Hash 函数的设计优化

天津南开中学 李羽修

【摘要】

Hash 是一种在信息学竞赛中经常用到的数据结构。一个好的 Hash 函数可以很大程度上提高程序的整体时间效率和空间效率。本文对面向各种不同标本（关键值）的 Hash 函数进行讨论，并对多种常用的 Hash 函数进行了分析和总结。

【关键字】

Hash 函数，字符串，整数，实数，排列组合

【正文】

对于一个 Hash 函数，评价其优劣的标准应为随机性，即对任意一组标本，进入 Hash 表每一个单元（cell）之概率的平均程度，因为这个概率越平均，数据在表中的分布就越平均，表的空间利用率就越高。由于在竞赛中，标本的性质是无法预知的，因此数学推理将受到很大限制。我们用实验的方法研究这个随机性。

一、整数的 **Hash** 函数

常用的方法有三种：直接取余法、乘积取整法、平方取中法。下面我们对这三种方法分别进行讨论。以下假定我们的关键字是 k ，Hash 表的容量是 M ，Hash 函数为 $h(k)$ 。

1．直接取余法

我们用关键字 k 除以 M ，取余数作为在 Hash 表中的位置。函数表达式可以写成:

$$h(k) = k \bmod M \text{ 。}$$

例如，表容量 $M = 12$，关键值 $k = 100$，那么 $h(k) = 4$。该方法的好处是实现容易且速度快，是很常用的一种方法。但是如果 M 选择的不好而偏偏标本又很特殊，那么数据在 Hash 中很容易扎堆而影响效率。

对于 M 的选择，在经验上，我们一般选择不太接近 2^n 的一个素数；如果关键字的值域较小，我们一般在此值域 1.1~1.6 倍范围内选择。例如 k 的值域为 $[0,600]$，那么 $M = 701$ 即为一个不错的选择。竞赛的时候可以写一个素数生成器或干脆自己写一个“比较像素数”的数。

我用 4000 个数插入一个容量为 701 的 Hash 表，得到的结果是:

测试数据	随机数据	连续数据

最小单元容量:	0	5
最大单元容量:	15	6
期望容量:	5.70613	5.70613
标准差:	2.4165	0.455531

可见对于随机数据，取余法的最大单元容量达到了期望容量的将近 3 倍。经测试，在我的机器（Pentium III 866MHz，128MB RAM）上，该函数的运行时间大约是 39ns，即大约 35 个时钟周期。

2．乘积取整法

我们用关键字 k 乘以一个在 $(0,1)$ 中的实数 A （最好是无理数），得到一个 $(0,k)$ 之间的实数；取出其小数部分，乘以 M ，再取整数部分，即得 k 在 Hash 表中的位置。函数表达式可以写成:

$$h(k)=\lfloor M(kA \bmod 1)\rfloor ;$$

其中 $kA \bmod 1$ 表示 kA 的小数部分，即 $kA-\lfloor kA\rfloor$ 。例如，表容量 $M=12$ ，种子 $A=\frac{\sqrt{5}-1}{2}$ （ $A=\frac{\sqrt{5}-1}{2}$ 是一个实际效果很好的选择），关键值 $k=100$ ，那么 $h(k)=9$ 。

同样用 4000 个数插入一个容量为 701 的 Hash 表（ $A=\frac{\sqrt{5}-1}{2}$ ），得到的结果是:

测试数据	随机数据	连续数据
最小单元容量:	0	4
最大单元容量:	15	7
期望容量:	5.70613	5.70613
标准差:	2.5069	0.619999

从公式中可以看出，这个方法受 M 的影响是很小的，在 M 的值比较不适合直接取余法的时候这个方法的表现很好。但是从上面的测试来看，其表现并不是非常理想，且由于浮点运算较多，运行速度较慢。经反复优化，在我的机器上仍需 892ns 才能完成一次计算，即 810 个时钟周期，是直接取余法的 23 倍。

