常见 Hash 算法的原理

产品名称	常见 Hash 算法的原理
公司名称	郑州易尚教育科技有限公司
价格	面议
规格参数	品牌:郑州尚学堂 型号:java培训 授课方式:面授
公司地址	郑州市金水区文化路82号硅谷广场B座9层015、016号
联系电话	0371-58500950

产品详情

散列表，它是基于高速存取的角度设计的，也是一种典型的“ 空间换时间 ”的做法。顾名思义，该数据结构能够理解为一个线性表，可是当中的元素不是紧密排列的，而是可能存在空隙。

散列表(Hash table，也叫哈希表)，是依据关键码值(Key value)而直接进行訪问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来訪问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。

比方我们存储70个元素，但我们可能为这70个元素申请了100个元素的空间。70/100=0.7，这个数字称为负载因子。我们之所以这样做，也是为了“ 高速存取 ”的目的。我们基于一种结果尽可能随机平均分布的固定函数H为每一个元素安排存储位置，这样就能够避免遍历性质的线性搜索，以达到高速存取。可是因为此随机性，也必定导致一个问题就是冲突。

所谓冲突，即两个元素通过散列函数H得到的地址同样，那么这两个元素称为“ 同义词 ”。这类似于70个人去一个有100个椅子的饭店吃饭。散列函数的计算结果是一个存储单位地址，每一个存储单位称为“ 桶 ”。设一个散列表有m个桶，则散列函数的值域应为[0,m-1]。

解决冲突是一个复杂问题。冲突主要取决于：

(1)散列函数，一个好的散列函数的值应尽可能平均分布。

(2)处理冲突方法。

(3)负载因子的大小。太大不一定就好，并且浪费空间严重，负载因子和散列函数是联动的。

解决冲突的办法：

(1)线性探查法：冲突后，线性向前试探，找到近期的一个空位置。缺点是会出现堆积现象。存取时，可能不是同义词的词也位于探查序列，影响效率。

(2)双散列函数法：在位置d冲突后，再次使用还有一个散列函数产生一个与散列表桶容量m互质的数c，依次试探(d+n*c)%m，使探查序列跳跃式分布。

经常使用的构造散列函数的方法

散列函数能使对一个数据序列的訪问过程更加迅速有效，通过散列函数，数据元素将被更快地定位：

1. 直接寻址法：取keyword或keyword的某个线性函数值为散列地址。即H(key)=key或H(key) = a·key + b，当中a和b为常数(这样的散列函数叫做自身函数)

2. 数字分析法：分析一组数据，比方一组员工的出生年月日，这时我们发现出生年月日的前几位数字大体同样，这种话，出现冲突的几率就会非常大，可是我们发现年月日的后几位表示月份和详细日期的数字区别非常大，假设用后面的数字来构成散列地址，则冲突的几率会明显减少。因此数字分析法就是找出数字的规律，尽可能利用这些数据来构造冲突几率较低的散列地址。

3. 平方取中法：取keyword平方后的中间几位作为散列地址。

4. 折叠法：将keyword切割成位数同样的几部分，最后一部分位数能够不同，然后取这几部分的叠加和(去除进位)作为散列地址。

5.
随机数法：选择一随机函数，取keyword的随机值作为散列地址，通经常使用于keyword长度不同的场合。

6. 除留余数法：取keyword被某个不大于散列表表长m的数p除后所得的余数为散列地址。即 H(key) = key MOD p, p<=m。不仅能够对keyword直接取模，也可在折叠、平方取中等运算之后取模。对p的选择非常重要，一般取素数或m，若p选的不好，easy产生同义词。

查找的性能分析

散列表的查找过程基本上和造表过程同样。一些关键码可通过散列函数转换的地址直接找到，还有一些关键码在散列函数得到的地址上产生了冲突，须要按处理冲突的方法进行查找。在介绍的三种处理冲突的方法中，产生冲突后的查找仍然是给定值与关键码进行比较的过程。所以，对散列表查找效率的量度，依旧用平均查找长度来衡量。

