原创

Data Structure -- 哈希表

2020-12-17 阅读 42

1. 什么是哈希表?

Main idea: Map the keys to a small range of integers and then use direct addressing.

根据关键码值(Key)获取一个地址（index）从而直接进行访问的数据结构。

2. 与数组的区别

如果我们通过数组的方式直接创建一个dictionary来储存(key - value)，它所对应的操作有：

Search（key）：根据key值确认是否存在对应的value， 时间复杂度：O(1)

Insert （key，value）：A[key] = value， 时间复杂度：O(1)；如果超出数组长度，则需要创建新的数组（通常是Double size），然后拷贝，最后插入， 时间复杂度：O(N)

Delete（key）：A[key] = null， 时间复杂度：O(1)

缺点：

1. 如果key值不是整数，dictionary将会失效。
2. 当储存个数 < 数组大小 的时候，会浪费空间。
3. 数组长度固定

3. 与链表的区别

如果我们通过链表的形式来储存（key - value), 他所对应的操作有：

Search（key）：根据key值确认是否存在对应的value，遍历链表，时间复杂度：O(N)

Insert （key，value）：在链表最后新建一个节点，Node.key = key, Node.value = value， 时间复杂度：O(1)

Delete（key）：先遍历链表找到key的节点，进行删除操作， 时间复杂度：O(N)

4. 哈希表（Hash Table)

哈希表结合了数组和链表的形式，如果一个哈希表T的大小是M， 定义了一个哈希函数 f: key -> Index （index 的范围是从0到M-1），T[f(key)] = value。哈希表最大的优点，就是把数据的存储和查找消耗的时间大大降低，几乎可以看成是常数时间

作者介绍

成都小展

关注 专栏

文章	阅读量	获赞	作者排名
2	268	4	6433

目录

1. 什么是哈希表?
2. 与数组的区别
3. 与链表的区别
4. 哈希表（Hash Table)
5. 哈希冲突 （Collisions）
Chaining
Open addressing

6. 哈希函数的独立性
7 Cuckoo Hashing
8. 哈希表的应用

Hashing example

$U = \mathbb{N}$, $M = 11$, $h(k) = k \bmod 11$.

The hash table stores keys 7, 13, 43, 45, 49, 92. (Values are not shown).

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

Index of key 7: h(7) = 7 mod 11 = 7

Index of key 13: h(13) = 13 mod 11 = 2

Index of key 43: h(43) = 43 mod 11 = 10

Index of key 45: h(45) = 45 mod 11 = 1

Index of key 49: h(49) = 49 mod 11 = 5

Index of key 92: h(92) = 92 mod 11 = 4

5. 哈希冲突 （Collisions）

当两个key带入同一个h哈希函数的时候，index相等（如上：h(46) = h(13) = 2 ），当前者插入哈希表的时候，哈希表已经被占领。

两种基本办法解决冲突：

（1）允许多个value值插入同一个index（chaining）

（2）允许value 值进入多个index （Open addressing: probe sequence， cuckoo hashing）

（定义Load Factor， 用来评估一个哈希表Search、Insert和Delete

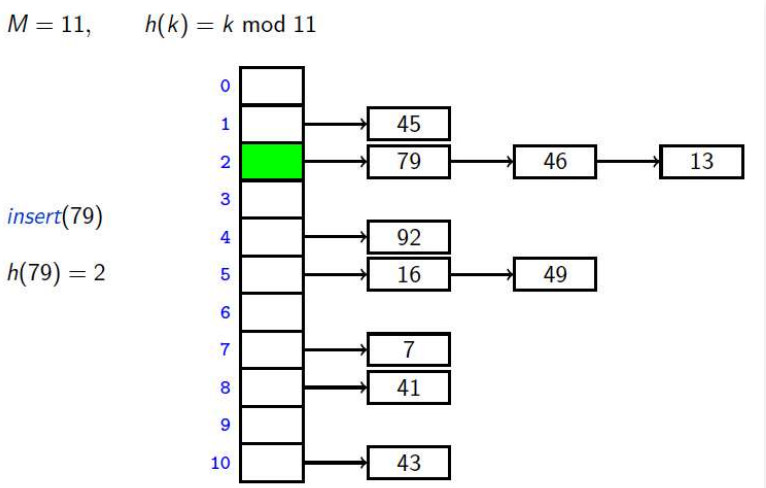
load factor a = n / M，n是当前key的数量，M是哈希表的大小）

