

## 基于闭散列 hash 表的数据库实现

刘洋 515030910486

### 第一部分 概述

本 database 为基于闭散列（close addressing）hash 表的 key(string) – value(string)型数据库，具有支持 C++标准 string 类，高度面向对象等特点。储存 key 时，database 进行三次 hash 不同算法的 hash 计算，第一次生成的 hash 值用于指示 key 在 bucket 中的初始位置，剩余两个 hash 值用于验证以保证冲突率降至最低。通过将所有 bucket 储存在磁盘，程序可以随时保存/读取哈希表。通过储存于磁盘的 hash table 实现的极低冲突率，本程序基本可以在  $O(1)$  时间内进行 insert, replace, fetch, delete 等操作。然而在使用本 database 存在不可突破的最大容量，并且本 database 必须牺牲部分磁盘空间以储存占用空间较大的 hash table。

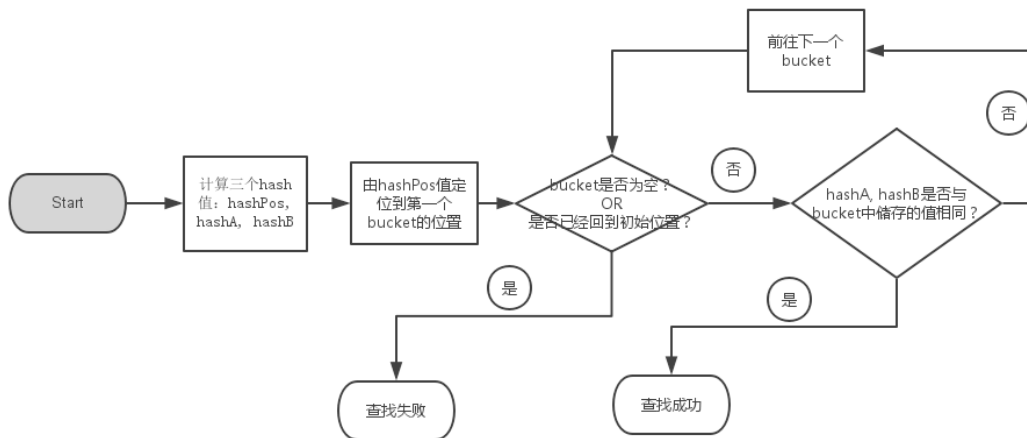
### 第二部分 database 具体构架

#### 2.1 MPQ hash 算法与多次 hash 检验机制

MPQ hash 算法是由 Mike O'Brien Pack 发明的高效 one-way-hash 算法，曾用于暴雪娱乐公司

（Blizzard Entertainment）旗下的多款游戏中。本数据库对于此算法的实现中用到了一个较为随机且生成方法固定的 crypt table，两个 unsigned long（4 字节）的 seed，以及一个用于影响在 crypt table 中定位的 unsigned long（4 字节）dwHashType。通过传入不同的 dwHashType 值可以从 crypt table 中取出不同的值，从而最终生成不同的 hash 值。

为了提高 hash table 的使用率并降低字符串 hash 值冲突几率，本 hash table 选择通过三次不同的 hash 计算来决定最终的入口点（bucket 的位置）。通过采用这种多次 hash 检验机制，不同 string 用三个不同的哈希算法算出的 hash 值均一致基本是不可能的，概率为  $1:1888946593147858085478$ ，大概是 10 的 22.3 次方分之一。本 hash table 采用线性试探（linear probing）法来解决开放定址问题。查找 hash 值的步骤如下：