3．平方取中法

我们把关键字 k 平方，然后取中间的 $\lfloor \log_2 M\rfloor$ 位作为 Hash 函数值返回。由于 k 的每一位都会对其平方中间的若干位产生影响，因此这个方法的效果也是不错的。但是对于比较小的 k 值效果并不是很理想，实现起来也比较繁琐。为了充分利用 Hash 表的空间， M 最好取 2 的整数次幂。例如，表容量 $M=2^4=16$ ，关键值 $k=100$ ，那么 $h(k)=8$ 。

用 4000 个数插入一个容量为 512 的 Hash 表（注意这里没有用 701，是为了利用 Hash 表的空间），得到的结果是:

测试数据	随机数据	连续数据

最小单元容量:	0	1
最大单元容量:	17	17
期望容量:	7.8125	7.8125
标准差:	2.95804	2.64501

效果比我们想象的要差，尤其是对于连续数据。但由于只有乘法和位运算，该函数的速度是最快的。在我的机器上，一次运算只需要 23ns，即 19 个时钟周期，比直接取余法还要快一些。

比较一下这三种方法:

	实现难度	实际效果	运行速度	其他应用
直接取余法	易	好	较快	字符串
乘积取整法	较易	较好	慢	浮点数
平方取中法	中	较好	快	无

从这个表格中我们很容易看出，直接取余法的性价比是最高的，因此也是我们竞赛中用得最多的一种方法。

对于实数的 Hash 函数，我们可以直接利用乘积取整法；而对于标本为其他类型数据的 Hash 函数，我们可以先将其转换为整数，然后再将其插入 Hash 表。下面我们来研究把其他类型数据转换成整数的方法。

二、字符串的 **Hash** 函数

字符串本身就可以看成一个 256 进制（ANSI 字符串为 128 进制）的大整数，因此我们可以利用直接取余法，在线性时间内直接算出 Hash 函数值。为了保证效果， M 仍然不能选择太接近 2^n 的数；尤其是当我们把字符串看成一个 2^p 进制数的时候，选择 $M = 2^p - 1$ 会使得该字符串的任意一个排列的 Hash 函数值都相同。（想想看，为什么？）

常用的字符串 Hash 函数还有 ELFHash，APHash 等等，都是十分简单有效的方法。这些函数使用位运算使得每一个字符都对最后的函数值产生影响。另外还有以 MD5 和 SHA1 为代表的杂凑函数，这些函数几乎不可能找到碰撞（MD5 前一段时间才刚刚被破解）。

我从 Mark Twain 的一篇小说中分别随机抽取了 1000 个不同的单词和 1000 个不同的句子，作为短字符串和长字符串的测试数据，然后用不同的 Hash 函数把它们变成整数，再用直接取余法插入一个容量为 1237 的 Hash 表，遇到冲突则用新字符串覆盖旧字符串。通过观察最后“剩下”的字符串的个数，我们可以粗略地得出不同的 Hash 函数实际效果。

	短字符串	长字符串	平均	编码难度
直接取余数	667	676	671.5	易
P. J. Weinberger Hash	683	676	679.5	难
ELF Hash	683	676	679.5	较难
SDBM Hash	694	680	687.0	易
BKDR Hash	665	710	687.5	较易
DJB Hash	694	683	688.5	较易

AP Hash	684	698	691.0	较难
RS Hash	691	693	692.0	较难
JS Hash	684	708	696.0	较易

把 1000 个随机数用直接取余法插入容量为 1237 的 Hash 表，其覆盖单元数也只达到了 694，可见后面的几种方法已经达到了极限，随机性相当优秀。然而我们却很难选择，因为不存在完美的、既简单又实用的解决方案。我一般选择 JS Hash 或 SDBM Hash 作为字符串的 Hash 函数。这两个函数的代码如下：

```
unsigned int JSHash(char *str)
{
    unsigned int hash = 1315423911; // nearly a prime - 1315423911
= 3 * 438474637
    while (*str)
    {
        hash ^= ((hash << 5) + (*str++) + (hash >> 2));
    }
    return (hash & 0x7FFFFFFF);
}

unsigned int SDBMHash(char *str)
{
    unsigned int hash = 0;
    while (*str)
    {
        // equivalent to: hash = 65599*hash + (*str++);
        hash = (*str++) + (hash << 6) + (hash << 16) - hash;
    }
    return (hash & 0x7FFFFFFF);
}
```

