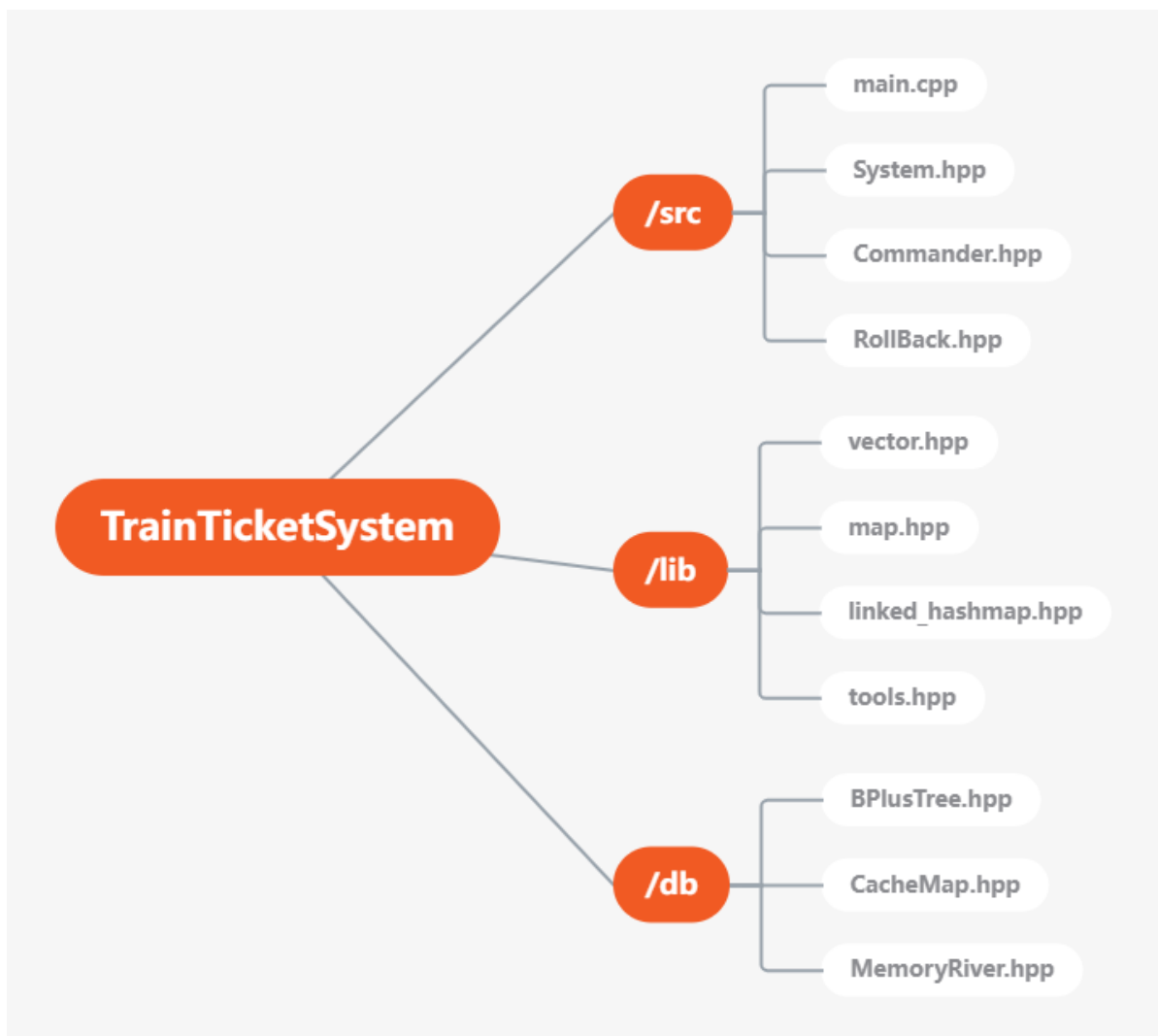


ACM TrainTicketSystem 开发文档

小组成员	组内分工
尹良升(@hnyls2002)	程序后端逻辑部分
董琿 (@RabbitCabbage)	文件存储和B+树主体

整体架构

模块划分图



火车票系统模块 /src

- main.cpp 最终的工作程序
- System.hpp 火车票系统主体
- Commander.hpp 命令行解析库
- RollBack.hpp 回滚类

