哈希函数相关

哈希函数：

$$
\begin{gathered}
y = f_{hash}(x) \\
x \in [-\infty, \infty], y \in [-s, s]
\end{gathered} \tag{1}
$$

例如：

$$
\begin{gathered}
f_{MD5} \to y \in [0, 2^{64} - 1] \\
f_{sha1} \to y \in [0, 2^{128} - 1]
\end{gathered} \tag{2}
$$

- 输入可以无限，输出一定有限
- 相同的输入，一定返回相同输出
- 不同的输入，可能导致相同的输出（哈希碰撞，概率比较低）
- 哈希函数的返回值均匀随机分布（离散型+均匀性）

一个特殊的操作：

$$
y = f_{hash}(x)\%m, y \in [0, m-1] \tag{3}
$$

问题一：

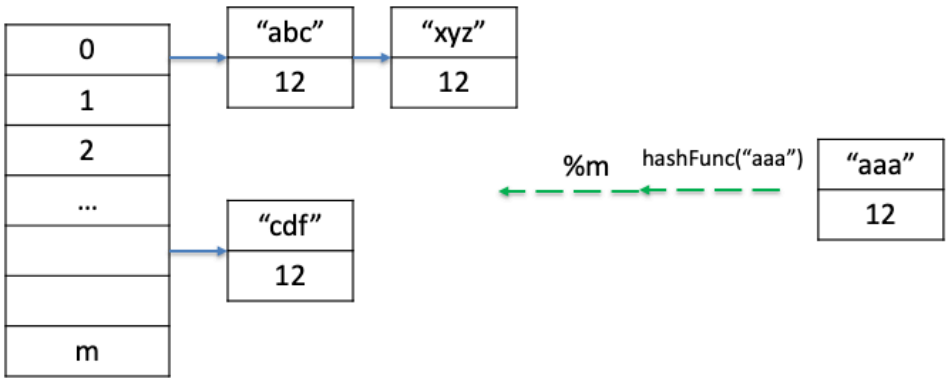
> 考虑一个很大的文件，字符数有40亿个，同时每个字符为无符号整型，也就是每个字符占用4个字节，如何在限制的低内存条件下统计这个大文件中出现频次最高的字符。

解决：

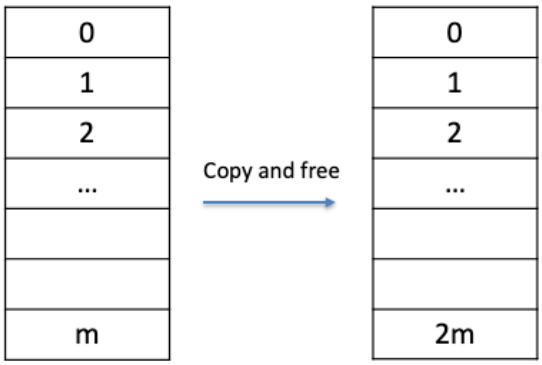
使用hash函数+%m的方式，将大文件中的所有字符分解到m个小文件中，然后可以用hashmap在每个小文件中统计，每个小文件中统计出频次最高的，在m最高频次之间进行一个比较就能得到大文件中频次最高的那个字符。

哈希表:

添加过程



扩容过程:



分析扩容的代价:

假设开始的长度为2，单链表的长度为k，那么给你N的字符，该如何扩容呢?

按照哈希函数的随机性，N个字符需要的空间为： $2^n \times k$,当N很大是，n也会很大$\rightarrow 2^n$，所以时间复杂度为$O(logN)$,每次扩容之后需要将原来的内容重新拷贝到新的哈希表中，代价为$O(N)$,所以总的代价为$O(NlogN)$,则单次的代价为$\frac{O(NlogN)}{N} = O(logN)$,如果考虑N不是很大的情况下，可以认为是逼近$O(1)$的代价