Chaining

Search(k): 在T[h(k)]的链表里查询key k 时间复杂度：平均O(1+ a)，最差情况O(n)

Insert(k, v): 将（key - value) 添加到T[h(k)] 链表的开头 时间复杂度：O(1)

Delete(k): 先执行Search， 再从链表里面删除 时间复杂度：平均O(1+ a)，最差情况O(n)



Insert 41: h(41) = 8

Insert 46: h(46) = 2

Insert 79: h(79) = 2

Open addressing

每个哈希表地址只能存储一个value， 但是任何一个key k 能存在任何位置。 根据probe sequence： <h(k,0), h(k,1), h(k,2)... > 一直找到empty 的index。

最常用的open addressing: linear probing

h(k, i) = (h(k) + i) mod M

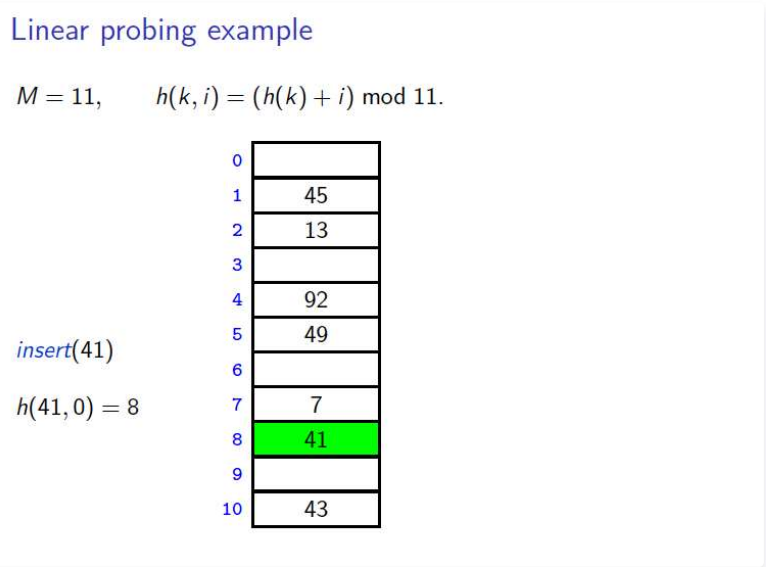
Search(k): 从 h(k，0)开始搜寻，直到找到key或者空的地址　　时间复杂度：平均O(1+ a)，最差情况O(n)

Insert(k, v): 将（key - value) 从h(k,0) 开始找一个空的地址存放　时间复杂度：平均O(1 + a)，最差情况O(n)

Delete(k): 先执行Search， 再从链表里面**延时删除**　　时间复杂度：平均O(1+ a)，最差情况O(n)

延迟删除：将删除的index标记为deleted，而不是直接清空， 这样保证在Search操作的时候，能够search足够远。

对于 h(k) = k mod 11



h(41, 0) = (41 mod 11 + 0) mod 11 = 8

Linear probing example

$M = 11, \quad h(k, i) = (h(k) + i) \bmod 11.$

insert(84)

$h(84, 2) = 9$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

i = 0: h(84, 0) = ((84 mod 11 + 0) mod 11) = 7 （已被占用）

i = 1： h(84, 1) = （(84 mod 11 +1) mod 11) = 8 （已被占用）

i = 1： h(84, 1) = （(84 mod 11 +2) mod 11) = 9 （未被占用）

Open addressing 采用的是延迟删除：将删除的index标记为deleted，而不是直接清空， 这样保证在Search操作的时候，能够search足够远。

