Project 设计与测试文档

F1203704 5120379091 高策

目录

第一章 设计过程

1.1 设计构想

Project 的基本要求是实现一个社交系统，使其能够进行基本的消息发布，信息共享，好友关注等功能。针对后台的操作，希望设计针对用户名的开放链表法的哈希索引，消息则使用链表实现，关注关系以有向图的十字链表存储。至于前台的设计，希望以一个界面类，实现全部的前台指令操作。在设计的时候，应当以功能为导向，为界面类提供应有的接口，以保证代码的正常运作。

文件存储的设想：

1. 尽可能减少内存的使用量，使用文件操作
2. 修改用户信息不需要修改索引值，以达到代码量的减少
3. 多索引，保证条件搜索的速度
4. 可变信息与长度不可变的信息分开存储，以实现叠加式地文件内容增长，而非重写

数据结构的设想：

1. 尽可能实现不修改索引的情况下实现数据文件的修改
2. 尽可能选用合适的数据结构，不同情况下使用的数据结构考虑实现为模板方式

对于程序的设想就是这样，在写程序的时候也一直尽量以之作为纲领。

1.2 文件存储

因为要保证用户的信息在程序结束后不消失，所以需要将其保存在文件中。我将用户的所有内容分为三类：长度不变的个人信息（如电话，等等），用户发布的信息，以及关注信息。所有的文件都是以二进制存储。用户的所有关于个人的信息都存储在 info 中，用户发布的，消息在 meg 中，而关注在 like 中。他们的组织形式如 Figure 1 所示。

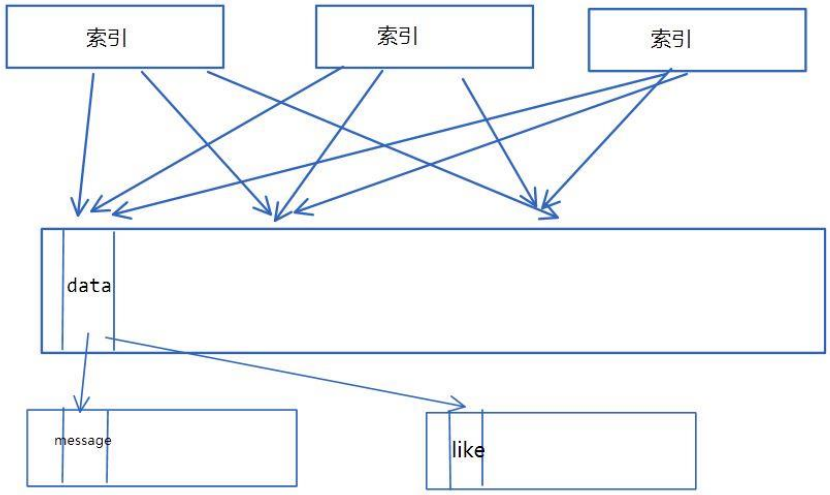

Figure 1 索引与数据的关系图

还有关于索引的存储，由于程序要求实现关于个人信息的各个条目的搜索，所以关于每一种信息的索引都单独存放。index 中存放的是用户名哈希索引，index_name 存放的是用户姓名索引，index_bir 是生日索引，index_tele 是电话索引，index_add 是地址索引。

