<https://blog.csdn.net/weiyuefei/article/details/52597455>

* [简介](http://blog.csdn.net/eaglex/article/details/6310727#%E7%AE%80%E4%BB%8B)
* [哈希方法学](http://blog.csdn.net/eaglex/article/details/6310727#%E5%93%88%E5%B8%8C%E6%96%B9%E6%B3%95%E5%AD%A6)
* [哈希函数和素数](http://blog.csdn.net/eaglex/article/details/6310727#%E5%93%88%E5%B8%8C%E5%87%BD%E6%95%B0%E5%92%8C%E7%B4%A0%E6%95%B0)
* [位偏向](http://blog.csdn.net/eaglex/article/details/6310727#%E4%BD%8D%E5%81%8F%E5%90%91)
* [各种形式的哈希](http://blog.csdn.net/eaglex/article/details/6310727#%E5%90%84%E7%A7%8D%E5%BD%A2%E5%BC%8F%E7%9A%84%E5%93%88%E5%B8%8C)
* [常用的哈希函数](http://blog.csdn.net/eaglex/article/details/6310727#%E5%B8%B8%E7%94%A8%E7%9A%84%E5%93%88%E5%B8%8C%E5%87%BD%E6%95%B0)
* [各版本哈希代码下载](http://blog.csdn.net/eaglex/article/details/6310727#%E4%B8%8B%E8%BD%BD)

**简介**

哈稀函数按照定义可以实现一个伪随机数生成器(PRNG)，从这个角度可以得到一个公认的结论：哈希函数之间性能的比较可以通过比较其在伪随机生成方面的比较来衡量。

一些常用的分析技术，例如泊松分布可用于分析不同的哈希函数对不同的数据的碰撞率(collision rate)。一般来说，对任意一类的数据存在一个理论上完美的哈希函数。这个完美的哈希函数定义是没有发生任何碰撞，这意味着没有出现重复的散列值。在现实中它很难找到一个完美的哈希散列函数，而且这种完美函数的趋近变种在实际应用中的作用是相当有限的。在实践中人们普遍认识到，一个完美哈希函数的哈希函数，就是在一个特定的数据集上产生的的碰撞最少哈希的函数。

现在的问题是有各种类型的数据，有一些是高度随机的，有一些有包含高纬度的图形结构，这些都使得找到一个通用的哈希函数变得十分困难，即使是某一特定类型的数据，找到一个比较好的哈希函数也不是意见容易的事。我们所能做的就是通过试错方法来找到满足我们要求的哈希函数。可以从下面两个角度来选择哈希函数：

1.数据分布

 一个衡量的措施是考虑一个哈希函数是否能将一组数据的哈希值进行很好的分布。要进行这种分析，需要知道碰撞的哈希值的个数，如果用链表来处理碰撞，则可以分析链表的平均长度，也可以分析散列值的分组数目。

2.哈希函数的效率

另个一个衡量的标准是哈希函数得到哈希值的效率。通常，包含哈希函数的算法的算法复杂度都假设为O(1)，这就是为什么在哈希表中搜索数据的时间复杂度会被认为是"平均为O(1)的复杂度"，而在另外一些常用的数据结构，比如图(通常被实现为红黑树)，则被认为是O(logn)的复杂度。

一个好的哈希函数必修在理论上非常的快、稳定并且是可确定的。通常哈希函数不可能达到O(1)的复杂度，但是哈希函数在字符串哈希的线性的搜索中确实是非常快的，并且通常哈希函数的对象是较小的主键标识符，这样整个过程应该是非常快的，并且在某种程度上是稳定的。

在这篇文章中介绍的哈希函数被称为简单的哈希函数。它们通常用于散列（哈希字符串）数据。它们被用来产生一种在诸如哈希表的关联容器使用的key。这些哈希函数不是密码安全的，很容易通过颠倒和组合不同数据的方式产生完全相同的哈希值。