工具库模块 /lib

- `vector.hpp`, `map.hpp`, `linked_hashmap.hpp`
- `tools.hpp` 自己写的一些工具

数据库内存管理模块 /db

- `BPlusTree.hpp` B+树的主体部分
- `CacheMap.hpp` 缓存类
- `MemoryRiver.hpp` 文件读写接口

设计思路

逻辑主体

- 定长的信息直接用B+树单键值索引的方式，存在外存中。
- 不定长的信息，例如**每个用户的所有订单**，**每一个站的所有经过车次**等，使用二元组来存储在B+树中，形如 `std::pair<class A, class B>` 的形式。查找时可以先固定第一维的大小，这样可以实现所有第一维相同元素的遍历。
- 获取键值为一段连续区间的信息，采用B+树内置的迭代器完成。
- 每一条指令均由 `Commander.hpp` 解析后返回一个标准化指令类，该指令类经过 `system.hpp` 的函数处理后配送到不同的操作函数去。

回滚策略

- 对于三种操作，**插入**的逆操作为**删除**，**删除**的逆操作为**插入**，**修改**的逆操作为**修改**。
- 于是对于每一颗 `bptree`，开一个外存的栈来实现逆操作序列的存储。
- 外存的栈同样采用分块的形式，对象中自带一个大小约为 4096 的块进行缓存。

缓存策略

- 为加快缓存的速度，缓存使用hash table来实现，并且采用闭散列表，防止内存溢出
- 有一个valid位，表示这个hash值是否被命中，当我们访问一个元素标记他被命中
- 有一个index位，表示这个hash值对应的键值是什么，如果再次访问时hash值键值都相同才能说明访问的是同一条记录
- 有一个dirty位表示的是这个hash值所对应的元素是否被修改，如果再次访问的时候已修改并且发生冲突就会写穿到外存
- 发生修改的元素使用链表连接，以加快 `bptree` 中执行 `lower_bound` 操作或者程序结束时外存的更新，遍历链表执行 `flush` 操作。

空间回收

- 在bpt中实现了空间回收，在另外的一个文件中记录下要回收的块的位置和数目
- 其实相当于是一个空位的栈结构
- 每一次要写一个新的块的时候优先到这个文件中找可以写入的回收位置

类和成员的设计

逻辑部分主要由 hny1s2002 负责，为了让各个功能更加清晰，此处的类与成员的设计均根据功能展开。

火车票系统主体 /src/System.hpp

核心类

- `class System`

整个火车票系统的**核心类**，所有的操作都在此类中完成。

用户信息的维护

- `struct UserInfo`

用户信息类，储存每一个用户的**基本信息**。

- `bptree UserDb (UserName -> UserInfo)`

用于储存用户的信息的**数据库**，添加和删除用户信息在 `bptree` 上操作。

用户的登录状态

- `struct LoggedUser (UserId -> int)`

一个由**平衡树**或者**哈希表**来维护的目前登录的用户，里面只存 `UserID` 和 `privilege`。

车次信息的维护

- `struct BasicTrainInfo`

每一列车次的**最为基本的信息**（不包括所有停靠的车站名），由于**除了站名之外的所有信息**都会频繁地被访问，而所有停靠站的站名**空间占用十分庞大**，所以这样的设计可以尽可能地减少一个对象的大小，从而增大一个数据块的数据个数，**减少树高**。

- `bptree BasicTrainDb (TrainID -> BasicTrainInfo)`

用于储存列车的信息的**数据库**。

- `class TrainInfo`、`bptree TrainDb`

主要用于储存一辆列车**经过的所有车站**，可以发现只有在 `query_train` 操作中才会要连续地访问这些车站，于是整个 `TrainDb` 只会在 `query_train` 中被用到。

车票查询&换乘功能

需要对于每一个车站，查找所有经过该站的车次，于是用 `bptree` 维护一个**一对多**的数据类型。

- `struct StInfo`、`bptree StDb`

即一个站对应的列车信息，不难发现只需要简单维护**这个站在这辆车上的序号**，**票价前缀和**，**时间前缀和**等几个简单的数据即可。

当日车次状态查询

- `struct DayTrain`

每一天的列车由于售票情况不同因此需要一个**单独的对象**来表示。

- `struct DayTrainDataBase ({Train,day} -> DayTrain)`

用于储存某一天某一列车信息的**数据库**。

查询订单功能

- `struct Order`

用于保存一份订单的**基本信息**，每一位用户的**订单记录**也是使用B+树的**一对多**类型数据。

候补列表查询

- `struct Pender`

候补的用户为一个**队列结构**，同样采用**一对多**类型存储在B+树种，储存的信息类为 `Pender`。

自定义工具类 `/lib/algorithm.hpp`

- `struct Date`

用于存储火车的日期。

- `struct Time`

用于存储火车的时间。

- `struct fstr<len>`

固定长度字符串

函数接口（逻辑部分）

火车票系统主体 `/src/System.hpp`

通过 `Opt` 来调用相应的操作函数，所有的返回值均用 `vector<string>` 的形式来存储，输出的时候再进行转换，函数的 `arguments` 类型待定。

```
vector<std::string>Add_User
vector<std::string>Login
vector<std::string>Logout
vector<std::string>Query_Profile
vector<std::string>Modify_Profile
vector<std::string>Add_Train
vector<std::string>Release_Train
vector<std::string>Query_Train
vector<std::string>Query_Ticket
vector<std::string>Query_Transfer
vector<std::string>Buy_Ticket
vector<std::string>Query_Order
vector<std::string>Refund_Ticket

std::string Opt(const CmdType & arg);
```