插入与删除 hash 值的步骤与查找类似，在此便不再赘述。

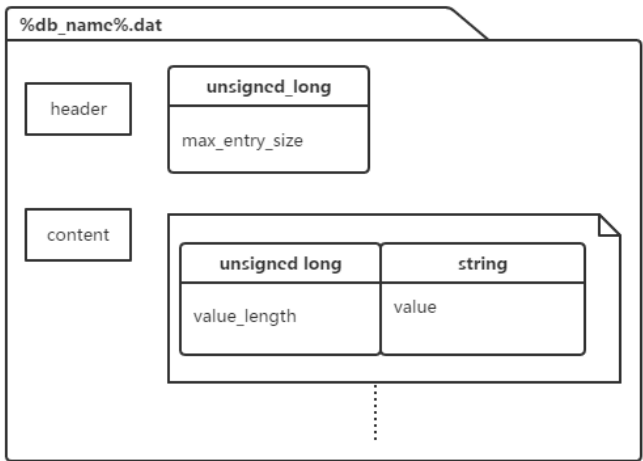
## 2.2 文件系统概述

本数据库会根据数据库名生成四个作用不同的文件。假设数据库名为%db\_name%，则生成的文件名及它们的作用将会是：

- %db\_name%.dat，保存所有 value 值
- %db\_name%.dat.meta，保存所有可覆盖 value 的位置与大小
- %db\_name%.idx，保存 hash table 及 index 条目
- %db\_name%.idx.meta，保存所有可覆盖 index 条目的位置大小

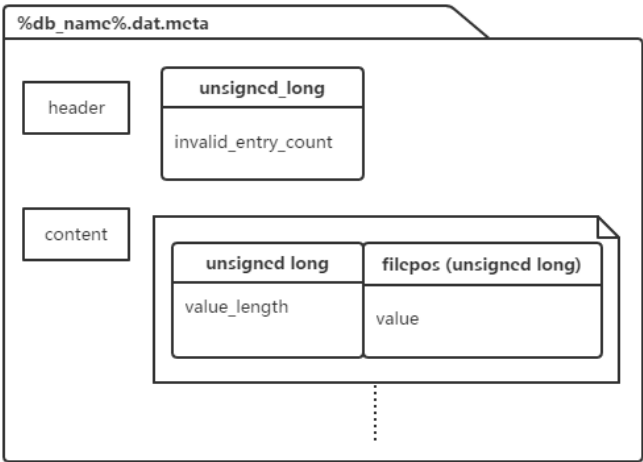
### 2.2.1 %db\_name%.dat

本文件保存所有 value 值。本文件同时会保存单个 value 值的有效性（便于恢复）和大小（便于取会 string value）。%db\_name%.dat 的结构如下：