1.3 前台设计

程序的前台功能都是在 Interface 类中实现的，Interface 同时也是连结前台后台的主要类。同时为了测试模式，Interface 也提供了测试接口。详细接口如图 Figure 2，具体见..\P2Project\interface.h。

```
class Interface
{
public:
    Interface(bool saveflag);
    ~Interface();
    void beginTowork();

    //for test
    void load();            //load before the program
    void login();           //login
    bool login(Information &info);
    void pushMessage(Message meg);
    void register_t(Information &newuser);
    void quitDo_offline();  //save after the quit
    void likeSomeone(char *buf);        //like someone
    void changeName();
    void cancelLike(char *x);
    void deleteMessage(int number);
private:
    bool quitFlag,logFlag,saveflag;
    string cmd_off, cmd_on;
    HashTable<RecordNode, SaveForm> *hash;
    HashTable<RecordNode_name, SaveForm_name> *hash_name;
    HashTable<RecordNode_bir, SaveForm_bir> *hash_bir;
    HashTable<RecordNode_tele, SaveForm_tele> *hash_tele;
    HashTable<RecordNode_add, SaveForm_add> *hash_add;
    OLGraph *relationship;
    vector<Information> recommandPeople;
    User user;

    void getInput_offline();
    void getInput_online();
    void usage_offline();
    void usage_online();
    void commandExec_offline();
    void commandExec_online();
    void display_offline();
    void display_online();

    void register_m();      //register
    void deleteUser();
    void renewUser();

    void changeInfo();      //change something
    void searchSomeone();   //search
    void multipleSearch();  //multiple search
    void likeSomeone();     //like
    void cancelLike();
    void people();          //watch who like me and I like who
    void recommand();

    void watchSomeone();
    void shareMessage();    //share messages
    void pushMessage();     //push messages
    void showMessages();    //show the messages
    void showMyMessages();
    void showAllMessages();
    void deleteMessage();

    void onlineWork();      //online
    void work();

    void printInfo(Information &x);
    void clear();
};

#endif
```

Figure 2 前台接口操作

1.4 数据结构

1.4.1 前台-字典树

本来是打算实现前缀搜索的功能的，后因时间紧迫，无奈放弃。不过在字典序输出关注列表时可以用到字典树，因此就保留了此数据结构。

在整个程序中，只有在前台显示列表的时候使用到了该数据结构，所以用处不是很大，就作为一个尝试吧。

1.4.2 后台-哈希表（模板实现）

在程序的后台操作中，出于各种考虑，最终选择了链表寻址式哈希表作为索引的数据结构。主要的信息都是以字符串的方式存储，因此哈希的函数是针对字符串设计的，采用了 BKDRHash 方法，哈希表共有 H 项，H 为宏定义，大小可改动，解决冲突的方法是拓展链表法。考虑到系统要求的查找操作不只针对用户名，因此哈希表以模板类的方式实现，接口如 Figure 3 所示，详细接口见..\P2Project\hash.h。

```
template <class T, class S>
class HashTable
{
public:
    HashTable(char *x, int size = H);
    ~HashTable();
    T contains(char *x);
    S containsFrom(char *x, S &t);
    void makeEmpty();
    bool insert(T &x);
    void deleteNode(T &x);
    void begin();
    void dump();
private:
    int myhash(T &x);
    int myhash(char *x);
    int currentSize;
    char *fileName;
};
```

Figure 3 索引接口

数据结构在第一次启动时，会建立哈希表项的空内容，之后会在索引文件中插入新的内容，在加入元素的时候会将其保存至硬盘中，以保证内存与硬盘内容的统一。在程序运行的过程中，数据结构始终不会被读入内存，索引项的寻址等操作大部分都是建立在文件操作的基础上。由于在实现的设想中，修改都是在数据文件中进行，而哈希索引只是存放指向数据文件的指针，因此是不会被修改，因此没有实现关于索引修改的接口，而对于信息修改导致的修改，采取的是先删除再插入的方法。

```
struct RecordNode
{
    char username[LENGTHOFID];
    int pointer;
    bool operator==(const RecordNode& x)
    {
        return  this->username == x.username;
    }
};

struct SaveForm
{
    RecordNode record;
    long pointerToNext;
};
```

Figure 4 索引的存储格式

索引的格式如下所示，key 值是索引值，而 pointer 则指向索引值所在文件的位置，是一个文件指针。只要知道索引的键值，可以将其在文件中的位置找到，如果保证索引与数据的同步性，就可以大大提高遍历的效率，这也是本次 project 最有难度的地方吧。

Key1	Pointer1
Key2	Pointer2
…	…

至于存储格式的设置，首先索引是有其键值以及指向 info 中对应信息所在位置的指针，而存储还需要保存一个指向下一个哈希索引节点的指针。即如下所示，Saveform 是由索引和指针构成，指针指向链表的下一个节点，依靠此建立了哈希链表。