命令行解析库 `/src/Commander.hpp`

传入由 `main.cpp` 读取的一列字符串，将其发送给解析类，解析类返回一个标准命令类。

```
CmdType Parser(const std::string &arg);
```

B+ Tree

文件读写部分由董坤(@RabbitCabbage)负责

设计用B+树+链表的储存结构，分别存储需要支持随机访问和顺序访问的数据库
先封装一个MemoryRiver类，进行底层的文件读写操作

```
template<typename T>
class MemoryRiver {
private:
    std::fstream file;
    char file_name[20];
public:
    MemoryRiver(char *file_name);

    ~MemoryRiver();

    void write(const T &t, const int index);

    T Read(const int index);
};
```

用数据结构B+树实现信息的存储，用户信息和列车信息都以B+树的形式存储到外存
同时实现信息的缓存以减少外存读取次数，最终封装为CacheMap的public函数，可以直接调用
BPlusTree提供的函数接口如下：（具体实现时会有其他某些private工具函数），并且是写给Cache调用的

```
template<typename Key, typename Info, const int max_key_num = 5, const int
max_rcd_num = 5, class KeyCompare = std::less<Key>>

class BPlusTree {
public:
    struct Node {
        int location = -1;
        bool isleaf = true; // 标记是不是叶节点
        int children_num = 0;
        int children[max_key_num];
        Key keys[max_key_num];
    };

    struct Block {
        int next = -1, prior = -1;
        int location = -1;
        int size = 0;
        Key keys[max_rcd_num];
        Info record[max_rcd_num];
    };

    class iterator {
    private:
        Key key;
        Info info;
        BPlusTree<Key, Info, max_key_num, max_rcd_num, KeyCompare> *this_bpt =
        nullptr;
```

```

    Block block;//所在的block
    int block_index;//在block中的标号
    bool at_end = false;
public:
    iterator operator++(int x)
};

```

```

public:

```

```

    //B+树的构造函数,由一个文件构造
    BPlusTree(const char *file_name, int cache_size) {}

```

```

    //B+树的析构函数
    ~BPlusTree() {}

```

```

    //插入一个元素, 参数是插入元素的键值和记录的详细信息, 返回插入是否成功
    //如果说这个元素本来存在, 插入失败返回false

```

```

    bool Insert(const Key &key, const Info &info) {}

```

```

    //删除一个元素, 参数是要删除元素的键值, 返回是否删除成功,
    //如果这个元素在B+树中不存在就删除失败

```

```

    bool Erase(const Key &key) {}

```

```

    //查询一个元素, 参数是要查询元素的键值
    //返回值是一个pair, bool代表有没有找到
    //如果找到这个元素存在, 返回true, 同时返回记录信息的具体值
    //如果没有找到, 返回false, 这时的返回struct是随机值, 不应该被访问
    std::pair<bool, Info> Find(const Key &key) {}

```

```

    //修改一个元素, 参数是要修改元素的键值和修改之后的信息
    //返回一个bool, 修改成功返回true, 否则返回false
    //如果这个要修改的元素在B+树中不存在就会返回false
    bool Modify(const Key &key, const Info &new_info) {}

```

```

    //清空B+树的有关文件
    void clear() {}

```

```

    //返回现在总共有多少条记录
    int GetSize() {}

```

```

private:

```

```

    //这里是一些工具函数

```

```

    //一类是经常使用的查找函数, 封装成一个工具

```

```

    int BinarySearchLess(Key array[], int size, const Key &key)

```

```

    int BinarySearch(const Block &block, const Key &key)

```

```

    //还有这一类是在插入删除时经常用到的处理儿子的分裂和合并的函数

```

```

    //提高代码复用率

```

```

    void Split(Node &cur, Node &new_node, int &new_node_loc, int
    &splited_child_index, int num)

```

```

    void AddChild(Node &cur, const Key &first_key, const Key &added_first_key,
    int added_loc, int num)

```

```

    void MergeBrothers(Node &front, Node rear)

```

```

    void MergeBlocks(Block &front, Block &rear, int front_loc)

