借书
   e) 根据用户 id 查看其借还记录
   f) 根据书 id 查看其被借还记录
   g) 查看管理员账号的添加书籍，删除书籍记录

其他功能

1. 配置文件（热）加载

程序架构

模仿 MVC 模式，并将程序分为：前端、控制层、服务层、数据层。共 4 个大层。每个大层可能还会有所区分。

数据层完成数据存储的功能。实现了堆文件、哈希文件两种文件结构。支持对 MetaData（元数据）的写入、删除、覆写、查找等功能。

服务层提供数据服务。负责各类数据文件的管理、数据存储的逻辑调用。并且支持将 book 类转换为 MetaData，满足控制层（对各种类操作）到数据层（对 MetaData 操作）的数据切换。

控制层根据前端指令请求各种服务。例如在借阅书籍时，需要请求修改书籍，也需要请求修改记录。

前端满足用户需求。

数据存储形式

在数据层，所有的数据都以 MetaData（元数据）形式进行组织，不做语义上的区分。MetaData 即为固定长度的字符数组（char[BUFFER_LENGTH]，BUFFER_LENGTH 是长度，可在配置文件中修改）。

因此在访问文件时，一次仅能在 index * BUFFER_LENGTH 上取出 BUFFER_LENGTH 长度的数据，其中 index 是第几个 MetaData（可理解为元数据在文件下的地址）。至于 MetaData 中存储内容的含义，与数据表的含义有关。

```
#pragma once
#define BUFFER_LENGTH 800    //MetaData长度
#define ADDRESS_LENGTH 8     //地址长度，现在是一个元数据可以存储100个地址
//book堆表
#define ISBN_LENGTH 13       //ISBN长度
#define BOOKNAME_LENGTH 100 //书名长度
#define AUTHOR_LENGTH 100    //作者长度
#define PUBLISHER_LENGTH 100//出版社长度

#define ID_LENGTH 6 //id长度，暂时没啥用

#define PASSWORD_LENGTH 256

typedef char MetaData[BUFFER_LENGTH];    //元数据
typedef int ISBN[ISBN_LENGTH];           //ISBN
typedef char BookName[BOOKNAME_LENGTH];
typedef char Author[AUTHOR_LENGTH];
typedef char Publisher[PUBLISHER_LENGTH];
typedef char Address[ADDRESS_LENGTH];    //地址
typedef int ID[ID_LENGTH];               //id，暂时没用
```

配置文件表，暂时还不支持硬盘配置文件

基础文件管理类 FileManager（构造方法、Set 方法、Get 方法省略）

成员变量	含义	其他说明
file_name_ : char*	管理的文件名称	构造时初始化
name_length : int	文件名称长度	构造时初始化
file_length_: int	元数据个数	构造时统计，其他时间计算
bit_num_ : int	字节数	没用，bit 写错了不改了
ifs_ : ifstream	读文件	
ofs_ : fstream	写文件	fstream 支持读写，但此处不进行读文件功能。主要用于打开文件时不清空内容。如果文件不存在不能创建文件
方法（部分）	参数和返回值	其他说明
GetAllData(metadatas : MetaData*, num : int&) : int	参数： metadatas：接收所有 MetaData num：传入 metadatas 的长度，接收 metadatas 有多长	得到文件中所有 MetaData，将其放入 metadatas。 metadatas 必须预先开辟内存。并将预先长度放入 num 中，如果 num 小于文件长度则只能提取 num 个数据。
WriteData(data : char*,	参数： data：写入的数据	虚函数，需要重载！ 每次写入都会清空文件。

data_length : int) : int	data_length：长度 返回值： 成功返回 0 失败返回-1	data 如果大于 MetaData，则截断，小于则补\0。
WriteData(metadata : MetaData) : int	直接写入 metadata	同上
GetData(metadata : MetaData&, index : int) : int	参数： metadata：接收数据 index：寻找地址 返回值： 成功返回 0 失败返回-1	寻找第 index 个 MetaData，并通过参数返回。 如果 index 小于 0 或大于 file_length_则 metadata 所有位被设为\0。
OpenOfs() : int	返回值： 成功返回 0 失败返回-1	打开 Ofs_，此函数为保护函数，但需要重载。正常使用无需关心。