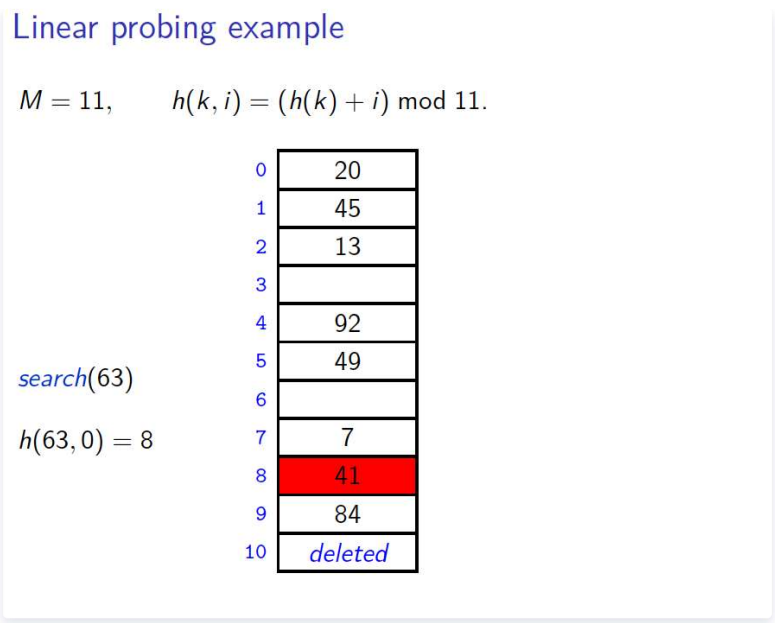
Linear probing example

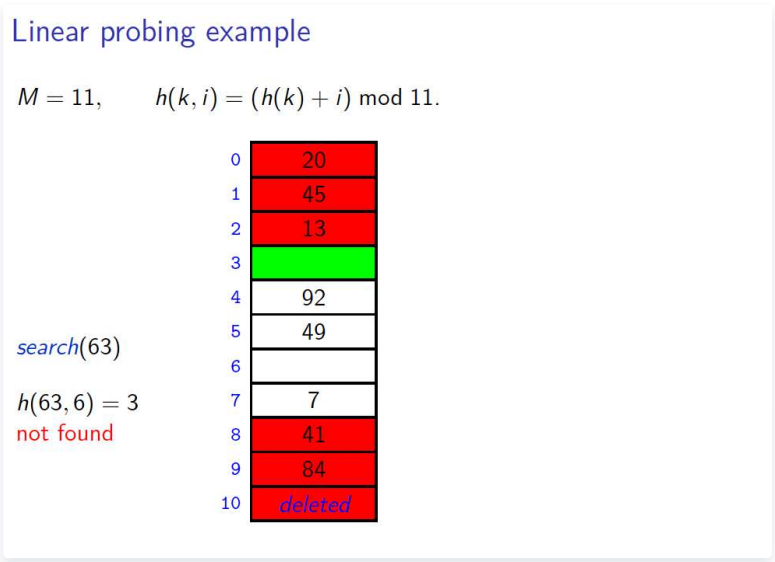
$M = 11, \quad h(k, i) = (h(k) + i) \bmod 11.$

delete(43)

$h(43, 0) = 10$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	*deleted*





6. 哈希函数的独立性

如果一些哈希表运用的方法包含了两个哈希函数 h1(k), h2(k)， 那么这两个函数应该独立存在。

（These hash functions should be independent in the sense that the random variables P(h1(k) = i) and P(h2(k) = j) are independent）

通常的两个函数选取，多多少少对导致相互依赖，对于h2(k) 的选取最好是采用 multiplicative method：

h(k) = [M(kA − [kA])] （ [X]： 表示取X的整数部分）

A的取值建议： 0.618

下面的例子将两个哈希函数运用到了开发地址法（open addessing) 上， 即：

h(k, i) = h1(k) + i · h2(k) mod M

Double hashing example

$M = 11, \quad h_1(k) = k \bmod 11, \quad h_2(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	
9	
10	43

Double hashing example

$M = 11, \quad h_1(k) = k \bmod 11, \quad h_2(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$

insert(41)

$h_1(41) = 8$

$h(41, 0) = 8$

0	
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	
10	43

Double hashing example

$M = 11, \quad h_1(k) = k \bmod 11, \quad h_2(k) = \lfloor 10(\varphi k - \lfloor \varphi k \rfloor) \rfloor + 1$

insert(194)