```

```

bool BorrowChild(Node &cur, int num, Node &parent, int
&brother_to_merge_index, Node &brother_to_merge)

std::pair<std::pair<int, int>, Info> RecursionFind(const Key &key)

//这个是插入的私有函数，递归实现
//它是向以cur作为根节点的子树进行插入，返回值表示这个节点是否发生了分裂
//在发生分裂时处理它的父节点，并且相同的方法向上回溯，直到根，特殊讨论
bool RecursionInsert(Node &cur, const Key &key, const Info &info, bool
&success, int &splited_child_index, Node &new_node, int &new_node_loc)

//这个是删除的私有函数，递归实现
//它是向以cur作为根节点的子树进行搜索删除，返回值表示这个节点是否需要兄弟节点的支援
//如果返回true就处理它的父节点，借一个孩子或者发生合并
bool RecursionRemove(Node &cur, const Key &key, bool &success, Node parent)
};

```

文件存储 Cache Map

Cache的实现用HashMap，用双hash解决冲突（这里默认键值应该是字符串，提供两个hash函数）

1. 记录一个bool类型的Valid数组，表示的是这里的键值是否有效，在每次程序运行时全部初始化为0
2. 记录另外一个hash的值作为Index，用来检查第一个hash值是否发生了碰撞
3. 记录一个dirty数组，表示的是这个键值对应的信息已经被修改，如果hash发生了碰撞，这里的信息要更新到外存，初始化的时候也全初始化为0；
4. 再记录一个数据集，表示的是命中之后的记录信息

当我们需要修改某个键值对应的信息时，从外存读取原始数据，修改了之后记录到Cache的数据记录中去，将对应的Valid值记为1，表示这里的信息被命中过；并且把dirty记录为1，表示的是这里的信息已经被修改，和外存中不同；当再修改一个键值对应的信息时，如果说Valid已经是1，说明这个hash已经被命中，要检查index是否相同，如果相同，直接修改，dirty记为1；如果不同说明发生了碰撞，如果说之前命中的这个元素的dirty是0，那么这个元素没有修改，直接把新元素放入cache；如果说dirty是1，就要把修改之后的信息写入外存，再放新元素；

当我们要插入一个新的元素，就把信息同时写入外存和cache，并且标注valid=1，dirty=0，相当于在cache中加入了一个被命中而没有被修改过的元素

当CacheMap的size达到上限时，以及程序结束需要析构时，就应该把它清空，调用B+Tree的Modify写入外存

```

template<typename Key, typename Info, const int max_size = 17, const int
max_key_num = 5, const int max_rcd_num = 5, class Hash=std::hash<Key>, class
KeyCompare = std::less<Key>>

class CacheMap {
private:
    struct node {
        int dirty_index = -1; //这个dirty位是哪一个
        node *nxt = nullptr; //记录的是，下一个dirty位是哪一个
        node *bef = nullptr;

        node(int ind = -1, node *n = nullptr, node *b = nullptr) {
            dirty_index = ind, nxt = n;
        }
    };

    bool valid[max_size];

```

```

    bool dirty[max_size];
    node *prev[max_size];
    node *head = nullptr, *rear = nullptr; //维护一个串起所有dirty的链表，每增加一个
dirty就在链表尾部加一个节点
    Key index[max_size]; //记下这时的这个hash所对应的是什么键值
    Info information[max_size];
    KeyCompare cmp;
    Hash hash_func;
public:
    CacheMap() {}
    ~CacheMap() {}
    //插入一个元素，参数是插入元素的键值和记录的详细信息，返回插入是否成功
    //如果说这个元素本来存在，插入失败返回false
    //直接操作到B+Tree，并且存入Cache
    bool Insert(const Key &key, const Info &info) {}

    //删除一个元素，参数是要删除元素的键值，返回是否删除成功，
    //如果这个元素在B+树中不存在就删除失败
    //直接操作到B+tree上
    bool Erase(const Key &key) {}

    //查询一个元素，参数是要查询元素的键值
    //返回值是一个pair，bool代表有没有找到
    //如果找到这个元素存在，返回true，同时返回记录信息的具体值
    //如果没有找到，返回false，这时的返回struct是随机值，不应该被访问
    //访问之后就要存入Cache，便于下次访问
    std::pair<bool, Info> Find(const Key &key) {}

    //修改一个元素，参数是要修改元素的键值和修改之后的信息
    //返回一个bool，修改成功返回true，否则返回false
    //如果这个要修改的元素在B+树中不存在就会返回false
    //修改的时候要进行检查valid,index,dirty的检查
    bool Modify(const Key &key, const Info &new_info) {}

    //清空Cache，把它全部都写到外存
    void clear() {}

    //返回现在总共有多少条记录
    int GetSize() {}
};

```