应用Hash函数（java描述）

2013-07-19 21:27 1308人阅读 评 报

分类： java编程（16）

作者：冲处宇宙
时间：2007.1.25

计算理论中，没有Hash函数的说法，只有单向函数的说法。所谓的单向函数，是一个复杂的定去看计算理论或者密码学方面的数据。用“人类”的语言描述单向函数就是：**如果某个函数在给定输入的时候，很容易计算出其结果来；而当给定结果的时候，很难计算出输入来，这就是单项函数。**各种加密函数都可以被认为是单向函数的逼近。Hash函数（或者成为散列函数）也可以看成是单向函数的一个逼近。即它接近于满足单向函数的定义。

Hash函数还有另外的含义。实际中的H**ash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，使得数据容易保存。**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，需要明白它的几个限制：

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。不然冲突就会很多。
2. 由于Hash逼近单向函数；所以，你可以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；比如，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详细的介绍。所以，本文只探讨用于查找的Hash函数。

Hash函数应用的主要对象是数组（比如，字符串），而其目标一般是一个int类型。以下我们都按照这种方式来说明。

一般的说，Hash函数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；
下面详细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的构造如下：

```java
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：

```java
static int rotatingHash(String key, int prime)
{
  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：

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

```java
static int bernstein(String key)
{
  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：

```java
//  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：

```java
```

```
    public static int FNVHash1(String data)
    {
        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：

```
[java]
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);
```


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。关于它的介绍，大家可以去看RFC 1950规范。

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：

```
[java]
static int crctab[256] = {
0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f,  0xe963a535, 0x9e6495
  0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,  0xf9b9df6f, 0x8ebe
};
int crc32(String key, int hash)
{
  int i;
  for (hash=key.length(), i=0; i<key.length(); ++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。

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函数的建议如下：

1. 字符串的Hash。最简单可以使用基本的乘法Hash，当乘数为33时，对于英文单词有很好的散列效果（小于6个的小写形式可以保证没有冲突）。复杂一点可以使用FNV算法（及其改进形式），它对于比较长的字符串，在速度和效果上都不错。

2. 长数组的Hash。可以使用http://burtleburtle.net/bob/c/lookup3.c这种算法，它一次运算多个字节，速度还算不错。

八 后记

本文简略的介绍了一番实际应用中的用于查找的Hash算法。Hash算法除了应用于这个方面以外，另外一个著名的应用是巨型字符串匹配（这时的Hash算法叫做：rolling hash，因为它必须可以滚动的计算）。设计好的Hash算法并不是一件容易的事情。做为应用来说，选择一个适合的算法是最重要的。