查找过程中，关键码的比较次数，取决于产生冲突的多少，产生的冲突少，查找效率就高，产生的冲突多，查找效率就低。因此，影响产生冲突多少的因素，也就是影响查找效率的因素。影响产生冲突多少有下面三个因素：

1. 散列函数是否均匀;

2. 处理冲突的方法;

3. 散列表的装填因子。

散列表的装填因子定义为： α = 填入表中的元素个数 / 散列表的长度

α 是散列表装满程度的标志因子。因为表长是定值， α 与“ 填入表中的元素个数 ”成正比，所以， α 越大，填入表中的元素较多，产生冲突的可能性就越大; α 越小，填入表中的元素较少，产生冲突的可能性就越小。

实际上，散列表的平均查找长度是装填因子 α 的函数，仅仅是不同处理冲突的方法有不同的函数。

了解了hash基本定义，就不能不提到一些著名的hash算法，MD5 和 SHA-1
能够说是眼下应用最广泛的Hash算法，而它们都是以 MD4 为基础设计的。那么他们都是什么意思呢?

这里简单说一下：

(1) MD4

MD4(RFC 1320)是 MIT 的 Ronald L. Rivest 在 1990 年设计的，MD 是 Message Digest
的缩写。它适用在32位字长的处理器上用快速软件实现 – 它是基于 32 位操作数的位操作来实现的。

(2) MD5

MD5(RFC 1321)是 Rivest
于1991年对MD4的改进版本号。它对输入仍以512位分组，其输出是4个32位字的级联，与 MD4
同样。MD5比MD4来得复杂，而且速度较之要慢一点，但更安全，在抗分析和抗差分方面表现更好

(3) SHA-1 及其它

SHA1是由NIST NSA设计为同DSA一起使用的，它对长度小于264的输入，产生长度为160bit的散列值，因此抗穷举(brute-force)性更好。SHA-1 设计时基于和MD4同样原理,而且模仿了该算法。

哈希表不可避免冲突(collision)现象：对不同的keyword可能得到同一哈希地址 即key1≠key2，而hash(key1)=hash(key2)。因此，在建造哈希表时不仅要设定一个好的哈希函数，并且要设定一种处理冲突的方法。可例如以下描写叙述哈希表：依据设定的哈希函数H(key)和所选中的处理冲突的方法，将一组keyword映象到一个有限的、地址连续的地址集(区间)上并以keyword在地址集中的“ 象 ”作为对应记录在表中的存储位置，这样的表被称为哈希表。

对于动态查找表而言，1) 表长不确定;2)在设计查找表时，仅仅知道keyword所属范围，而不知道确切的keyword。因此，普通情况需建立一个函数关系，以f(key)作为keyword为key的录在表中的位置，通常称这个函数f(key)为哈希函数。(注意：这个函数并不一定是数学函数)

哈希函数是一个映象，即：将keyword的集合映射到某个地址集合上，它的设置非常灵活，仅仅要这个地址集合的大小不超出同意范围就可以。

现实中哈希函数是须要构造的，而且构造的好才干使用的好。

那么这些Hash算法究竟有什么用呢?

Hash算法在信息安全方面的应用主要体如今下面的3个方面：

(1) 文件校验

我们比較熟悉的校验算法有奇偶校验和CRC校验，这2种校验并没有抗数据篡改的能力，它们一定程度上能检测并纠正传输数据中的信道误码，但却不能防止对数据的恶意破坏。

MD5 Hash算法的 ” 数字指纹 ” 特性，使它成为眼下应用最广泛的一种文件完整性校验和(Checksum)算法，不少Unix系统有提供计算md5 checksum的命令。

(2) 数字签名

Hash 算法也是现代password体系中的一个重要组成部分。因为非对称算法的运算速度较慢，所以在数字签名协议中，单向散列函数扮演了一个重要的角色。 对 Hash 值，又称 ” 数字摘要 ” 进行数字签名，在统计上能够觉得与对文件本身进行数字签名是等效的。并且这种协议还有其它的长处。