JSHash 的运算比较复杂，如果对效果要求不是特别高的话 SDBMHash 是一个很好的选择。

三、排列的 **Hash** 函数

准确的说，这里我们的研究不再仅仅局限在“Hashing”的工作，而是进化到一个“numerize”的过程，也就是说我们可以在排列和 1 到 A_n^m 的自然数之间建立一一对应的关系。这样我们就可以利用这个关系来直接定址，或者用作 Hash 函数；在基于状态压缩的动态规划算法中也能用上。

1．背景知识

自然数的 p 进制表示法我们已经很熟悉了，即：

$$n=\sum_{k=0}^{m}a_k p^k \quad , \quad 0\le a_k \le p$$

比如 $p=2$ 便是二进制数， $p=10$ 便是十进制数。

引理： $\forall n\in N^*$， $n!=1+\sum_{k=1}^{n-1}k\cdot k!$ 。

证明：对 n 使用数学归纳法。

1) $n=1$ 时，等式显然成立。

2)假设 $n=m$ 时等式成立，即 $m!=1+\sum_{k=1}^{m-1}k\cdot k!$ 。

则 $n=m+1$ 时，

$$n!=(m+1)!=(m+1)m!=m\cdot m!+m!=m\cdot m!+1+\sum_{k=1}^{m-1}k\cdot k!=1+\sum_{k=1}^{m}k\cdot k!$$

即 $n=m+1$ 时等式亦成立。

3)综上所述， $\forall n\in N^*$， $n!=1+\sum_{k=1}^{n-1}k\cdot k!$ 成立。

把这个式子变形一下：

$$n!-1=(n-1)(n-1)!+(n-2)(n-2)!+\cdots+2\cdot 2!+1\cdot 1!$$

上式和 $p^n-1=(p-1)p^{n-1}+(p-1)p^{n-2}+\cdots+(p-1)$ 类似。不难证明，从 0 到 $n!-1$ 的任何自然数 m 可唯一地表示为

$$m=a_{n-1}(n-1)!+a_{n-2}(n-2)!+\cdots+a_1 1!$$

其中 $0\le a_i\le i$， $i=1,2,\ldots,n-1$ 。甚至在式子后面加上一个 $a_0 0!$ 也无妨，在后面我们把这一项忽略掉。所以从 0 到 $n!-1$ 的 $n!$ 个自然数与

$$(a_{n-1},a_{n-2},a_{n-3},\ldots,a_2,a_1) \qquad (*)$$

一一对应。另一方面，不难从 m 算出 $a_{n-1},a_{n-2},a_{n-3},\ldots,a_2,a_1$ 。

我们可以把序列 $(a_{n-1},a_{n-2},a_{n-3},\ldots,a_2,a_1)$ 理解为一个“变进制数”，也就是第一位二进制，第二位三进制，……，第 i 位 $i+1$ 进制，……，第 $n-1$ 位 n 进制。这样，我们就可以方便的使用类似“除 p 取余法”的方法从一个自然数 m 算出序列 $(a_{n-1},a_{n-2},a_{n-3},\ldots,a_2,a_1)$ 。由于这样的序列共有 $n!$ 个，我们很自然的想到把这 $n!$ 个序列和 n 个元素的全排列建立一一对应。

2．全排列与自然数之一一对应

为了方便起见，不妨设 n 个元素为 $1,2,\ldots,n$ 。对应的规则如下：设序列（*）对应的某一排列 $p=p_1p_2\cdots p_n$，其中 a_i 可以看做是排列 p 中数 $i+1$ 所在位置右边比 $i+1$ 小的数的个数。以排列 4213 为例，它是元素 1,2,3,4 的一个排列。4 的右边比 4 小的数的数目为 3，所以 $a_3=3$。3 右边比 3 小的数的数目为 0，即 $a_2=0$。同理 $a_1=1$。所以排列 4213 对应于变进制的 301，也就是十进制的 19；反过来也可以从 19 反推到 301，再反推到排列 4213。

