(11条消息)经典字符串hash函数介绍及性能比较及最佳算法-------bkdrhash算法解析及扩展_Vincent的博客-CSDN博客

字符串Hash函数对比

今天根据自己的理解重新整理了一下几个字符串hash函数，使用了模板，使其支持宽字符串，代码如下：

```
/// @brief BKDR Hash Function
/// @detail 本算法由于在Brian Kernighan与Dennis Ritchie的《The C Programming Language》一书被展示而得名，是一种简单快捷的hash算法，也是Java目前采用的字符串的Hash算法（累乘因子为31）。
template<class T>
size_t BKDRHash(const T *str)
{
register size_t hash = 0;
while (size_t ch = (size_t)*str++)
    {
        hash = hash * 131 + ch;   // 也可以乘以31、131、1313、13131、131313..
// 有人说将乘法分解为位运算及加减法可以提高效率，如将上式表达为：hash = hash << 7 + hash << 1 + hash + ch;
// 但其实在Intel平台上，CPU内部对二者的处理效率都是差不多的，
// 我分别进行了100亿次的上述两种运算，发现二者时间差距基本为0（如果是Debug版，分解成位运算后的耗时还要高1/3）；
// 在ARM这类RISC系统上没有测试过，由于ARM内部使用Booth's Algorithm来模拟32位整数乘法运算，它的效率与乘数有关：
// 当乘数8-31位都为1或0时，需要1个时钟周期
// 当乘数16-31位都为1或0时，需要2个时钟周期
// 当乘数24-31位都为1或0时，需要3个时钟周期
// 否则，需要4个时钟周期
// 因此，虽然我没有实际测试，但是我依然认为二者效率上差别不大
    }
return hash;
}
/// @brief SDBM Hash Function
/// @detail 本算法是由于在开源项目SDBM（一种简单的数据库引擎）中被应用而得名，它与BKDRHash思想一致，只是种子不同而已。
template<class T>
size_t SDBMHash(const T *str)
{
register size_t hash = 0;
while (size_t ch = (size_t)*str++)
    {
        hash = 65599 * hash + ch;
//hash = (size_t)ch + (hash << 6) + (hash << 16) - hash;
    }
return hash;
}
/// @brief RS Hash Function
/// @detail 因Robert Sedgwicks在其《Algorithms in C》一书中展示而得名。
template<class T>
size_t RSHash(const T *str)
{
register size_t hash = 0;
size_t magic = 63689;
while (size_t ch = (size_t)*str++)
    {
        hash = hash * magic + ch;
        magic *= 378551;
    }
return hash;
}
/// @brief AP Hash Function
/// @detail 由Arash Partow发明的一种hash算法。
template<class T>
size_t APHash(const T *str)
{
register size_t hash = 0;
size_t ch;
for (long i = 0; ch = (size_t)*str++; i++)
    {
```

```
58. if ((i & 1) == 0)
59.         {
60.             hash ^= ((hash << 7) ^ ch ^ (hash >> 3));
61.         }
62. else
63.         {
64.             hash ^= (~((hash << 11) ^ ch ^ (hash >> 5)));
65.         }
66.     }
67. return hash;
68. }
69. /// @brief JS Hash Function
70. /// 由Justin Sobel发明的一种hash算法。
71. template<class T>
72. size_t JSHash(const T *str)
73. {
74. if(!*str)        // 这是由本人添加，以保证空字符串返回哈希值0
75. return 0;
76. register size_t hash = 1315423911;
77. while (size_t ch = (size_t)*str++)
78.     {
79.         hash ^= ((hash << 5) + ch + (hash >> 2));
80.     }
81. return hash;
82. }
83. /// @brief DEK Function
84. /// @detail 本算法是由于Donald E. Knuth在《Art Of Computer Programming Volume 3》中展示而得名。
85. template<class T>
86. size_t DEKHash(const T* str)
87. {
88. if(!*str)        // 这是由本人添加，以保证空字符串返回哈希值0
89. return 0;
90. register size_t hash = 1315423911;
91. while (size_t ch = (size_t)*str++)
92.     {
93.         hash = ((hash << 5) ^ (hash >> 27)) ^ ch;
94.     }
95. return hash;
96. }
97. /// @brief FNV Hash Function
98. /// @detail Unix system系统中使用的一种著名hash算法，后来微软也在其hash_map中实现。
99. template<class T>
100. size_t FNVHash(const T* str)
101. {
102. if(!*str)   // 这是由本人添加，以保证空字符串返回哈希值0
103. return 0;
104. register size_t hash = 2166136261;
105. while (size_t ch = (size_t)*str++)
106.     {
107.         hash *= 16777619;
108.         hash ^= ch;
109.     }
110. return hash;
111. }
112. /// @brief DJB Hash Function
113. /// @detail 由Daniel J. Bernstein教授发明的一种hash算法。
114. template<class T>
115. size_t DJBHash(const T *str)
116. {
117. if(!*str)   // 这是由本人添加，以保证空字符串返回哈希值0
118. return 0;
119. register size_t hash = 5381;
120. while (size_t ch = (size_t)*str++)
121.     {
122.         hash += (hash << 5) + ch;
```

```
    }
return hash;
}
/// @brief DJB Hash Function 2
/// @detail 由Daniel J. Bernstein 发明的另一种hash算法。
template<class T>
size_t DJB2Hash(const T *str)
{
if(!*str)   // 这是由本人添加，以保证空字符串返回哈希值0
return 0;
register size_t hash = 5381;
while (size_t ch = (size_t)*str++)
    {
        hash = hash * 33 ^ ch;
    }
return hash;
}
/// @brief PJW Hash Function
/// @detail 本算法是基于AT&T贝尔实验室的Peter J. Weinberger的论文而发明的一种hash算法。
template<class T>
size_t PJWHash(const T *str)
{
static const size_t TotalBits       = sizeof(size_t) * 8;
static const size_t ThreeQuarters   = (TotalBits  * 3) / 4;
static const size_t OneEighth       = TotalBits / 8;
static const size_t HighBits        = ((size_t)-1) << (TotalBits - OneEighth);

register size_t hash = 0;
size_t magic = 0;
while (size_t ch = (size_t)*str++)
    {
        hash = (hash << OneEighth) + ch;
if ((magic = hash & HighBits) != 0)
        {
            hash = ((hash ^ (magic >> ThreeQuarters)) & (~HighBits));
        }
    }
return hash;
}
/// @brief ELF Hash Function
/// @detail 由于在Unix的Extended Library Function被附带而得名的一种hash算法，它其实就是PJW Hash的变形。
template<class T>
size_t ELFHash(const T *str)
{
static const size_t TotalBits       = sizeof(size_t) * 8;
static const size_t ThreeQuarters   = (TotalBits  * 3) / 4;
static const size_t OneEighth       = TotalBits / 8;
static const size_t HighBits        = ((size_t)-1) << (TotalBits - OneEighth);
register size_t hash = 0;
size_t magic = 0;
while (size_t ch = (size_t)*str++)
    {
        hash = (hash << OneEighth) + ch;
if ((magic = hash & HighBits) != 0)
        {
            hash ^= (magic >> ThreeQuarters);
            hash &= ~magic;
        }
    }
return hash;
}
```