RecordNode1	PointerToNext1
RecordNode2	PointerToNext2
…	…

在每次插入新索引时，原本链表的最后一个节点会将 PointerToNext 指向新建立的索引结点，从而形成新的索引。这样就完完全全在文件中构建起了一个链表拓展法的哈希索引，提高了搜索的效率。

1.4.3 后台-有向图十字链表

程序要求可以查看用户的关注以及关注用户的用户，因此选用有向图作为存储关注关系的数据结构，而十字链表是比较好的存储方式，因为其可以以链表的方式同时查看用户的关注以及关注用户的用户。该有向图也是常驻硬盘的。接口如图 Figure 5 所示，详细见..\P2Project\graph.h。

```
typedef struct OLGArc
{
    long tailvex;
    long headvex;
    long hlink;
    long tlink;
}OLGArc;

typedef struct VexNode
{
    char username[LENGTHOFID];
    long firstin;
    long firstout;
}OLGVNode;

#define MAX_VERTEX_NUM 100000

class OLGraph
{
private:
    long pointer;
    void DeleteArc(long po1, long pointer);
public:
    friend class Interface;
    OLGraph();
    void load();
    void dump();
    long addVex(char *x);
    bool contains(long pointer);
    void DeleteVex();
    void InsertArc(long pointer);
    void DeleteArc(long pointer);
    void setPo(long pointer);
};

#endif
```

Figure 5 有向图接口

所有关于关注关系的操作全部建立在 OLGraph 的基础上进行。至于关注关系的结构，由用户结点和关注弧构成。对于关注的文件存储，比较难的地方在于结点与弧都是增长的，因此保存起来不方便。最开始的想法是结点与弧保存在两个文件中，后来为了整齐考虑，最后把两个保存在一个文件 like 中。由于每个用户只需要知道与自己相关的关注关系，因此用户的信息里有一个指向 like 中该用户的用户结点，这样不需担心结点与弧会找不到位置，就可以实现混合存储了。

虽说理论上可以实现结点与弧混合存储，但是具体实现还是比较麻烦。首先，结点的存储格式如下，第一个元素是唯一的用户名，接下来就是十字链表的两根链表的头项，分别是关注该用户名的与该用户名关注的。

Username	Firstin	Firstout
…	…	…

至于弧的存储方式，如下所示，四个元素都是 long 变量，前面两个是二元关系的两个变量，代表 tailvex->headvex，而两个变量是指向有向图结点的指针。接下来的两个变量是指向关系链表中的下一个元素的指针，即指向下一个弧的指针。

Tailvex	Headvex	Headlink	Taillink
…	…	…	…

而由于两者都存储在一个文件中，所以是混合存储，用户数据中有一个指向其结点的指针，这样就可以由用户找到对应的结点，从而找到对应的关注关系。所以整个有向图的存储如下所示。

Username	Firstin	Firstout	
Tailvex	Headvex	Headlink	Taillink
Tailvex	Headvex	Headlink	Taillink
Tailvex	Headvex	Headlink	Taillink
Tailvex	Headvex	Headlink	Taillink
Username	Firstin	Firstout	
…	…	…	…

1.4.4 后台-双向链表

最初，用户发送的信息的方式是以双向链表时间顺序的方式存放的，由于数据结构较简单，没有特意写为一个类。

最终，考虑到对发送信息的响应的要求，所有信息的文件操作都是在退出登录的时候进行的，因此保存较复杂，且在程序退出前消息要存放在内存中，大大增加了内存的使用量。因此在 ver8 之后，我改变了存储方式，改为按照时间倒序，单向链表即时存储的方式，这样无论是在效率还是在内存使用量上都比前一种方式更加好。最终消息的存储如下，首先是 message，然后会有一个用来存储 username 的变量，以及一个存储时间的变量，这些是会用来显示在屏幕上的存储内容，剩下的是为了组织数据而存在的，包括指向链表中下一个元素的文件指针，指向上一个分享来源的文件指针，指向分享源头的文件指针，以及分享 flag 和分享次数纪录。这样类似十字链表的存储方式可以满足程序对信息发布以及分享方面的要求。