堆表文件管理类 HeapFileManager

继承自 FileManager。

数据无序存储，存储位置不可控。但也支持强制覆写（手动指定位置）。堆表也维护一个子文件，用于记录堆表中空闲位置。

增加 MetaData 时，MetaData 被优先填充在空闲位置。如果不存在空闲位置则添加在文件尾部。

删除 MetaData 时，对应位置的 MetaData 被空 MetaData 覆盖。同时将空闲位置记录到文件。

空闲位置仅在增加数据时和删除数据时改变。强制覆写数据时不改变。如果恰好覆写到空闲位置，不视为合法覆写。(强制覆写本身就是不安全的操作)。

成员函数（部分）	参数和返回值	说明
WriteData(data : char*, data_length : int) : int	参数同父类 返回值： 成功返回下标 失败返回-1	重载后，写入数据的规则见上文。
UpDataData(index : int data : char*, data_length : int) : int	参数: index：覆写位置 其他略 返回值： 成功 0，失败-1	不安全的覆写操作。请确保使用在安全的环境。 （函数名拼写错误） 同时提供传入 MetaData 的重载。
DeleteData(index : int) : int	参数： index：删除位置 返回值： 成功 0，失败-1	将 index 处的 MetaData 覆盖为空。同时记录空闲位置。

哈希表文件管理类 HashMapManager

继承自 FileManager。设计目的用于存储地址类型数据。

存储位置可控且必须手动控制。每次存入数据仅能输入 Address。Address 即为地址类型数据，暂定长度 8 位，每个 MetaData 可存储 100 个 Address（暂定）。当存入 Address 时，会将 Address 插入对应位置的 MetaData 内。即同一个位置最多插入 100 个 Address。

每次读出数据则按照地址读取。对应地址的 MetaData 被切割成若干 Address 返回。

删除数据时需要指定删除地址和 Address。如果对应地址的 MetaData 中包含 Address 则将 Address 覆盖为空。如果不存在 Address 则不进行任何操作。如果传入的 Address 为空，则将对应地址的 MetaData 全删。

同时也支持直接读取 MetaData 和强制覆写。（暂时没用）

方法（部分）	参数和返回值	说明
WriteContents(key : int, address : Address) : int	参数： key：键 address：写入内容	见上文
DeleteContents(key : int, address : Address) : int	参数： key：键 address：删除的内容	如果 address 为空 Address 则全删。反之在 MetaData 中寻找 address 并删除。
GetContents(key : int, num : int&, addresses : Address*) : int	参数： key：键 num：返回 addresses 的个数 addresses：返回 Address 返回值： 成功 0，失败-1	得到对应 key 存储的 Address。 必须预先给 addresses 分配最大空间。其最长是 BUFFER_LENGTH / ADDRESS_LENGTH。（现在是 100 个）
AddAddress2Meta(metadata : MetaData&, address : Address) : int	参数： metadata：提供原数据，接收返回值 address：等待插入	将 address 插入到 metadata 的空闲位置。使用此函数后 metadata 会改变
SplitMeta2Address(metadata : MetaData, num : int&, addresses : Address*) : int	参数： metadata：待分割 num：addresses 个数 addresses：返回分割后 Address	将 metadata 分割成 num 个 Address。 使用前 addresses 分配最大空间。

逻辑执行形式

处理的逻辑整体分为两部分。一是针对图书进行逻辑处理。二是针对记录进行逻辑处理。二者关系不大，因此分开讨论。

图书处理

图书处理再次分为三个部分：图书查询、针对用户的借书还书、针对管理员的增删改。

三个部分复杂度依次增大。但总体思想都为通过用户输入来对数据库进行操作。详细来讲，当图书查询时，用户输入查询条件，程序根据查询条件生成一本书（不真实的书），然后在数据库中寻找满足条件的若干本书（真实的书）；当用户借还时，会将“真书”从数据库中提走或存入；当管理员增删改时，也会对数据库中“真书”进行操作。

上文中“真书”中“真实”的意思即为**这本书具有 ID 编号**。用户在操作时，如果一些图书的 ISBN 相同，则用户即可认为这些书是同一本书。但在数据库中，每一本书即使 ISBN 相同，ID 也是不同的。用户的任何操作都是对“虚书”进行，进而控制数据库中的“真书”。**因此控制层在于组织用户输入，并生成“虚书”。服务层的作用在于“虚书”和“真书”的互相转换。**