我对这些hash的散列质量及效率作了一个简单测试，测试结果如下：

测试1：对100000个由大小写字母与数字随机的ANSI字符串（无重复，每个字符串最大长度不超过64字符）进行散列：

字符串函数	冲突数	除1000003取余后的冲突数
BKDRHash	0	4826

SDBMHash	2	4814
RSHash	2	4886
APHash	0	4846
ELFHash	1515	6120
JSHash	779	5587
DEKHash	863	5643
FNVHash	2	4872
DJBHash	832	5645
DJB2Hash	695	5309
PJWHash	1515	6120

测试2：对100000个由任意UNICODE组成随机字符串（无重复，每个字符串最大长度不超过64字符）进行散列：

字符串函数	冲突数	除1000003取余后的冲突数
BKDRHash	3	4710
SDBMHash	3	4904
RSHash	3	4822
APHash	2	4891
ELFHash	16	4869
JSHash	3	4812
DEKHash	1	4755
FNVHash	1	4803
DJBHash	1	4749
DJB2Hash	2	4817
PJWHash	16	4869

测试3：对1000000个随机ANSI字符串（无重复，每个字符串最大长度不超过64字符）进行散列：

字符串函数	耗时（毫秒）
BKDRHash	109
SDBMHash	109
RSHash	124
APHash	187
ELFHash	249
JSHash	172
DEKHash	140
FNVHash	125
DJBHash	125
DJB2Hash	125
PJWHash	234