Message	Username	localTime	pointerToNext	pointerToS	pointerTo0	isShared	sharedTimes
Message	Username	localTime	pointerToNext	pointerToS	pointerTo0	isShared	sharedTimes
…	…	…	…	…	…	…	…

第二章　功能实现以及测试

程序的所有测试均在..\P2Project\program.cpp 实现，在进入程序的时候输入 0 进入测试模式，大部分测试均是在测试模式下进行，还有一些测试比如哈希冲突，是在 UI mode 下手动输入比对完成。

2.1 功能实现

要求功能：

2.1 用户： 全部实现
2.2 基本个人信息： 全部实现
2.3 附加个人信息： 全部实现
2.4 修改密码： 全部实现
2.5 修改个人信息： 全部实现
2.6 删除用户: 全部实现
2.7 关注： 全部实现
2.8 用户查找： 全部实现（包括多条件查找）
2.9 关注推荐： 全部实现
2.10 关注列表： 全部实现
2.11 取消关注： 全部实现
2.12 消息： 全部实现
2.14 发布消息： 全部实现（包括汉字）
2.15 消息列表： 全部实现
2.16 用户消息： 实现（在好友列表中选择用户暂未实现）
2.17 转发消息： 全部实现（包括评论）
2.18 折叠消息： 全部实现
2.19 转发计数： 全部实现
2.20 删除消息： 全部实现

自定义功能：

用户恢复
用户数记录
用户关注与被关注列表时间排序

2.2 用户

用户的基本信息，附加信息，修改密码以及个人信息，删除用户功能均已实现，即 2.1 到 2.6 的功能。

由于选用的数据结构是哈希表，因此预测程序在用户名注册环节，如果仅仅是建立用户名哈希索引，那么其时间与注册数的关系应该是线性增长的。

关于正确性的测试，每增长 1 万用户的时候，程序会接受一个输入的用户名，接着登录测试其正确性，测试输入 10 次，通过十次。

效率测试如 Figure 6，数据见 Statistic_register.txt，数据每隔 100 个用户取一次点，使用 excel 作图。由拟合直线可以看出，时间是随着注册用户数线性增长的，符合哈希表的特性。说明哈希表的查找，与插入都是在常数时间内完成的，是一个非常有效率的索引结构。

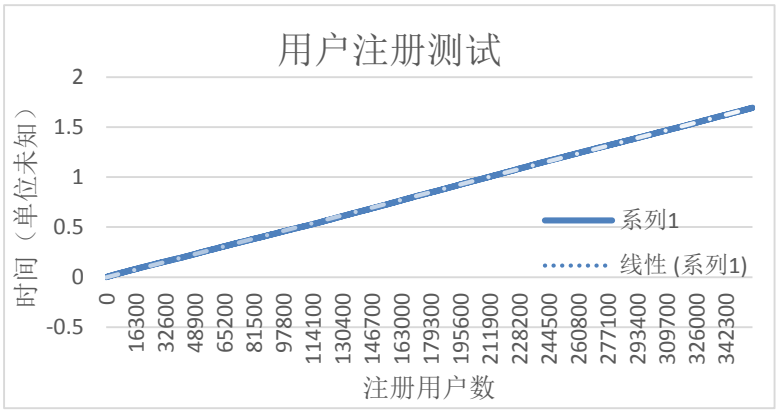

Figure 6 用户注册效率测试

至于检查哈希冲突，我将哈希的表项缩小至 100，即每 100 个元素起码有一次哈希冲突，正确性仍无误，时间开销较上面差距较大，尤其是在数据量大的时候。

2.3 关注

关注，用户查找，关注推荐，添加关注，关注列表，取消关注，均已实现。关注的正确性测试，是根据关注的用户名进行比对而检测的，即输入时的关注用户名与关注列表功能得到的关注用户名可一一匹配，且关注所有用户时关注数与总用户数相差 1，证明关注实现。关注时不能关注自己以及已经被关注过的人。