对书而言，分为 3 类。基本书类 BasicBook，抽象书类 AbstractBook，实体书类 ObjBook。

基本书类拥有一本书的所有信息：ISBN、书名、作者、出版社。

抽象书类继承自基本书，额外拥有统计信息：总量 total_，可借量 enable_。统计信息为统计实体书所得。只有存在实体书才能生成抽象书。

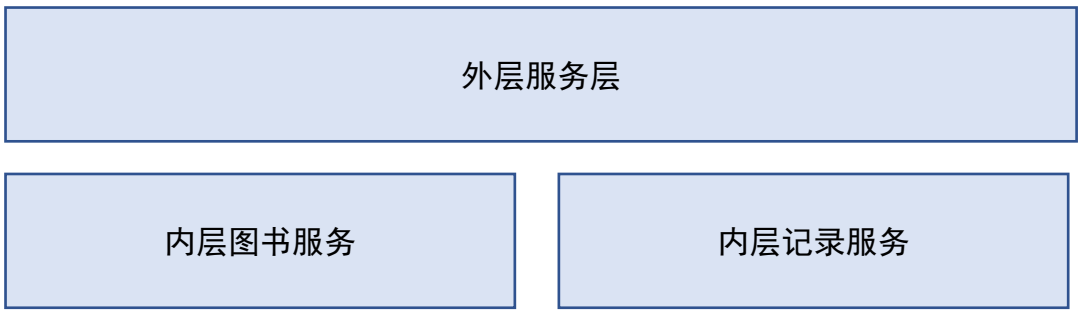
实体书继承自基本书，额外拥有 id 号。

书的 ID

其中书的 ID 号需要特别讨论。在本程序中，书的 ID 号是自动生成的。生成方法是当书被添加进入数据库后占据了一个地址。因此将地址作为书的 ID 号。所以 MetaData 中不存储 ID，MetaData 的地址就是 ID。（利弊需要讨论，暂时所有功能不需要 MetaData 存储 ID。图书分 ID 是后来加上的，因此没有更改老代码。个人认为 ID 本身就是逻辑的产物，因此使用逻辑存储 ID 即可无需使用实体内存存储 ID）。

服务层

因服务层涉及抽象书和实体书间的转换，且服务层不仅仅涉及图书服务还涉及记录服务。因此将服务层再次分为**内层**和**外层**。内层使用面向对象逻辑实现，专注于从数据库中提取实体书和写入实体书。外层使用面向过程逻辑实现，专注于暂存实体书并统计实体书形成抽象书，同时外层也同时涉及图书服务和记录服务。



内层服务层

内层服务层负责管理文件。

内层图书服务维护文件如下：

1. 图书堆表 book_heap。用于存储 BasicBook 内容，其中下标即为 objBook 的 id。
2. ISBN 哈希表 isbn_hash。用于存储 ISBN 到堆表的映射。key 是 ISBN 哈希运算，value 是 book_heap 内容的地址。
3. Name 哈希表 name_hash。用于存储图书名称到堆表的映射。

4. Author 哈希表 author_hash。用于存储作者名称到堆表的映射。
5. 被借哈希表 borrowed_hash。用于存储某本书被某人借。key 是书的 ID，value 是人的 ID。理论每本书仅能被 1 个人借。
6. 借书哈希表 borrow_hash。用于存储某个人借了哪些书。key 是人的 ID，value 是书的 ID。理论一个人可以借多本书。

具体操作

查询书

1. 用户在前端输入 ISBN、作者、出版社信息。
2. 控制层生成基本书类，传入外层服务层。
3. 外层服务层将基本书类传入内层。
4. 内层解析基本书，并生成 3 个 key。
5. 使用 key 到对应 hash 表中找到 k 个 Address。
6. 使用 n 个 Address 在 heap 表中提取出 MetaData。
7. 每个 MetaData 和对应 Address 构成实体书。
8. 内层服务层使用每个 key 都获得了 n 个实体书。n1+n2+n3 个实体书借助 id 取交集获得 m 个实体书。(如果 m 是 0，说明没找到)。
9. m 个实体书可能有的被借了，提取 m 个 ID 去被借出 hash 表中进行筛选。筛选时做标记。
10. 外层服务层按照前端输入的条件筛选一遍 m 个实体书（可能存在哈希碰撞的情况，或者其他错误，正常情况下此步骤完毕后还是 m 本)。
11. 外层服务层根据 ISBN 统计，进而生成 p 类抽象书。并返回控制层。
12. 控制层保存抽象书，前端显示。

借阅

1. 搜素之后，控制层保存抽象书。前端选择抽象书后，控制层检测选择是否合法，然后将抽象书对应的实体书的 ID 传递给外层服务层。
2. 外层服务层将 ID 传递给内层服务层（记录操作略)。
3. 内层服务层在被借出 hash 表的 ID 位置添加操作用户 ID（视为已借出)，在借出 hash 表的用户 ID 位置添加该书的 ID（视为已借出)。

查询用户已借书

1. 控制层将用户 ID 传递外层服务层。
2. 外层服务层将用户 ID 传递给内层。
3. 内层使用用户 ID 在借出 hash 表中，查询到 n 本已借的实体书。传递给外层。
4. 外层将 n 本实体书，根据 ISBN 统计，生成 m 本抽象书。并返回控制层。
5. 控制层保存抽象书，前端显示。