常用hash算法类：


```java
package lotusroots.algorithms.math;

import java.security.MessageDigest;

/**
 * Hash算法大全<br>
 * 推荐使用FNV1算法
 *
 * @algorithm None
 * @author Goodzzp 2006-11-20
 * @lastEdit Goodzzp 2006-11-20
 * @editDetail Create
 */
public class HashAlgorithms {
 /**
  * 加法hash
  *
  * @param key
  *            字符串
  * @param prime
  *            一个质数
  * @return hash结果
  */
 public 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);
 }

 /**
  * 旋转hash
  *
  * @param key
  *            输入字符串
  * @param prime
  *            质数
  * @return hash值
  */
 public static int rotatingHash(String key, int prime) {
  int hash, i;
  for (hash = key.length(), i = 0; i < key.length(); ++i)
   hash = (hash << 4) ^ (hash >> 28) ^ key.charAt(i);
  return (hash % prime);
  // return (hash ^ (hash>>10) ^ (hash>>20));
 }

 // 替代:
 // 使用: hash = (hash ^ (hash>>10) ^ (hash>>20)) & mask;
 // 替代: hash %= prime;

 /**
  * MASK值，随便找一个值，最好是质数
  */
 static int M_MASK = 0x8765fed1;

 /**
  * 一次一个hash
  *
```

```
60.   * @param key
61.   *            输入字符串
62.   * @return 输出hash值
63.   */
64.  public static int oneByOneHash(String key) {
65.   int hash, i;
66.   for (hash = 0, i = 0; i < key.length(); ++i) {
67.    hash += key.charAt(i);
68.    hash += (hash << 10);
69.    hash ^= (hash >> 6);
70.   }
71.   hash += (hash << 3);
72.   hash ^= (hash >> 11);
73.   hash += (hash << 15);
74.   // return (hash & M_MASK);
75.   return hash;
76.  }
77.
78.  /**
79.   * Bernstein's hash
80.   *
81.   * @param key
82.   *            输入字节数组
83.   * @param level
84.   *            初始hash常量
85.   * @return 结果hash
86.   */
87.  public static int bernstein(String key) {
88.   int hash = 0;
89.   int i;
90.   for (i = 0; i < key.length(); ++i)
91.    hash = 33 * hash + key.charAt(i);
92.   return hash;
93.  }
94.
95.  //
96.  // // Pearson's Hash
97.  // char pearson(char[]key, ub4 len, char tab[256])
98.  // {
99.  // char hash;
100. // ub4 i;
101. // for (hash=len, i=0; i<len; ++i)
102. // hash=tab[hash^key[i]];
103. // return (hash);
104. // }
105.
106. // // CRC Hashing，计算crc,具体代码见其他
107. // ub4 crc(char *key, ub4 len, ub4 mask, ub4 tab[256])
108. // {
109. // ub4 hash, i;
110. // for (hash=len, i=0; i<len; ++i)
111. // hash = (hash >> 8) ^ tab[(hash & 0xff) ^ key[i]];
112. // return (hash & mask);
113. // }
114.
115. /**
116.  * Universal Hashing
117.  */
118. public static int universal(char[] key, int mask, int[] tab) {
119.  int hash = key.length, i, len = key.length;
120.  for (i = 0; i < (len << 3); i += 8) {
121.   char k = key[i >> 3];
122.   if ((k & 0x01) == 0)
123.    hash ^= tab[i + 0];
124.   if ((k & 0x02) == 0)
125.    hash ^= tab[i + 1];
126.   if ((k & 0x04) == 0)
127.    hash ^= tab[i + 2];
128.   if ((k & 0x08) == 0)
129.    hash ^= tab[i + 3];
130.   if ((k & 0x10) == 0)
131.    hash ^= tab[i + 4];
132.   if ((k & 0x20) == 0)
133.    hash ^= tab[i + 5];
134.   if ((k & 0x40) == 0)
135.    hash ^= tab[i + 6];
136.   if ((k & 0x80) == 0)
137.    hash ^= tab[i + 7];
138.  }
```

```
139.  return (hash & mask);
140. }
141.
142. /**
143.  * Zobrist Hashing
144.  */
145. public static int zobrist(char[] key, int mask, int[][] tab) {
146.  int hash, i;
147.  for (hash = key.length, i = 0; i < key.length; ++i)
148.   hash ^= tab[i][key[i]];
149.  return (hash & mask);
150. }
151.
152. // LOOKUP3
153. // 见Bob Jenkins(3).c文件
154.
155. // 32位FNV算法
156. static int M_SHIFT = 0;
157.
158. /**
159.  * 32位的FNV算法
160.  *
161.  * @param data
162.  *            数组
163.  * @return int值
164.  */
165. public static int FNVHash(byte[] data) {
166.  int hash = (int) 2166136261L;
167.  for (byte b : data)
168.   hash = (hash * 16777619) ^ b;
169.  if (M_SHIFT == 0)
170.   return hash;
171.  return (hash ^ (hash >> M_SHIFT)) & M_MASK;
172. }
173.
174. /**
175.  * 改进的32位FNV算法1
176.  *
177.  * @param data
178.  *            数组
179.  * @return int值
180.  */
181. public static int FNVHash1(byte[] data) {
182.  final int p = 16777619;
183.  int hash = (int) 2166136261L;
184.  for (byte b : data)
185.   hash = (hash ^ b) * p;
186.  hash += hash << 13;
187.  hash ^= hash >> 7;
188.  hash += hash << 3;
189.  hash ^= hash >> 17;
190.  hash += hash << 5;
191.  return hash;
192. }
193.
194. /**
195.  * 改进的32位FNV算法1
196.  *
197.  * @param data
198.  *            字符串
199.  * @return int值
200.  */
201. public static int FNVHash1(String data) {
202.  final int p = 16777619;
203.  int hash = (int) 2166136261L;
204.  for (int i = 0; i < data.length(); i++)
205.   hash = (hash ^ data.charAt(i)) * p;
206.  hash += hash << 13;
207.  hash ^= hash >> 7;
208.  hash += hash << 3;
209.  hash ^= hash >> 17;
210.  hash += hash << 5;
211.  return hash;
212. }
213.
214. /**
215.  * Thomas Wang的算法，整数hash
216.  */
217. public static int intHash(int key) {
```

```
    key += ~(key << 15);
    key ^= (key >>> 10);
    key += (key << 3);
    key ^= (key >>> 6);
    key += ~(key << 11);
    key ^= (key >>> 16);
    return key;