关于关注的效率，由于程序需要保证不能关注已经关注的人，因此在插入的时候遍历有 O(n)的复杂度，不过插入本身是 O(1)的复杂度，因此预测关注的用户数量与时间存在 2 次幂函数关系。效率测试所得数据见 Statistic_like.txt，数据图证明预测关系成立。

之前对于关注的想法是一个用户使用两个链表，一个存储关注信息，一个存储被关注信息。而这种有向图十字链表的实现方式比第一种更为节省硬盘空间。

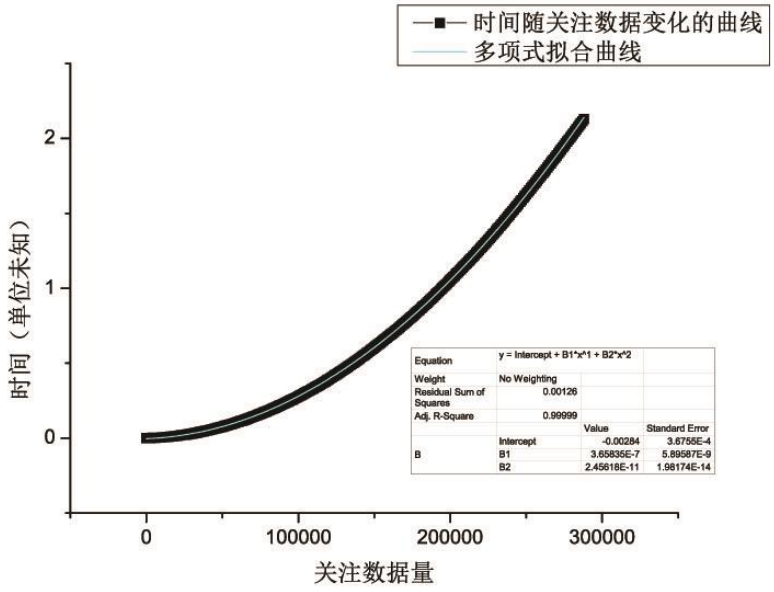

Figure 7 关注效率测试

2.4 信息

信息的存储实现了两种方式，第一种方式是在程序退出后保存，信息按照事先先后保存，在程序未退出时保存在内存里，这种方式不是很可取，效率低，且代码冗长，在单次程序中发送信息过多还会造成明显的内存占用。在 ver8 之前均为这种方法。而后来实现了一个即时存储的方法，而且是按照时间的倒序存储，在显示信息列表的时候不需要做一次遍历。

由于信息是在程序退出的时候保存的，在发送信息的时候只是把信息放在内存中，因此程序在发送信息的时候会很快，会在常数时间内完成。但是在程序退出时则需要较多时间，且在退出存储的版本中使用的是时间顺序存储，存储新的数据时需要遍历一遍数据寻找位置，所以时间复杂度应为 O(n)，测试如图 Figure 6 所示。而在程序退出时会将内存的修改刷入硬盘，因此会比即时刷入硬盘要慢。符合预期的想法。这种方式与即时保存各有利弊，之所以这样实现是因为之前的所有实现均是即时保存，即时读入，因此想尝试下启动读入，退出写入的方式的写法如何。实践证明在发送信息的时候确实会比预想快不少。至于退出时候的复杂度，预测是 O(n)的插入复杂度，因此数据量随时间大概是二次幂函数关系。