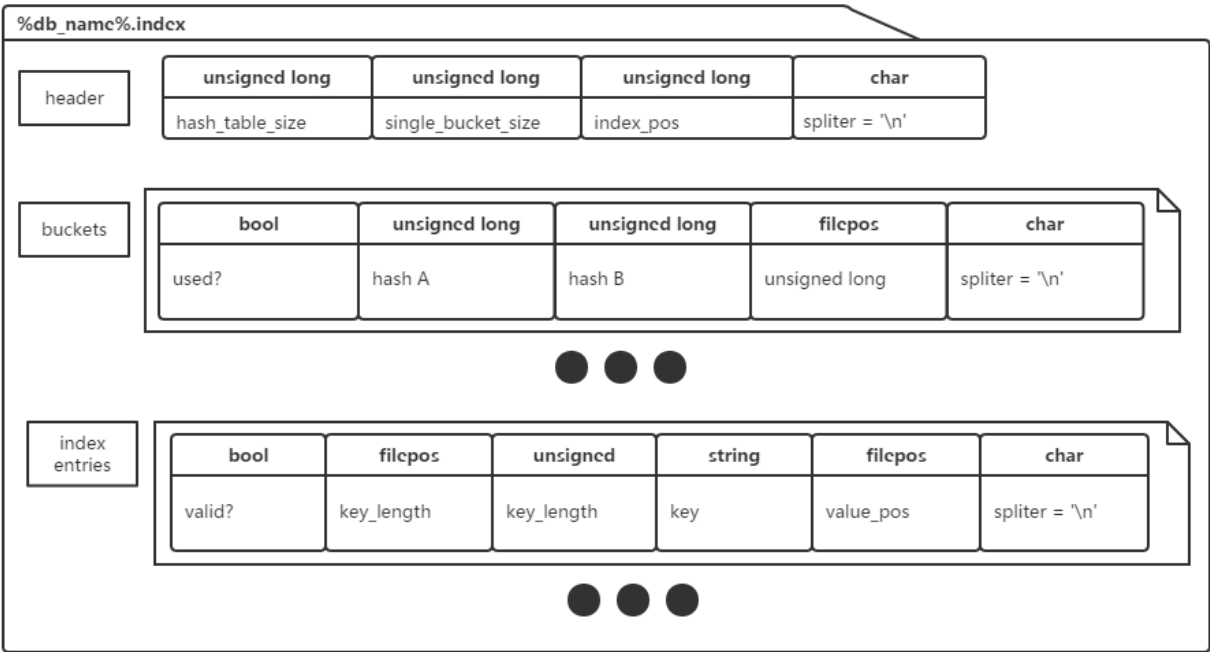
### 2.2.2 %db\_name%.dat.meta

此文件只会在关闭数据库时被新建（覆写）。%db\_name%.dat.meta 的结构如下：



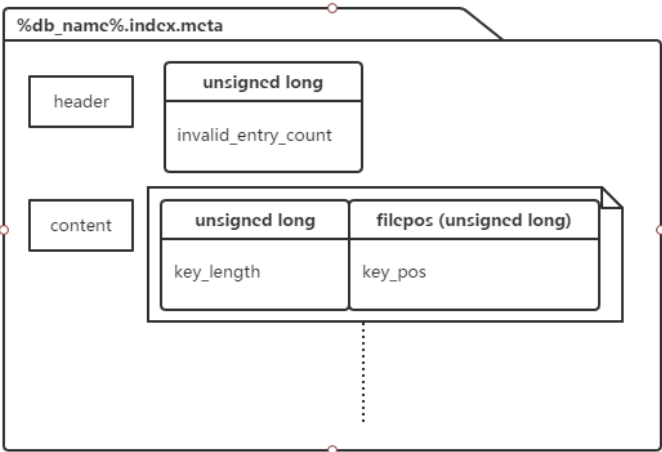
2.2.3 %db\_name%.idx

本文件分为文件头，hash table(buckets)，index entries 三个部分。Hash table 部分只会在数据库被关闭时被保存，其余时间均会在内存中。%db\_name%.idx 的结构如下：



2.2.2 %db\_name%.idx.meta

此文件只会在关闭数据库时被新建（覆写）。%db\_name%.idx.meta 的结构如下：



## 2.3 接口概述

本数据库提供面向对象的 DB 类，相关操作均由类的成员函数实现。需要说明的开放接口有：

```
class HashDatabase {
public:
    HashDatabase(const string & _file_name, unsigned long _table_size = MAXTABLESIZE);
    ~HashDatabase();

    int db_insert(const string& key, const string& value, int flag = 0);
    int db_replace(const string& key, const string& value);
    string db_fetch(const string& key);
    int db_delete(const string& key);
}

typedef HashDatabase DB;
```

### 2.3.1 构造函数 (constructor)

```
#define MAXTABLESIZE 2000000
```

```
HashDatabase(const string & _file_name, unsigned long _table_size = MAXTABLESIZE);
```

`_file_name` 为数据库的地址（包括数据库的名称），数据库所生成的四个文件的文件名将由这个参数决定。`_table_size` 为数据库的容量，也就是数据库的最大条目数，默认值为 2000000。值得注意的是，在数据库达到最大条目数之后，尝试继续插入数据将会失败。初始化文件系统的操作也由此构造函数过程。如果已经存在数据库文件，此函数会尝试读取。而如果不存在数据库文件，此函数会新建数据库所需的各项文件。

### 2.3.2 析构函数 (destructor)

```
~HashDatabase();
```

在 DB 类被析构之际将会调用此析构函数。此析构函数将会将内存中的 hash table 写入 idx 文件，并将没被覆写的已删除 index entry 之相关信息写入 idx.meta 文件。dat 和 dat.meta 文件将会由另一个类 DataManager 负责，可以保证相关文件的正确读写。

### 2.3.3 插入 (insert)

```
int db_insert(const string& key, const string& value, int flag = 0);
```

调用此成员函数可以向数据库中插入一组 {key, value} 数据。如果成功，将会返回 0。需要注意的是，如果数据库中已经存在了该 key，插入将会失败。

### 2.3.4 更改 (replace)

```
int db_replace(const string& key, const string& value);
```

调用此成员函数可以修改数据库中 key 指向的 value。如果成功，将会返回 0。

### 2.3.5 读取 (fetch)

```
string db_fetch(const string& key);
```

调用此成员函数可以尝试在数据库中找回 key 对应的 value。如果成功，将以 string 类的形式返回 value 值。如果失败，将返回空 string ("" )。

