哈希算法

什么是哈希算法?

将任意长度的二进制值串映射为固定长度的二进制值串，这个映射的规则就是`哈希算法`，而通过原始数据映射之后得到的二进制值串就是`哈希值（散列值）`．一个优秀的哈希算法需要满足：

- 从哈希值不能反向推导出原始数据（所以哈希算法也叫单向哈希算法）；
- 对输入数据非常敏感，哪怕原始数据只修改了一个 Bit，最后得到的哈希值也大不相同；
- 散列冲突的概率要很小，对于不同的原始数据，哈希值相同的概率非常小；
- 哈希算法的执行效率要尽量高效，针对较长的文本，也能快速地计算出哈希值

散列方法的主要思想是根据结点的关键码值来确定其存储地址：以关键码值K为自变量，通过一定的函数关系h(K)(称为散列函数)，计算出对应的函数值来，把这个值解释为结点的存储地址，将结点存入到此存储单元中。检索时，用同样的方法计算地址，然后到相应的单元里去取要找的结点。通过散列方法可以对结点进行快速检索。散列（hash，也称“哈希”）是一种重要的存储方式，也是一种常见的检索方法。

Hash算法可以简单的划分为如下几类：

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；

实例实现

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的构造如下：

```
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是任意的质数，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。比如，标准的旋转Hash的构造如下：

```
static int rotatingHash(String key, int prime)
{undefined
int hash, i;
```

```
for (hash=key.length(), i=0; i<key.length(); ++i)
hash = (hash<<4)^(hash>>28)^key.charAt(i);
return (hash % prime);
}
```

先移位，然后再进行各种位运算是这种类型Hash函数的主要特点。比如，以上的那段计算hash的代码还可以有如下几种变形：

```
hash = (hash<<5)^(hash>>27)^key.charAt(i);
hash += key.charAt(i);
hash += (hash << 10);
hash ^= (hash >> 6);
if((i&1) == 0)
{undefined
hash ^= (hash<<7) ^ key.charAt(i) ^ (hash>>3);
}
else
{undefined
hash ^= ~((hash<<11) ^ key.charAt(i) ^ (hash >>5));
}
hash += (hash<<5) + key.charAt(i);
hash = key.charAt(i) + (hash<<6) + (hash>>16) – hash;
hash ^= ((hash<<5) + key.charAt(i) + (hash>>2));
```

三、乘法Hash

###

这种类型的Hash函数利用了乘法的不相关性（乘法的这种性质，最有名的莫过于平方取头尾的随机数生成算法，虽然这种算法效果并不好）。比如，

```
static int bernstein(String key)
{undefined
int hash = 0;
int i;
for (i=0; i<key.length(); ++i) hash = 33*hash + key.charAt(i);
return hash;
}
```

jdk5.0里面的String类的hashCode()方法也使用乘法Hash。不过，它使用的乘数是31。推荐的乘数还有：131, 1313, 13131, 131313等等。

使用这种方式的著名Hash函数还有：

```
//  32位FNV算法

int M_SHIFT = 0;

public int FNVHash(byte[] data)

{undefined

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)

{undefined

final int p = 16777619;

int hash = (int)2166136261L;

for(int i=0;i<data.length();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，比如：

```
static int RSHash(String str)

{undefined

int b   = 378551;

int a   = 63689;

int hash = 0;

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

{undefined

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

a   = a * b;

}

return (hash & 0x7FFFFFFF);

}
```

```
虽然Adler32算法的应用没有CRC32广泛，不过，它可能是乘法Hash里面最有名的一个了

###     四 除法Hash
```

除法和乘法一样，同样具有表面上看起来的不相关性。不过，因为除法太慢，这种方式几乎找不到真正的应用。需要注意的是，我们在前面看到的hash的 结果除以一个prime的目的只是为了保证结果的范围。如果你不需要它限制一个范围的话，可以使用如下的代码替代"hash%prime"： hash = hash ^ (hash>>10) ^ (hash>>20)

五 查表Hash

查表Hash最有名的例子莫过于CRC系列算法。虽然CRC系列算法本身并不是查表，但是，查表是它的一种最快的实现方式。下面是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,

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中有名的例子有：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，比如MD5、Tiger都属于这个范围。它们一般很少在面向查找的Hash函数里面使用。

Hash函数构造

所谓的 hash 算法就是将字符串转换为数字的算法。通常有以下几种构造 Hash 函数的方法：

2.1 直接定址法
取关键字或者关键字的某个线性函数为 Hash 地址，即address(key) = a * key + b; 如知道学生的学号从2000开始，最大为4000，则可以将address(key)=key-2000(其中a = 1)作为Hash地址。