3．更一般性的排列

受到这个思路启发，我们同样可以把更一般性的排列与自然数之间建立一一对应关系。想一想从 n 个元素中选 m 个的排列数 A_n^m 的公式是怎么来的？根据乘法原理，我们有

$$A_n^m=n(n-1)(n-2)\cdots(n-m+1)$$

这是由于在排列的第 1 个位置有 n 种选择，在排列的第 2 个位置有 $n-1$ 种选择，……，在排列的第 m 个位置有 $n-m+1$ 种选择。既然这样，我们可以定义一种“m-n 变进制数”，使其第 1 位是 $n-m+1$ 进制，第 2 位是 $n-m+2$ 进制，……，第 m 位是 n 进制。这样，0 到 A_n^m-1 之间的任意一个自然数 k 都可以唯一地表示成：

$$k=a_mA_{n-1}^{m-1}+a_{m-1}A_{n-2}^{m-2}+\cdots+a_2A_{n-m+1}^1+a_1$$

其中 $0\le a_i\le n-m+i-1$，$1\le i\le m$。注意到 $A_n^m-1=\sum_{k=1}^{m}(n-m+k-1)A_{n-m+k-1}^{k-1}$（证明略，可直接变形结合前面的引理推得），所以从 0 到 A_n^m-1 的 A_n^m 个自然数可以与序列

$$(a_m,a_{m-1},a_{m-2},\ldots,a_2,a_1)$$

一一对应。类似地，可以用取余法从自然数 k 算出 $a_m,a_{m-1},a_{m-2},\ldots,a_2,a_1$。

我们设 n 个元素为 $1,2,\ldots,n$，从中取出 m 个。对应关系如下：维护一个首元素下标为 0 的线性表 L，初始时 $L=(1,2,\ldots,n)$。对于某一排列 $p=p_1p_2\cdots p_m$，我们从 p_1 开始处理。首先在 L 中找到 p_1 的下标记为 a_m，然后删除 $L[a_m]$；接着在 L 中找到 p_2 的下标记为 a_{m-1}，然后删除 $L[a_{m-1}]$……直到 $L[a_1]$ 被删除为止。以在 5 个元素 1,2,3,4,5 中取出 2,4,3 为例，这时 $n=5,m=3$。首先在 L 中取出 2，记下 $a_3=1$，L 变为 1,3,4,5；在 L 中取出 4，记下 $a_2=2$，L 变为 1,3,5；在 L 中取出 3，记下 $a_1=1$，L 变为 1,5。因此排

列 243 对应于“3-5 变进制数”121，即十进制数 19；反过来也可以从十进制数 19 反推到 121，再反推到排列 243。各序列及其对应的排列如下表：

$p_1p_2p_3$	$a_3a_2a_1$	N	$p_1p_2p_3$	$a_3a_2a_1$	N
123	000	0	341	220	30
124	001	1	342	221	31
125	002	2	345	222	32
132	010	3	351	230	33
134	011	4	352	231	34
135	012	5	354	232	35
142	020	6	412	300	36
143	021	7	413	301	37
145	022	8	415	302	38
152	030	9	421	310	39
153	031	10	423	311	40
154	032	11	425	312	41
213	100	12	431	320	42
214	101	13	432	321	43
215	102	14	435	322	44
231	110	15	451	330	45
234	111	16	452	331	46
235	112	17	453	332	47
241	120	18	512	400	48
243	121	19	513	401	49
245	122	20	514	402	50
251	130	21	521	410	51
253	131	22	523	411	52
254	132	23	524	412	53
312	200	24	531	420	54
314	201	25	532	421	55
315	202	26	534	422	56
321	210	27	541	430	57
324	211	28	542	431	58
325	212	29	543	432	59