$h_1(194) = 7$

$h(194, 0) = 7$

$h_2(194) = 9$

$h(194, 1) = 5$

$h(194, 2) = 3$

0	
1	45
2	13
3	194
4	92
5	49
6	
7	7
8	41
9	
10	43

h（194， 0） = (h1(194) + 0 * h2(194)) mod 11 = 7 （哈希冲突)

h（194， 1） = h1(194) + 1 * h2(194) =

= (9 + 1 * [10 * (0.618 * 11 - [0.618 * 11]]) mod 11

= (9 + 7) mod 11

= 5 (哈希冲突)

h（194， 2） = h1(194) + 2 * h2(194) =

= (9 + 2 * [10 * (0.618 * 11 - [0.618 * 11]]) mod 11

= 3 （未被占用，插入）

7 Cuckoo Hashing

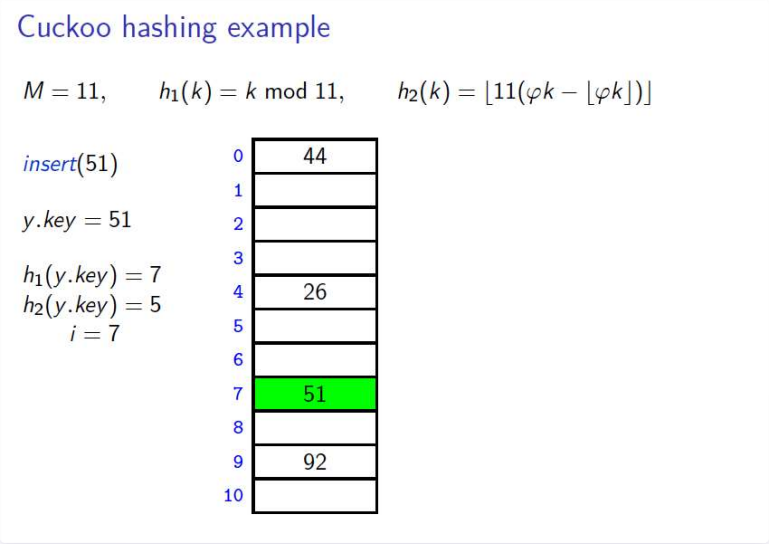
Cuckoo Hashing 方法包含了两个哈希函数 h1(k), h2(k), 且key的存放地址**只能在 T[h1(k)] 或者 T[h2(k)]**

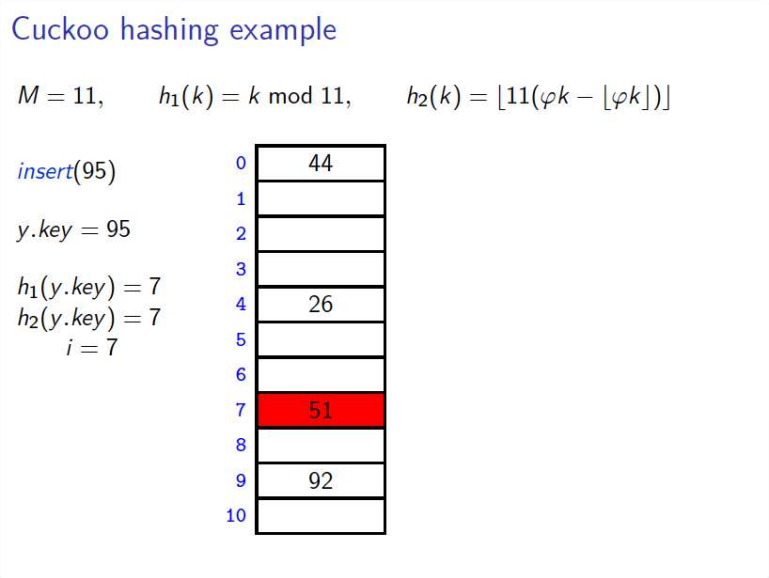
这个方法的好处是Search 和 Delete 时间复杂度都变成了O(1)

Search(k): 找T[h1（k)] 或者 T[h2(k)]　　时间复杂度：O(1)

Delete(k): 找T[h1（k)] 或者 T[h2(k)], 再删除　　时间复杂度：O(1)

Insert(k, v): 将（key - value) 从T[h1（k)] 开始，如果T[h1（k)] 被占用，将T[h1（k)] 的里原有值踢出去（kich-out)，这个原有的值会重新插入。





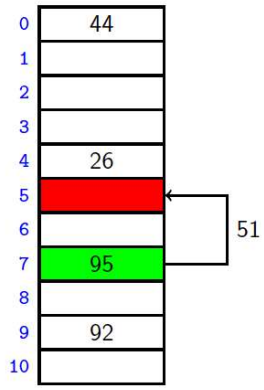
Cuckoo hashing example

$M = 11, \quad h_1(k) = k \bmod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(95)