2.2 平方取中法
对关键字进行平方计算，取结果的中间几位作为 Hash 地址。如有以下关键字序列 {421，423，436}，平方之后的结果为 {177241，178929，190096}，那么可以取中间的两位数 {72，89，00} 作为 Hash 地址。

2.3 折叠法
将关键字拆分成几部分，然后将这几部分组合在一起，以特定的方式进行转化形成Hash地址。如图书的 ISBN 号为 8903-241-23，可以将 address(key)=89+03+24+12+3 作为 Hash 地址。

2.4 除留取余法
如果知道 Hash 表的最大长度为 m，可以取不大于m的最大质数 p，然后对关键字进行取余运算，address(key)=key % p。这里 p 的选取非常关键，p 选择的好的话，能够最大程度地减少冲突，p 一般取不大于m的最大质数。

Hash表大小的确定
Hash 表的空间如果远远大于实际存储的记录数据的个数，则造成空间浪费；如果过小，则容易造成冲突。Hash 表大小确定通常有这两种思路：

如果最初知道存储的数据量，则需要根据存储个数 和 关键字的分布特点来确定 Hash 表的大小。

事先不知道最终需要存储的记录个数，需要动态维护Hash表的容量，此时可能需要重新计算 Hash 地址。

算法	输出长度（位）	输出长度（字节）
MD5	128 bits	16 bytes
SHA-1	160 bits	20 bytes
RipeMD-160	160 bits	20 bytes
SHA-256	256 bits	32 bytes
SHA-512	512 bits	64 bytes

以MD5为例

{MD5补充

> 一种可以将任意长度的输入转化为固定长度输出的算法（严格来说不能称之为一种加密算法，但是它可以达到加密的效果）

1.优点

（1）容易计算及不可逆性：

> 现在主流的编程语言基本都支持MD5算法的实现，所以非常容易计算出一个数据的MD5值。而且MD5算法是不可逆的，也 就是说我们无法通过常规的方式从MD5值倒推出它的原文。

（2）压缩性：

任意长度的数据，其MD5值都是一个32位长度的十六进制字符串，区分大小写（所以要和安卓、服务端商量好是用大写还 是小写）。

（3）抗修改性：

对原数据做一丁点的改动，MD5值就会有巨大的变动。逆反过来理解一下这个特性，比如说两个原数据的MD5值非常相似，但是你不能想当然的认为它们俩对应的原数据也相似。如果你想要很轻易的由MD5倒猜出原文数据是不可能的，因此这个特性在某种程度上表明了MD5算法是安全的。

（4）抗碰撞性：

抗碰撞性分为两种：

第一种，知道了原数据及其MD5值，想要碰撞出这个MD5值，从而猜测出原数据，是非常困难的。这种碰撞的难度表明的 是，强制破解MD5算法是非常困难的，更别说上面的轻易倒推了。

第二种，我们知道MD5值总是一个32位的十六进制字符串，换算成二进制就是一个128位的字符串，因此所有的MD5值一共有2的128次方种可能性这种碰撞的难度表明的是，我们可以放心的去使用MD5算法，不必担心不同的数据拥有相同的MD5值。依照现在计算机的计算能力，碰撞被认为在实际中是不可能发生的。因此，这个特性也在某种程度上表明了MD5算法的安全性。

2.缺点：在较短时间发现碰撞，以在对安全性要求较高的场合，不建议直接使用MD5算法。

###

Hash 表的实际应用

上述说了这么多关于 Hash 表的知识点，但是 Hash 表在代码的世界中，实际上又有什么应用场景，可能有些读者会一头雾水，这里笔者就以简单的三个例子来说明 Hash 表的实际应用场景。

1 、找出两文件找出重复的元素

假设有两个文件，文件中均包含一些短字符串，字符串个数分别为n。它们是有重复的字符串，现在需要找出所有重复的字符串。

最笨的解决办法可能是：遍历文件 1 中的每个元素，取出每一个元素分别去文件 2 中进行查找，这样的时间复杂度为O（n^2）。

但是借助 Hash 表可以有一种相对巧妙的方法，分别遍历文件 1 中的元素和文件 2 中的元素，然后放入 Hash Table 中，对于遍历的每一个元素我们只要简单的做一下计数处理即可。最后遍历整个 Hash 列表，找出所有个数大于 1 的元素即为重复的元素。