【总结】

本文对几个常用的 Hash 函数进行了总结性的介绍和分析，并将其延伸到应用更加广泛的“与自然数建立一一对应”的过程。Hash 是一种相当有效的数据结构，充分体现了“空间换时间”的思想。在如今竞赛中内存限制越来越松的情况下，要做到充分利用内存空间来换取宝贵的时间，Hash 能够给我们很大帮助。我们应当根据题目的特点，选择适合题目的数据结构来优化算法。对于组合与自然数的一一对应关系，我还没有想到好的方法，欢迎大家讨论。

【参考文献】

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Riverst, Clifford Stein. *Introduction to Algorithms.* Second Edition. The MIT Press, 2001

[2] 刘汝佳，黄亮.《算法艺术与信息学竞赛》.北京：清华大学出版社，2004

[3] 卢开澄，卢华明.《组合数学》（第3版）.北京：清华大学出版社，2002

【附录】

常用的字符串Hash函数之源代码：

```
// RS Hash Function
unsigned int RSHash(char *str)
{
    unsigned int b = 378551;
    unsigned int a = 63689;
    unsigned int hash = 0;

    while (*str)
    {
        hash = hash * a + (*str++);
        a *= b;
    }

    return (hash & 0x7FFFFFFF);
}

// JS Hash Function
unsigned int JSHash(char *str)
{
    unsigned int hash = 1315423911;

    while (*str)
    {
        hash ^= ((hash << 5) + (*str++) + (hash >> 2));
    }

    return (hash & 0x7FFFFFFF);
}

// P. J. Weinberger Hash Function
unsigned int PJWHash(char *str)
{
    unsigned int BitsInUnignedInt = (unsigned int)(sizeof(unsigned
int) * 8);
```

```
    unsigned   int   ThreeQuarters            =   (unsigned   int)
((BitsInUnignedInt  * 3) / 4);
    unsigned  int  OneEighth                   =  (unsigned  int)
(BitsInUnignedInt / 8);
    unsigned int HighBits          = (unsigned int)(0xFFFFFFFF) <<
(BitsInUnignedInt - OneEighth);
    unsigned int hash             = 0;
    unsigned int test             = 0;

    while (*str)
    {
        hash = (hash << OneEighth) + (*str++);
        if ((test = hash & HighBits) != 0)
        {
            hash = ((hash ^ (test >> ThreeQuarters)) & (~HighBits));
        }
    }

    return (hash & 0x7FFFFFFF);
}

// ELF Hash Function
unsigned int ELFHash(char *str)
{
    unsigned int hash = 0;
    unsigned int x    = 0;

    while (*str)
    {
        hash = (hash << 4) + (*str++);
        if ((x = hash & 0xF0000000L) != 0)
        {
            hash ^= (x >> 24);
            hash &= ~x;
        }
    }

    return (hash & 0x7FFFFFFF);
}

// BKDR Hash Function
unsigned int BKDRHash(char *str)
{
```

```
    unsigned int seed = 131; // 31 131 1313 13131 131313 etc..
    unsigned int hash = 0;

    while (*str)
    {
        hash = hash * seed + (*str++);
    }

    return (hash & 0x7FFFFFFF);
}

// SDBM Hash Function
unsigned int SDBMHash(char *str)
{
    unsigned int hash = 0;

    while (*str)
    {
        hash = (*str++) + (hash << 6) + (hash << 16) - hash;
    }

    return (hash & 0x7FFFFFFF);
}

// DJB Hash Function
unsigned int DJBHash(char *str)
{
    unsigned int hash = 5381;

    while (*str)
    {
        hash += (hash << 5) + (*str++);
    }

    return (hash & 0x7FFFFFFF);
}

// AP Hash Function
unsigned int APHash(char *str)
{
    unsigned int hash = 0;
    int i;
```

```
    for (i=0; *str; i++)
    {
        if ((i & 1) == 0)
        {
            hash ^= ((hash << 7) ^ (*str++) ^ (hash >> 3));
        }
        else
        {
            hash ^= (~((hash << 11) ^ (*str++) ^ (hash >> 5)));
        }
    }

    return (hash & 0x7FFFFFFF);
}
```