结论：也许是我的样本存在一些特殊性，在对ASCII码字符串进行散列时，PJW与ELF Hash（它们其实是同一种算法）无论是质量还是效率，都相当糟糕；例如："b5"与"aE"，这两个字符串按照PJW散列出来的hash值就是一样的。 另外，其它几种依靠异或来散列的哈希函数，如：JS/DEK/DJB Hash，在对字母与数字组成的字符串的散列效果也不怎么好。相对而言，还是BKDR与SDBM这类简单的Hash效率与效果更好。

其他：

作者：icefireelf

出处：http://blog.csdn.net/icefireelf/article/details/5796529

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。

常用字符串哈希函数有 BKDRHash，APHash，DJBHash，JSHash，RSHash，SDBMHash，PJWHash，ELFHash等等。对于以上几种哈 希函数，我对其进行了一个小小的评测。

Hash函数	数据1	数据2	数据3	数据4	数据1得分	数据2得分	数据3得分	数据4得分	平均分
BKDRHash	2	0	4774	481	96.55	100	90.95	82.05	92.64
APHash	2	3	4754	493	96.55	88.46	100	51.28	86.28
DJBHash	2	2	4975	474	96.55	92.31	0	100	83.43
JSHash	1	4	4761	506	100	84.62	96.83	17.95	81.94
RSHash	1	0	4861	505	100	100	51.58	20.51	75.96
SDBMHash	3	2	4849	504	93.1	92.31	57.01	23.08	72.41
PJWHash	30	26	4878	513	0	0	43.89	0	21.95
ELFHash	30	26	4878	513	0	0	43.89	0	21.95

其中数据1为100000个字母和数字组成的随机串哈希冲突个数。数据2为100000个有意义的英文句子哈希冲突个数。数据3为数据1的哈希值与 1000003(大素数)求模后存储到线性表中冲突的个数。数据4为数据1的哈希值与10000019(更大素数)求模后存储到线性表中冲突的个数。

经过比较，得出以上平均得分。平均数为平方平均数。可以发现，BKDRHash无论是在实际效果还是编码实现中，效果都是最突出的。APHash也 是较为优秀的算法。DJBHash,JSHash,RSHash与SDBMHash各有千秋。PJWHash与ELFHash效果最差，但得分相似，其算 法本质是相似的。

```
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);
}

// 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)
```

```
72.     {
73.         hash = (hash << 4) + (*str++);
74. if ((x = hash & 0xF0000000L) != 0)
75.         {
76.             hash ^= (x >> 24);
77.             hash &= ~x;
78.         }
79.     }
80.
81. return (hash & 0x7FFFFFFF);
82. }
83.
84. // BKDR Hash Function
85. unsigned int BKDRHash(char *str)
86. {
87. unsigned int seed = 131; // 31 131 1313 13131 131313 etc..
88. unsigned int hash = 0;
89.
90. while (*str)
91.     {
92.         hash = hash * seed + (*str++);
93.     }
94.
95. return (hash & 0x7FFFFFFF);
96. }
97.
98. // DJB Hash Function
99. unsigned int DJBHash(char *str)
100. {
101. unsigned int hash = 5381;
102.
103. while (*str)
104.     {
105.         hash += (hash << 5) + (*str++);
106.     }
107.
108. return (hash & 0x7FFFFFFF);
109. }
110.
111. // AP Hash Function
112. unsigned int APHash(char *str)
113. {
114. unsigned int hash = 0;
115. int i;
116.
117. for (i=0; *str; i++)
118.     {
119. if ((i & 1) == 0)
120.         {
121.             hash ^= ((hash << 7) ^ (*str++) ^ (hash >> 3));
122.         }
123. else
124.         {
125.             hash ^= (~((hash << 11) ^ (*str++) ^ (hash >> 5)));
126.         }
127.     }
128.
129. return (hash & 0x7FFFFFFF);
130. }
```

编程珠玑中的一个hash函数

```
1.
2. //用跟元素个数最接近的质数作为散列表的大小
3. #define NHASH 29989
4. #define MULT 31
5.
6. unsigned in hash(char *p)
7. {
8. unsigned int h = 0;
9. for (; *p; p++)
10.         h = MULT *h + *p;
11. return h % NHASH;
```

出处：http://blog.csdn.net/wanglx_/article/details/40300363