### 2.3.6 删除 (delete)

```
int db_delete(const string& key);
```

调用此成员函数可以删除一组 {key, value} 数据。如果成功，将会返回 0。

## 第三部分 测试

### 3.1 正确性测试

#### 3.1.1 测试 A

测试 A 先随机插入  $nrec$  条记录，接着读取这  $nrec$  条记录，接着随机替换掉其中一条记录，再尝试读取被替换的记录，最后删除所有数据。Insert, replace, fetch, delete 的正确性都可以在测试 A 中被检验。如果其中没有报错，则说明有数据库的实现有很大概率是正确的。

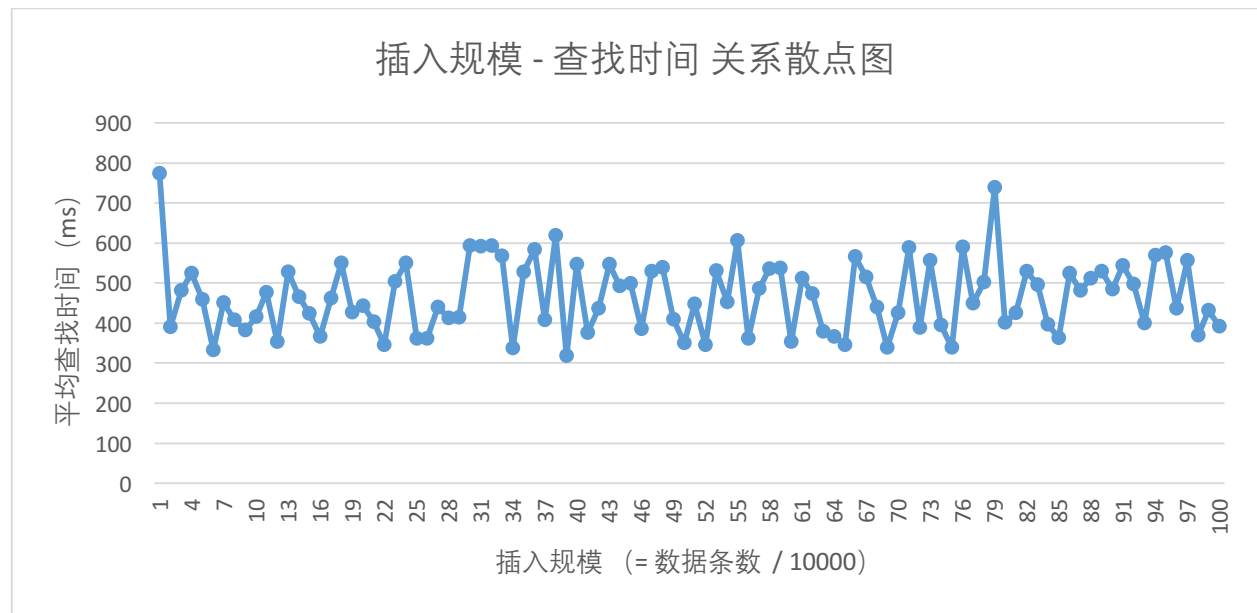
#### 3.1.2 测试 B

观察运行以下测试时程序是否会报错：

- (1) 向数据库写  $nrec$  条记录。
- (2) 通过关键字读回  $nrec$  条记录。
- (3) 执行下面的循环  $nrec \times 5$  次。
  - (a) 随机读一条记录。
  - (b) 每循环 37 次，随机删除一条记录。
  - (c) 每循环 11 次，随机添加一条记录并读取这条记录。
  - (d) 每循环 17 次，随机替换一条记录为新记录。在连续两次替换中，一次用同样大小的记录替换，一次用比以前更长的记录替换。
- (4) 将此进程写的所有记录删除。每删除一条记录，随机地寻找 10 条记录。