(3) 鉴权协议

例如以下的鉴权协议又被称作挑战 – 认证模式：在传输信道是可被侦听，但不可被篡改的情况下，这是一种简单而安全的方法。

文件hash值

MD5-Hash-文件的数字文摘通过Hash函数计算得到。无论文件长度怎样，它的Hash函数计算结果是一个

固定长度的数字。与加密算法不同，这一个Hash算法是一个不可逆的单向函数。採用安全性高的Hash算法，如MD5、SHA时，两个不同的文件差点儿不可能得到同样的Hash结果。因此，一旦文件被改动，就可检测出来。

Hash函数还有另外的含义。实际中的Hash函数是指把一个大范围映射到一个小范围。把大范围映射到一个小范围的目的往往是为了节省空间，使得数据easy保存。除此以外，Hash函数往往应用于查找上。所以，在考虑使用Hash函数之前，须要明确它的几个限制：

1. Hash的主要原理就是把大范围映射到小范围;所以，你输入的实际值的个数必须和小范围相当或者比它更小。不然冲突就会非常多。

2. 因为Hash逼近单向函数;所以，你能够用它来对数据进行加密。

3. 不同的应用对Hash函数有着不同的要求;比方，用于加密的Hash函数主要考虑它和单项函数的差距，而用于查找的Hash函数主要考虑它映射到小范围的冲突率。

应用于加密的Hash函数已经探讨过太多了，在作者的博客里面有更具体的介绍。所以，本文仅仅探讨用于查找的Hash函数。

Hash函数应用的主要对象是数组(比方，字符串)，而其目标通常是一个int类型。下面我们都依照这样的方式来说明。

一般的说，Hash函数能够简单的划分为例如以下几类：

1. 加法Hash;

2. 位运算Hash;

3. 乘法Hash;

4. 除法Hash;

5. 查表Hash;

6. 混合Hash;

以下具体的介绍以上各种方式在实际中的运用。

一 加法Hash

所谓的加法Hash就是把输入元素一个一个的加起来构成最后的结果。标准的加法Hash的构造例如以下：

```
static int additiveHash(String key, int prime)

{

int hash, i;

for (hash = key.length(), i = 0; i < key.length(); i++)

hash += key.charAt(i);

return (hash % prime);

}
```

这里的prime是随意的质数，看得出，结果的值域为[0,prime-1]。

二 位运算Hash

这类型Hash函数通过利用各种位运算(常见的是移位和异或)来充分的混合输入元素。比方，标准的旋转Hash的构造例如以下：

```
static int rotatingHash(String key, int prime)

{

int hash, i;

for (hash=key.length(), i=0; i

hash = (hash<>28)^key.charAt(i);

return (hash % prime);

}
```

先移位，然后再进行各种位运算是这样的类型Hash函数的主要特点。比方，以上的那段计算hash的代码还能够有例如以下几种变形：

```
hash = (hash<>27)^key.charAt(i);

hash += key.charAt(i);

hash += (hash << 10);

hash ^= (hash >> 6);

if((i&1) == 0)

{

hash ^= (hash<>3);

}
```

```
else

{

hash ^= ~((hash<>5));

}

hash += (hash<

hash = key.charAt(i) + (hash<>16) ? hash;

hash ^= ((hash<>2));
```

三 乘法Hash

这样的类型的Hash函数利用了乘法的不相关性(乘法的这样的性质，最有名的莫过于平方取头尾的随机数生成算法，尽管这样的算法效果并不好)。比方，

```
static int bernstein(String key)

{

int hash = 0;

int i;

for (i=0; i

return hash;
```

```
}
```

jdk5.0里面的String类的hashCode()方法也使用乘法Hash。只是，它使用的乘数是31。推荐的乘数还有：131, 1313, 13131, 131313等等。使用这样的方式的著名Hash函数还有：

```
// 32位FNV算法

int M_SHIFT = 0;

public int FNVHash(byte[] data)

{

int hash = (int)2166136261L;

for(byte b : data)

hash = (hash * 16777619) ^ b;

if (M_SHIFT == 0)

return hash;

return (hash ^ (hash >> M_SHIFT)) & M_MASK;

}
```