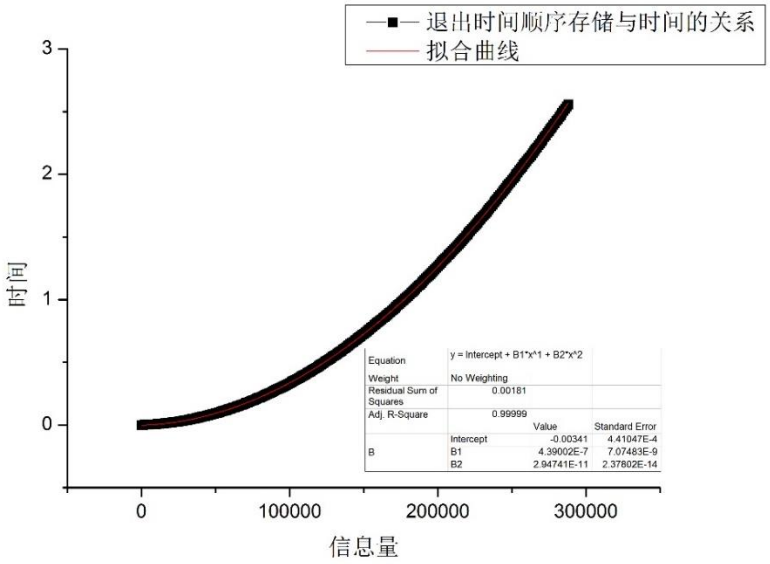

Figure 8　ver8 之前实现的消息效率测试

而在后来实现的即时存储的实现版本中，是按照时间倒序插入，因此在插入时不需要遍历，插入是常数时间进行的，因此预测曲线应该是线性的。不过如果同样是时间倒序的话，即时保存理论上与退出保存并无不同，只是退出保存需要占用较大内存。

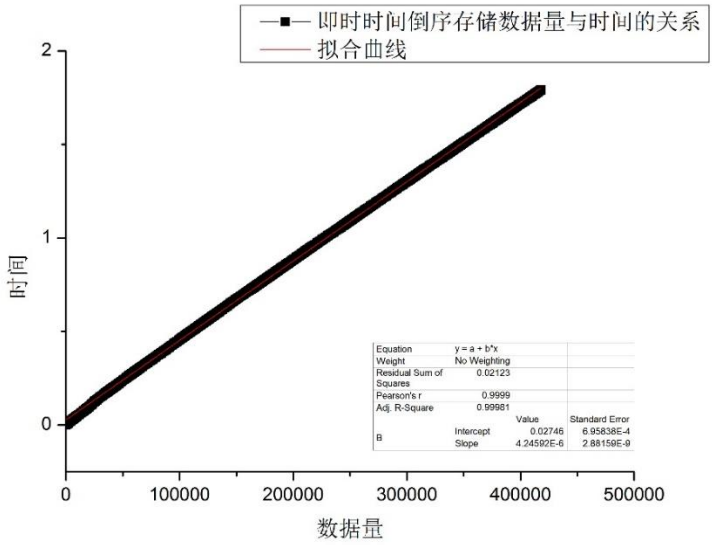

Figure 9 最终实现的消息效率测试

最终就如 Figure 9 所显示的那样，整个信息的发布是在常数时间内完成，可见对信息发布的改进十分有效。

第三章 总结

3.1 程序总结

整个程序的耗时较多，几乎花费了半个寒假在不断地完善，在写代码的过程中也的确学到了不少东西，比如哈希索引的工作原理，文件操作的一些注意事项等等，总的而言收获不小。

3.2 遇到的问题

程序在开始设计的时候希望可以分为三个主要模块：信息，关注，用户，其中信息与关注是与信息相互结合，然后一起组成用户，然后由 UI 调用。但是最终代码走样，导致界面类结构臃肿，主要是不太懂得软件结构的设计方法，希望本学期学习软件工程原理后可以写出更加漂亮的代码结构。

此外，由于程序是在 visual studio 2013 下编写，因此在进行 C 风格字符串操作的时候编译器会常常报错，原因为编译器认为 c 风格字符串函数存在安全问题，最终在配置属性里添加了忽略警告才能继续使用。另外，在文件操作时起初不知道怎样在不对文件进行覆盖的前提下怎样进行文件修改操作。最终是使用 fstream 的读写模式同时打开完成了修改，但是原理不知。

3.3 可改进的地方

程序在界面上还可以更加友好一点，而且没有实现自己信息的分页，希望以后会有机会继续完善下去。