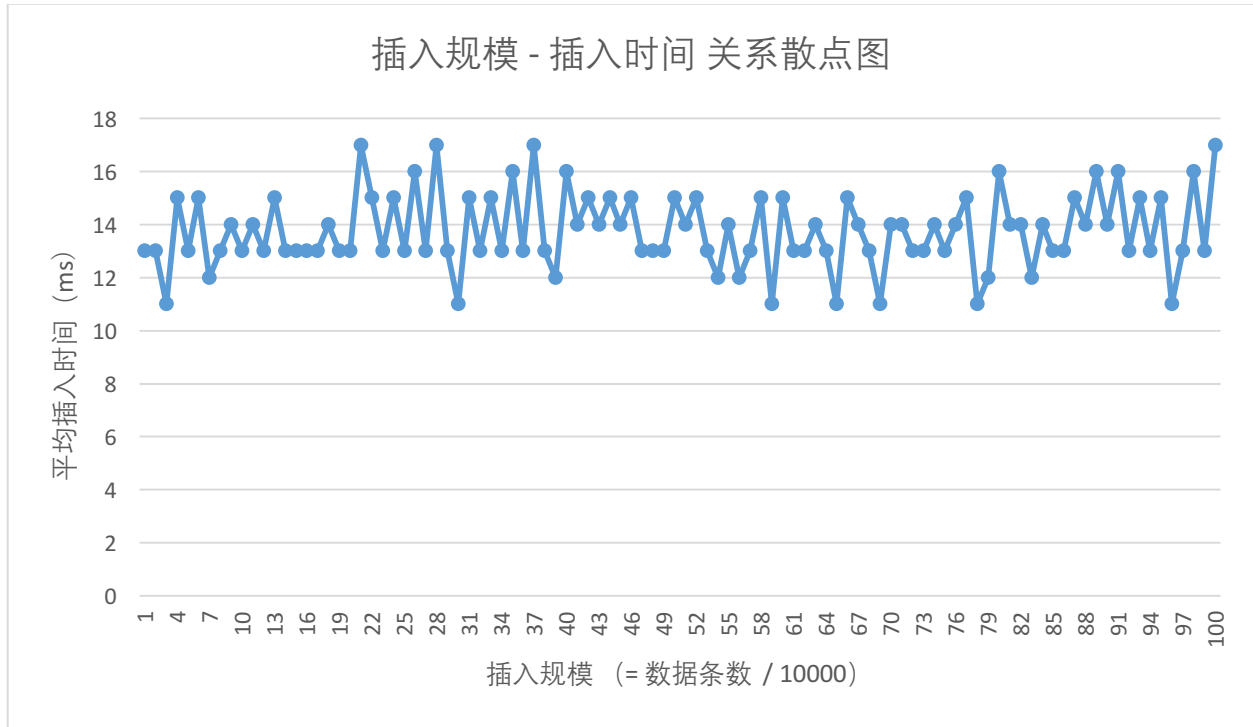
### 3.2 性能测试

首先将测试 fetch 操作耗时与数据规模的关系。通过储存于磁盘的 hash table 实现的极低冲突率，本程序基本可以在  $O(1)$  时间内进行 fetch 操作。以下是数据量在 1000000 一下时，进行 fetch 操作耗时与数据规模的关系图。测试时，每写入 10000 组数据便会读取最近写入的 100 组数据，并计算平均读取时间。



通过观察此散点图可以发现，查找时间的分布在一定的范围内抖动，可以基本认为 fetch 操作的时间复杂度为  $O(1)$ 。进行 fetch 操作所需的时间与数据规模并没有太大的相关性。

其次将测试 insert 操作耗时与数据规模的关系。可以推断，此操作的时间复杂度也应为  $O(1)$ 。以下的插入规模 - 插入时间 关系散点图 从侧面印证了这一点。



通过进行性能测试，可以发现制约插入速度的主要瓶颈是文件写入的方法有改进的空间。现有的方法是每新建一条数据便对文件进行一次读写，而这种较为随机离散的读写方法很难有很快的速度。为进一步改进此数据库，下一步可以考虑将部分{key, value}数据先保存在内存进行读写，当内存中的数据组数达到一定数量后，再进行磁盘的读写。