问题二:

```
【题目】
设计一种结构RandomPool，在该结构中有如下三个功能
insert（key）：将某个key加入到该结构，做到不重复加入，

delete（key）：将原本在结构中的某个key移除
getrandom（）：等顺率随机返回结构中的任何一个key
【要求】
insert，  delete和 Iget Rand方法的时间复杂度都是0（1）
```

```
class RandomPool
{
public:
    RandomPool()
    {
        size=0;
    };
    void insertKey(char c)
    {
        if(strToIndex.find(c)!=strToIndex.end()) return; // no repeat insert
        strToIndex.insert(make_pair(c,size));
        indexToStr.insert(make_pair(size,c));
        size++;

    };
    void deleteKey(char c){
        if(strToIndex.find(c)==strToIndex.end()) return; // no this char.
        // swap the item and last item
        if(strToIndex.size()<1) return;
        char lastC=indexToStr.find(size-1)->second;
        int deleteCIndex=strToIndex.find(c)->second;
        indexToStr[deleteCIndex]=lastC; // swap the last c to delete index
        indexToStr.erase(indexToStr[size-1]); //delete last recorder.
        size--;
        //strToIndex swap
        strToIndex.find(indexToStr[deleteCIndex])->second=deleteCIndex; // lastC: deleteCIndex
        strToIndex.erase(strToIndex.find(c)); //remove char c

    };

    char getRandomKey()
    {
        // gen random index in [0,size-1]
```

```
34          int randomIndex=rand()%size;
35
36          return indexToStr[randomIndex];
37
38      }
39
40      int randomPoolSize()
41      {
42          return size;
43      }
44
45
46  private:
47      map<char,int> strToIndex;
48      map<int,char> indexToStr;
49      int size;
50
51  };
```

布隆过滤器

为了解决一个很大的集合，需要确定输出的元素是否在这个大的集合中，能在较短的时间内查询到该元素。

设计一个bitmap，一个长度为m，每个位置都是一个bit的存储空间，输入的每个元素都通过调用k个哈希函数再%m，得到k个位置，然后在bitmap相应的位置上置1（初始都是0）。通过这样的方式，查询的时候，输入相同的元素，通过上述过程，如果要查询的位置都为1，那么就认为有这个元素的存在。

由于哈希函数本身的性质，即相同的输出一定有相同的输出，所以不存在：本来元素在大集合中，而查询的结果是没有；由于bitmap的长度m有限，所以会有不同的输入有着相同的查询结果，导致错误率的产生。

$$
\begin{gathered}
n\text{样本量，}p\text{失误率，}m\text{：}bitmap\text{长度，}k\text{：哈希函数数量} \\
m=-\frac{n*lnp}{(ln2)^2} \\
k=ln2*\frac{m}{n}\approx 0.71*\frac{m}{n} \\
p_{\text{真}}=(1-e^{\frac{n*k_{\text{真}}}{m_{\text{真}}}})^{k_{\text{真}}}
\end{gathered}
\tag{4}
$$

```
1   // bitmap implement
2
3   class BitMap
```

```
4   {
5   public:
6       BitMap(int size){
7           m=size*32;
8           vector<int> tmp(size);
9           for(auto &item : tmp)
10          {
11              item=0;
12          }
13          bitmap=tmp;
14      };
15      void setIndexTrue(int index)
16      {
17          Index *res=locationIndex(index); // get the bit index and num index.
18  //        cout<<res->numIndex<<endl;
19  //        cout<<res->bitIndex<<endl;
20          bitmap[res->numIndex] = bitmap[res->numIndex] | (1<<(res->bitIndex));
21      };
22      void setIndexFalse(int index)
23      {
24          Index *res=locationIndex(index); // get the bit index and num index.
25
26          bitmap[res->numIndex]= bitmap[res->numIndex] & (~(1<<(res->bitIndex)));
27      }
28      int  getIndexState(int index){
29          Index *res=locationIndex(index); // get the bit index and num index.
30          return (bitmap[res->numIndex] >> (res->bitIndex)) & 1;
31      }
32
33      int size()
34      {
35          return m;
36      }
37  private:
38
39      vector<int> bitmap;
40      int m;
41      struct Index{
42          int numIndex;
43          int bitIndex;
44          Index(int i,int j):numIndex(i),bitIndex(j){};
45      };
46
47      Index*  locationIndex(int i)
48      {
49          int numIndex=i / 32;
50          int bitIndex= i % 32;
```

```
51          return new Index(numIndex,bitIndex);
52  //          return p;
53      };
54
55  };
```