以及改进的FNV算法：public static int FNVHash1(String data)

```
{

final int p = 16777619;
```

```
int hash = (int)2166136261L;

for(int i=0;i

hash = (hash ^ data.charAt(i)) * p;

hash += hash << 13;

hash ^= hash >> 7;

hash += hash << 3;

hash ^= hash >> 17;

hash += hash << 5;

return hash;

}
```

除了乘以一个固定的数，常见的还有乘以一个不断改变的数，比方：

```
static int RSHash(String str)

{

int b = 378551;

int a = 63689;

int hash = 0;
```

```
for(int i = 0; i < str.length(); i++)

{

hash = hash * a + str.charAt(i);

a = a * b;

}

return (hash & 0x7FFFFFFF);

}
```

尽管Adler32算法的应用没有CRC32广泛，只是，它可能是乘法Hash里面最有名的一个了。关于它的介绍，大家能够去看RFC 1950规范。

四 除法Hash

除法和乘法一样，相同具有表面上看起来的不相关性。只是，由于除法太慢，这样的方式差点儿找不到真正的应用。须要注意的是，我们在前面看到的hash的 结果除以一个prime的目的仅仅是为了保证结果的范围。假设你不须要它限制一个范围的话，能够使用例如以下的代码替代 ”hash%prime ”： hash = hash ^ (hash>>10) ^ (hash>>20)。

五 查表Hash

查表Hash最有名的样例莫过于CRC系列算法。尽管CRC系列算法本身并非查表，可是，查表是它的一种最快的实现方式。以下是CRC32的实现：

```
static int crctab[256] = {
```

0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988, 0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7, 0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172, 0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59, 0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924, 0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433, 0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e, 0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65, 0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0, 0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f, 0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a, 0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1, 0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,0xjobbole

0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc, 0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b, 0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236, 0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d, 0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38, 0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777, 0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9, 0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94, 0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d

};

```
int crc32(String key, int hash)

{

int i;

for (hash=key.length(), i=0; i

hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k.charAt(i)];

return hash;

}
```

查表Hash中有名的样例有：Universal Hashing和Zobrist Hashing。他们的表格都是随机生成的。

六 混合Hash

混合Hash算法利用了以上各种方式。各种常见的Hash算法，比方MD5、Tiger都属于这个范围。它们一般非常少在面向查找的Hash函数里面使用。

七 对Hash算法的评价

http://www.burtleburtle.net/bob/hash/doobs.html
这个页面提供了对几种流行Hash算法的评价。我们对Hash函数的建议例如以下：

1. 字符串的Hash。最简单能够使用主要的乘法Hash，当乘数为33时，对于英文单词有非常好的散列效果(小于6个的小写形式能够保证没有冲突)。复杂一点能够使用FNV算法(及其改进形式)，它对于比較长的字符串，在速度和效果上都不错。

```
public override unsafe int GetHashCode()
```

```
{//微软System.String 字符串哈希算法

fixed (char* str= ((char*) this))

{

char* chPtr = str;

intnum = 0x15051505;

intnum2 = num;

int* numPtr = (int*) chPtr;

for (inti = this.Length; i > 0; i -= 4)

{

num = (((num << 5) + num) + (num >> 0x1b)) ^ numPtr[0];

if (i <= 2)

{

break;

}

num2 = (((num2 << 5) + num2) + (num2 >> 0x1b)) ^ numPtr[1];

numPtr += 2;
```

```
}

return (num + (num2 * 0x5d588b65));

}

}
```

2. 长数组的Hash。能够使用http://burtleburtle.net/bob/c/lookup3.c这样的算法，它一次运算多个字节，速度还算不错。