还书

1. 查询之后，控制层保存抽象书。前端选择抽象书后，控制层检测是否合法，然后将抽象书对应的实体书的 ID 传递给外层服务层。
2. 外层服务层将 ID 传递给内层服务层。
3. 内层服务层在被借出 hash 表的 ID 位置清空（一本书就一个人借)，在借出 hash 表的用户 ID 位置删除该书 ID。

直接添加

1. 管理员在前端输入 ISBN、书名、作者、出版社信息。
2. 控制层生成基本书。传入外层服务层。
3. 外层服务层将基本书传入内层。
4. 内层将基本书转化为 MetaData，存入堆表。获得地址 index。
5. 地址 index 转换为 Address 类型。
6. 内层解析基本书，生成 3 个 key。
7. 针对每一个 key，将 Address 插入到 hash 表。

查询添加

1. 查询之后，控制层保存抽象书。前端选择抽象书后，控制层检测是否合法。合法则控制层生成基本书。传入外层服务层。
2. 上面过程从 3 步骤开始即可。

查询删除

1. 查询之后，控制层保存抽象书。前端选择抽象书后，控制层检测是否合法（只能删除没被借出去的)。然后将抽象书对应的实体书传递给外层服务层。
2. 外层服务层将实体书传递给内层服务层。
3. 内层服务层借助 ID 删除堆表数据。
4. 将 ID 转换为 Address 类型数据。
5. 实体书解析成 3 个 key，每个哈希表 key 的位置删除 Address。

查询修改

1. 查询之后，控制层保存抽象书。前端选择抽象书。
2. 用户输入新数据，控制层将抽象书的数据替换成为新输入的。然后将抽象书传入外层服务层。
3. 外层服务层将抽象书传递到内层服务层。
4. 内层服务层将抽象书转换为 MetaData。
5. 提取抽象书 ID。
6. 将 MetaData 覆写到 ID 对应的位置。

MVC 优势

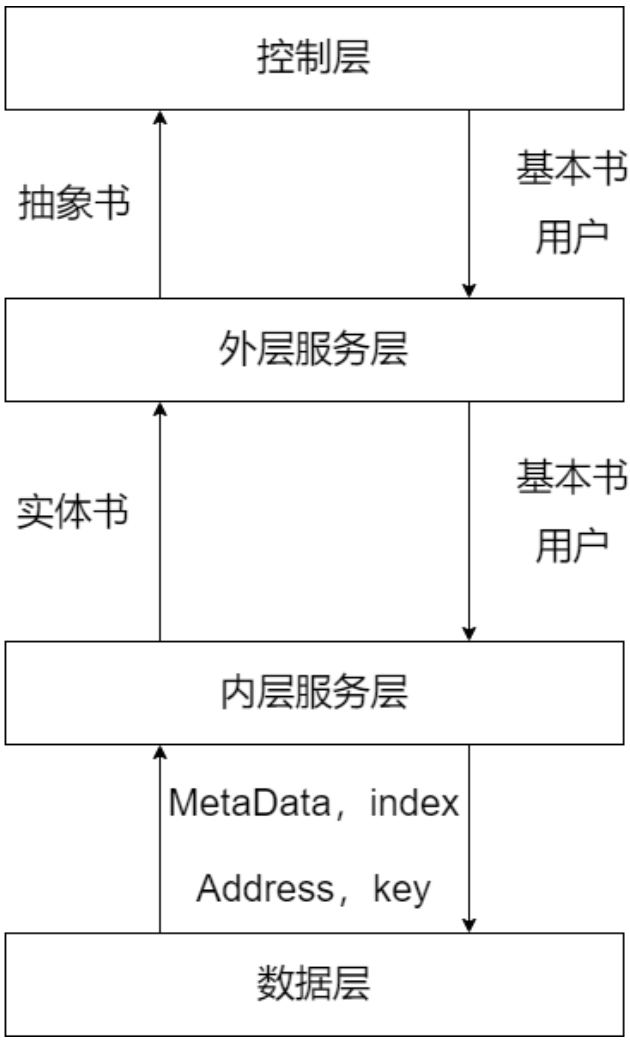
每一层确保数据的统一。

控制层从前端接收字符串，转换为基本书传递到服务层。

外层服务层将基本书转换为实体书，传递给内层服务层。

内层服务层将实体书转换为 MetaData 或 Address。可能使用实体书提供的 ID 修改数据库。或者修改数据库后使用数据库返回的 ID 改变实体书。

有时可能不需要实体书，仅仅只需要 ID 号。但是这样也可以生成一个只有 ID 的实体书进行 ID 传递。提高了多态性。


控制层
抽象书
基本书
用户
外层服务层
实体书
基本书
用户
内层服务层
MetaData，index
Address，key
数据层