一致性哈希原理

考虑分布式数据服务器数据迁移代价

queation：

考虑有一个分布式系统，这个系统中有m台数据服务器，专门用于数据存储用途，如何设计数据服务器的寻址方式可以做到：

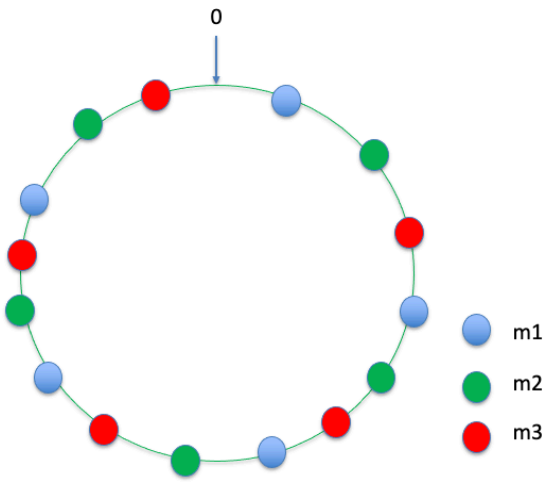
- 每台数据服务器的负载可以人为的设计（不同的机器有不同的性能）
- 数据的迁移成本尽可能的低：添加新的服务器/移除原来的一定数量的服务器

（google改变世界的三驾马车之一算法）

首先需要选取hash函数的输入key，对于m台数据服务器，总可以用ip地址 /host name/ mac地址/..,等这些信息来区分，从中选择一种左右key即可，假设选择机器的mac地址作为hash算法的输入的一部分，针对每台服务器设计k_i个虚拟节点（为了设计不同的负载均衡以及较均匀的抢占在“环”上的位置），这样总可以为每台服务器的虚拟节点赋值一个值：$mac_{address}+i$,作为哈希函数的key，所以对于第i台数据服务器：

$$S_i:F_{hash}(Mac_{address}+j),j\in[0,k_i-1] \tag{5}$$

因为hash函数的输出在$[0,2^n-1]$,也就会产生2^n个位置，数值可以看作服务器的“标号”，可以将这些位置看作是组成了一个“环”，m台服务器，每台服务器都有k_i个虚拟节点在环上几乎均匀的分布；



- 数据增删改查

  一条记录通过hash函数总可以得到一个“位置”，然后在“环”上顺时针找到最近的虚拟节点，然后对应到实际的服务器即可，然后进行数据库的各种操作即可；

```
// 查询过程伪代码
// input parameters: data_item
// hash function: hash
// unreal nodes arr: unrealnodes

int index=hash(data_item);
// 在unrealnodes中二分找到>=index的值，就是要选择的虚拟节点的值，
// 如果大于了2^n -1 或者小于0， 则看作在2^n -1 ~ 0 之间
node* nodeIndex = sortFindIndex(index);
// 对应到具体的真实服务器
int serverId=assign(nodeIndex);

// 进行正常的操作...
```

- 数据迁移代价较小
  - 如果增加新的服务器，同样的也要设计固定数量的虚拟节点，实际上在“环”上，**每个虚拟节点所要负责的可能位置为从自身逆时针到下一个虚拟节点的范围**，所以增加了新的虚拟节点之后相当于在“环”上内插了更多的虚拟节点，按照上述的负责模式，从其他的虚拟节点中拷贝出“自己负责的那部分数据”，然后在原来的节点中“释放”空间即可；
  - 如果删除服务器，也是同样的过程，拷贝删除服务器的数据到顺时针下一个节点上即可

每台服务器的负载均衡可以通过在环上的虚拟节点的数量也就是k_i来决定，可以很容易设计。