$y.key = 51$

$h_1(y.key) = 7$
$h_2(y.key) = 5$
$i = 5$



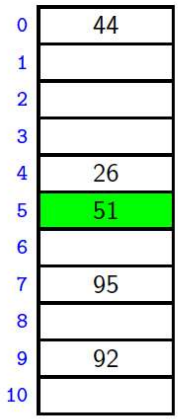
Cuckoo hashing example

$M = 11, \quad h_1(k) = k \bmod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(95)

$y.key = 51$

$h_1(y.key) = 7$
$h_2(y.key) = 5$
$i = 5$



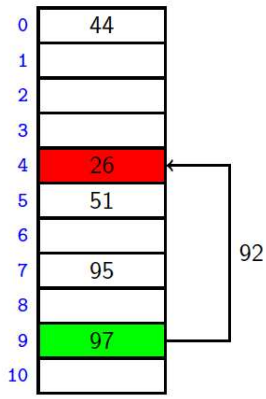
Cuckoo hashing example

$M = 11, \quad h_1(k) = k \bmod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(97)

$y.key = 92$

$h_1(y.key) = 4$
$h_2(y.key) = 9$
$i = 4$



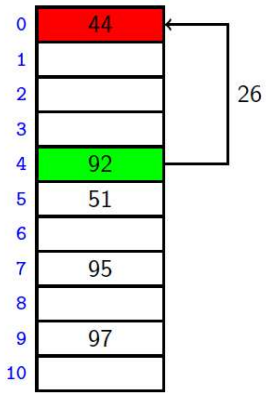
Cuckoo hashing example

$M = 11, \quad h_1(k) = k \bmod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(97)

$y.key = 26$

$h_1(y.key) = 4$
$h_2(y.key) = 0$
$i = 0$



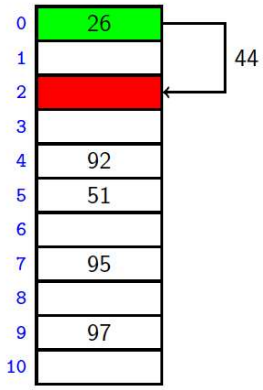
Cuckoo hashing example

$M = 11, \quad h_1(k) = k \bmod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(97)

$y.key = 44$

$h_1(y.key) = 0$
$h_2(y.key) = 2$
$i = 2$



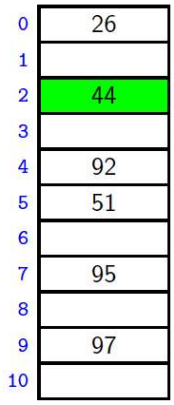
Cuckoo hashing example

$M = 11, \quad h_1(k) = k \bmod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

insert(97)

$y.key = 44$

$h_1(y.key) = 0$
$h_2(y.key) = 2$
$i = 2$



Cuckoo hashing example

$M = 11, \quad h_1(k) = k \bmod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

search(26)

$h_1(26) = 4$
$h_2(26) = 0$

0	26
1	
2	44
3	
4	92
5	51
6	
7	95
8	
9	97
10	

Cuckoo hashing example

$M = 11, \quad h_1(k) = k \bmod 11, \quad h_2(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

delete(26)

$h_1(26) = 4$
$h_2(26) = 0$

0	
1	
2	44
3	
4	92
5	51
6	
7	95
8	
9	97
10	

Complexity of open addressing strategies

For any open addressing scheme, we *must* have $\alpha < 1$ (why?).
Cuckoo hashing requires $\alpha < 1/2$.

The following gives asymptotic average-cast costs:

	search	*insert*	*delete*
Linear Probing	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{1-\alpha}$
Double Hashing	$\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)$
Cuckoo Hashing	1	$\frac{\alpha}{(1-2\alpha)^2}$	1

Summary: All operations have $O(1)$ average-case run-time if the hash-function is uniform and α is kept sufficiently small.