2、找出两文件找出出现次数最多的元素

同找出两文件找出重复的元素这样的问题解决方案类似，只是在最后遍历的时找计数最大的元素，即为出现次数最多的元素。

3、路由算法

多线程处理数据的场景下，通常需要将一个数据集分给不同的线程进行处理，同时要保证，相同的元素需要分到相同的处理线程上。这
其实这个就是一个很典型的 Hash 值应用场景，对于很多的计算引擎默认都是用 Hash 算法去解决这个问题。因为相同元素的 Hash 值相同，那么我们可以取 Hash 之后进行模运算，运算结果分配到不同的线程。

Hash 表的优缺点及注意点

优点

哈希表的效率非常高，查找、插入、删除操作只需要接近常量的时间即0(1）的时间级。如果需要在一秒种内查找上千条记录通常使用哈希表，哈希表的速度明显比树快，树的操作通常需要O(N)的时间级。哈希表不仅速度快，编程实现也相对容易。如果不需要遍历数据，不二的选择。

缺点

它是基于数组的，数组创建后难于扩展。有些情况下，哈希表被基本填满时，性能下降得非常严重，所以开发者必须要清楚表中将要存储的数据量。或者也可以定期地把数据转移到更大的哈希表中，不过这个过程耗时相对比较大。

注意点

在设计Hash算法的时候。一定要保证相同字符串产生的 Hash 值相同，同时要尽量的减小Hash冲突的发生，这样才算是好的 hash 算法。

Hash 冲突及解决方案

Hash冲突产生

有这样一个问题：因为我们是用数组大小对哈希值进行取模，有可能不同键值所得到的索引值相同，这里就是冲突。如在最初的实例中，如果多出了sizhang这样一个元素，那么就存在两个 756。

858	433	644	665	756	619	756
zhangsan	lisi	wanger	wangwu	zhangsi	gaofei	sizhang

显然出现的这种情况是不合理的，解决该冲突的方法就是改变数据结构。我们将数组内的元素改变为一个链表，这样就能容下足够多的元素了，冲突问题也能得到解决。具体如何解决请看下面的链地址法。

Hash 冲突解决

开放定址法

发生冲突时，使用某种探测技术在 Hash 表中形成一个探测序列，然后沿着这个探测序列依次查找下去，当碰到一个空的单元时，则插入其中。比较常用的探测方法有线性探测法，如有一组关键字{12，13，25，23，38，34，6，84，91}，Hash 表长为14，Hash 函数为 address(key) = key % 11，当插入12，13，25时可以直接插入，而当插入 23 时，地址 1 被占用了（因为 12%11 和 23%11 的结果相同）。

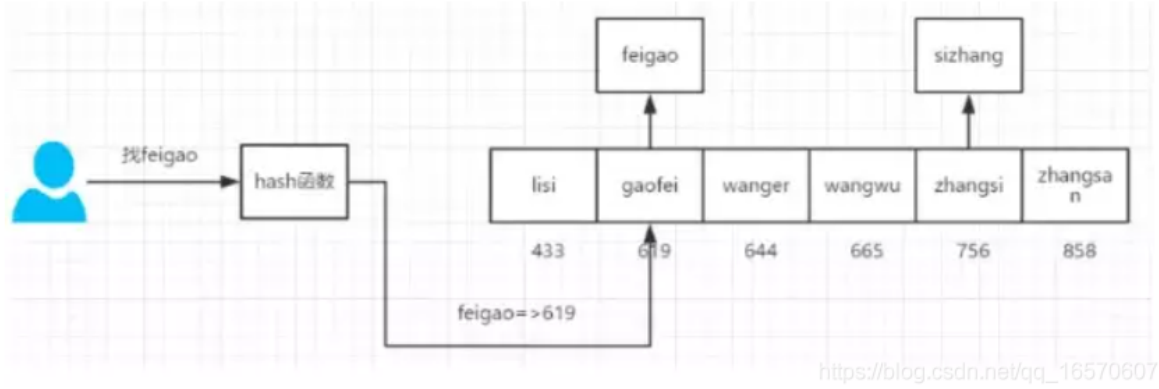
此时沿着地址 1 依次往下探测(探测步长可以根据情况而定)，直到探测到地址4，发现为空，则将 23 插入其中。

链地址法

采用数组和链表相结合的数据结构，将 Hash 地址相同的记录存储在一张线性表中，而每张表的表头的序号即为计算得到的Hash地址。如下图最左边是数组结构，数组内的元素为链表结构。

采用链地址法形成的 Hash 表存储形式

所以针对之前案列冲突的解决方案如下：



检索的时候可以这样检索，首先找到gaofei后，之后再遍历链表，找到feigao了。同理对于 sizhang 的冲突也是如此解决。