  }

  /**
   * RS算法hash
   */
  public 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);
  }

  /* End Of RS Hash Function */

  /**
   * JS算法
   */
  public static int JSHash(String str) {
    int hash = 1315423911;

    for (int i = 0; i < str.length(); i++) {
      hash ^= ((hash << 5) + str.charAt(i) + (hash >> 2));
    }

    return (hash & 0x7FFFFFFF);
  }

  /* End Of JS Hash Function */

  /**
   * PJW算法
   */
  public static int PJWHash(String str) {
    int BitsInUnsignedInt = 32;
    int ThreeQuarters = (BitsInUnsignedInt * 3) / 4;
    int OneEighth = BitsInUnsignedInt / 8;
    int HighBits = 0xFFFFFFFF << (BitsInUnsignedInt - OneEighth);
    int hash = 0;
    int test = 0;

    for (int i = 0; i < str.length(); i++) {
      hash = (hash << OneEighth) + str.charAt(i);

      if ((test = hash & HighBits) != 0) {
        hash = ((hash ^ (test >> ThreeQuarters)) & (~HighBits));
      }
    }

    return (hash & 0x7FFFFFFF);
  }

  /* End Of P. J. Weinberger Hash Function */

  /**
   * ELF算法
   */
  public static int ELFHash(String str) {
    int hash = 0;
    int x = 0;

    for (int i = 0; i < str.length(); i++) {
      hash = (hash << 4) + str.charAt(i);
      if ((x = (int) (hash & 0xF0000000L)) != 0) {
        hash ^= (x >> 24);
        hash &= ~x;
      }
```

```
297.    }
298.
299.    return (hash & 0x7FFFFFFF);
300.   }
301.
302.   /* End Of ELF Hash Function */
303.
304.   /**
305.    * BKDR算法
306.    */
307.   public static int BKDRHash(String str) {
308.    int seed = 131; // 31 131 1313 13131 131313 etc..
309.    int hash = 0;
310.
311.    for (int i = 0; i < str.length(); i++) {
312.     hash = (hash * seed) + str.charAt(i);
313.    }
314.
315.    return (hash & 0x7FFFFFFF);
316.   }
317.
318.   /* End Of BKDR Hash Function */
319.
320.   /**
321.    * SDBM算法
322.    */
323.   public static int SDBMHash(String str) {
324.    int hash = 0;
325.
326.    for (int i = 0; i < str.length(); i++) {
327.     hash = str.charAt(i) + (hash << 6) + (hash << 16) - hash;
328.    }
329.
330.    return (hash & 0x7FFFFFFF);
331.   }
332.
333.   /* End Of SDBM Hash Function */
334.
335.   /**
336.    * DJB算法
337.    */
338.   public static int DJBHash(String str) {
339.    int hash = 5381;
340.
341.    for (int i = 0; i < str.length(); i++) {
342.     hash = ((hash << 5) + hash) + str.charAt(i);
343.    }
344.
345.    return (hash & 0x7FFFFFFF);
346.   }
347.
348.   /* End Of DJB Hash Function */
349.
350.   /**
351.    * DEK算法
352.    */
353.   public static int DEKHash(String str) {
354.    int hash = str.length();
355.
356.    for (int i = 0; i < str.length(); i++) {
357.     hash = ((hash << 5) ^ (hash >> 27)) ^ str.charAt(i);
358.    }
359.
360.    return (hash & 0x7FFFFFFF);
361.   }
362.
363.   /* End Of DEK Hash Function */
364.
365.   /**
366.    * AP算法
367.    */
368.   public static int APHash(String str) {
369.    int hash = 0;
370.
371.    for (int i = 0; i < str.length(); i++) {
372.     hash ^= ((i & 1) == 0) ? ((hash << 7) ^ str.charAt(i) ^ (hash >> 3)) : (~
((hash << 11) ^ str.charAt(i) ^ (hash >> 5)));
373.    }
374.
```

```
375.    // return (hash & 0x7FFFFFFF);
376.    return hash;
377.   }
378.
379.   /* End Of AP Hash Function */
380.
381.   /**
382.    * JAVA自己带的算法
383.    */
384.   public static int java(String str) {
385.    int h = 0;
386.    int off = 0;
387.    int len = str.length();
388.    for (int i = 0; i < len; i++) {
389.     h = 31 * h + str.charAt(off++);
390.    }
391.    return h;
392.   }
393.
394.   /**
395.    * 混合hash算法，输出64位的值
396.    */
397.   public static long mixHash(String str) {
398.    long hash = str.hashCode();
399.    hash <<= 32;
400.    hash |= FNVHash1(str);
401.    return hash;
402.   }
403.
404.   /**
405.    * 计算sha1
406.    *
407.    * @param text
408.    *            文本
409.    * @return 字节数组
410.    * @throws Exception
411.    */
412.   public static byte[] sha1(String text) throws Exception {
413.    MessageDigest md;
414.    md = MessageDigest.getInstance("SHA-1");
415.    byte[] sha1hash = new byte[40];
416.    byte[] input = text.getBytes("utf-8");
417.    md.update(input, 0, input.length);
418.    sha1hash = md.digest();
419.    return sha1hash;
420.   }
421.
422.   // 4位值对应16进制字符
423.   static char[] m_byteToHexChar = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b',
424.
425.   /**
426.    * 计算sha1
427.    *
428.    * @param text
429.    *            文本
430.    * @return 16进制表示的hash值
431.    * @throws Exception
432.    */
433.   public static String sha1_text(String text) throws Exception {
434.    byte[] hash = sha1(text);
435.    StringBuilder ret = new StringBuilder(hash.length * 2);
436.    for (byte b : hash) {
437.     int d = (b & 0xff);
438.     ret.append(m_byteToHexChar[(d & 0xf)]);
439.     d >>= 4;
440.     ret.append(m_byteToHexChar[(d & 0xf)]);
441.    }
442.    return ret.toString();
443.   }
444.  }
```

Linux 常用命令使用方法大搜刮上一篇：
用java实现生产者和消费者问题下一篇：