8. 哈希表的应用

高效的数据存储和查找均可以用哈希表

1、对等网络(P2P)中的应用

a) 基于分布式哈希表的系统

对于对等计算系统而言，能够适应的网络规模是一项非常重要的指标。然而，早期设计的系统，比如 Gnutella 和 Napster，在这方面都有一定的缺陷。前者使用的是不适合大规模系统的洪泛策略，后者引入了集中式的目录管理。

在这样的背景下，一批基于分布式哈希表的系统应时而生，包括 Tapestry[52]、Pastry[40]、Chord[47]和 Content-Addressable Networks (CAN)[39]。在这些系统中，文件根据系统生成的标识（ID）排列。这种标识通常是文件名经过哈希计算的结果。系统中的每一个结点都和一个特定区段内的标识关联，并保存相关联标识对应的文件的信息。当分布式哈希表系统对标识进行查询时，相应的结点便会返回对应的信息。

分布式哈希表系统的核心是路由协议。系统中的分布式哈希表结点构成一个覆盖网，每一个查询操作都是通过这个覆盖网找到目标结点。所以，分布式哈希表系统的性能就取决于其所采用的路由协议的效率。虽然各种分布式哈希表系统的路由协议都不相同，但它们都具有一个共同的特点，就是每一个结点在覆盖网中拥有的邻居数目为 O(LogN) 1，完成每一次路由所需步数都会在O(LogN) 内，其中 N 为系统总结点数。

b)基于分布式哈希表的关键词搜索

结构化对等计算网络都实现了分布式哈希表，并利用分布式哈希表将数据项映射到结点。上层应用可以插入一对<key, value="">到系统中，并通过 key 得到 value，哈希表在EJB中用的较多，简单的聊天室中可以靠Hash表来维持用户的数据。

2、用哈希函数压缩序数索引

在数学上将这种n位数转换为m（其中m<n< font="">）位数称为哈希转换（hashing）。哈希转换可以将一个索引器空间（indexers space）转换为哈希表（hash table）。哈希函数实现哈希转换。以社保号的例子来说，哈希函数H()表示为：H(x) = x 的后四位，哈希函数的输入可以是任意的九位社保号，而结果则是社保号的后四位数字。

数学术语中，这种将九位数转换为四位数的方法称为哈希元素映射，显然映射未必全是单设，这必将导致冲突的产生，处理冲突的相关机制不是本文探讨范围。

3、信息安全方面的应用

a)攻击路径重构

在路由器上利用哈希表记录IP报文头部信息，实现攻击路径的重构，从而追踪到攻击主机的地址。

出现的问题：1、瞬时攻击和追踪中，该方法不是适合源主机和目标主机之间跳数太多的网络。2、更新路由器或增大内存导致硬件成本提高

b)在信息加密方面的应用

利用哈希函数的非单射构造不可逆的加密算法，从而实现信息的安全传输。

4、数据库中的数据查找

由于它在记录查找时一次存取便能得到所查记录，所以在电信领域中对大型话单文件进行处理时，显示相当高的效率。例如：广东电信公用电话200话单处理中利用哈希表实现了话单统计。

相关文章

哈夫曼编码和数据压缩Java

第二步：创建新的HFMTreeNode, 在优先队列里面添加每一个Huffman Node

成都小展

Hashmap源码阅读

HashMap是什么想必大家都是知道的，日常开发中经常使用，而且常驻于笔试题目及面试中，那么今天将从源码的…

呼延十

JDK源码分析之集合04HashMap

代码改变世界-coding

实例讲解redis字符串类型

章鱼喵

HashMap源码详解

Entry 是一个 static class，其中包含了 key 和 value，也就是键值对，另外还包含了一个 next 的 Entry 指针。 我们可以...

提莫队长

redis支持的数据结构

key可以包含任意得字符，比如将一张JPEG文件内容作为key。空字符串也是一个有效的key。redis的key使用时，有一些建议：

爬蜥

散列表

是根据键 (Key) 而直接访问在内存存储位置的数据结构。也就是说，它通过计算一个关于键值的函数，将所需查询的…

周三不加班

轻松理解 Java HashMap 和 ConcurrentHas…

Map 这样的 Key Value 在软件开发中是非常经典的结构，常用于在内存中存放数据。

烂猪皮

新手村：Redis 基础补充知识

新手村的第一篇文章(新手村：最适合新手的 Redis 基础)是村民的处女作，在之后的学习中回看这篇文章，觉得有一些纰漏之处，甚是惭愧，这也是本篇文章的由来和内容...

syy

HashMap？ConcurrentHashMap？相信看完…

Map 这样的 Key Value 在软件开发中是非常经典的结构，常用于在内存中存放数据。

Java团长

更多文章

0

0