**哈希方法学**

哈希函数通常是由他们产生哈希值的方法来定义的，有两种主要的方法：

1.基于加法和乘法的散列

这种方式是通过遍历数据中的元素然后每次对某个初始值进行加操作，其中加的值和这个数据的一个元素相关。通常这对某个元素值的计算要乘以一个素数。

2.基于移位的散列

和加法散列类似，基于移位的散列也要利用字符串数据中的每个元素，但是和加法不同的是，后者更多的而是进行位的移位操作。通常是结合了左移和右移，移的位数的也是一个素数。每个移位过程的结果只是增加了一些积累计算，最后移位的结果作为最终结果。

**哈希函数和素数**

没有人可以证明素数和伪随机数生成器之间的关系，但是目前来说最好的结果使用了素数。伪随机数生成器现在是一个统计学上的东西，不是一个确定的实体，所以对其的分析只能对整个的结果有一些认识，而不能知道这些结果是怎么产生的。如果能进行更具体的研究，也许我们能更好的理解哪些数值比较有效，为什么素数比其他数更有效，为什么有些素数就不行，如果能用可再现的证明来回答这些问题，那么我们就能设计出更好的伪随机数生成器，也可能得到更好的哈希函数。

围绕着哈希函数中的素数的使用的基本的概念是，利用一个素质来改变处理的哈希函数的状态值，而不是使用其他类型的数。处理这个词的意思就是对哈希值进行一些简单的操作，比如乘法和加法。这样得到的一个新的哈希值一定要在统计学上具有更高的熵，也就是说不能有为偏向。简单的说，当你用一个素数去乘一堆随机数的时候，得到的数在bit这个层次上是1的概率应该接近0.5。没有具体的证明这种不便向的现象只出现在使用素数的情况下，这看上去只是一个自我宣称的直觉上的理论，并被一些业内人士所遵循。

决定什么是正确的，甚至更好的方法和对散列素数的使用最好的组合仍然是一个很有黑色艺术。没有单一的方法可以宣称自己是最终的通用散列函数。最好的一所能做的就是通过试错演进和获得适当的散列算法，以满足其需要的统计分析方法。

**位偏向**

位序列发生器是纯粹随机的或者说在某种程度上确定性的，可以按照一定的概率产生某种状态或相反状态的比特，这个概率就是位偏向。在纯粹随机的情况下，产生高位或者低位的位偏向应该是50%。

然后在伪随机产生器中，算法将决定在产生器在最小输出模块的位偏向。

假设一个PRNG的产生8位作为其输出块。出于某种原因，MSB始终是设置为高，MSB的位偏向将是100％的概率被置高。这一结论是，即使有256个本PRNG的产生可能的值，值小于128将永远不会产生。为简单起见，假设其他位正在生成纯粹是随机的，那么有平等的机会，128和255之间的任何值将产生，但是在同一时间，有0％的机会，一个小于128的值会产生。

所有PRNGs，无论是杂凑函数，密码，msequences或其他任何产生比特流的产生器都会有这样一个位偏向。大多数PRNGs他们将尝试收敛位偏向到一个确定值，流密码就是一个例子，而其他产生器在不确定的位偏向下效果更好。

混合或位序列加扰是一种产生在一个共同的平等流位偏向的方法。虽然我们必须要小心，以确保他们不会混合至发散位偏向。密码学中的一个混合使用的形式被称为雪崩，这就是一个位块使用用另一个块来替换或置换混合在一起，而另一块产生与其他快混合的输出。

正如下图中显示的，雪崩过程始于一个或多个二进制数据块。对数据中的某些位操作（通常是一些输入敏感位入减少位逻辑）生产的第i层片数据。然后重复这个过程是在第i层数据，以生成一个i+1个层数据，是当前层的位数将小于或等于前层的位数。

这一反复的过程将导致一个依靠之前数据所有位的位。应该指出的是，下图是一个单纯的概括，雪崩过程不一定是这一进程的唯一形式。

**各种形式的哈希**

哈希是一个在现实世界中将数据映射到一个标识符的工具，下面是哈希函数的一些常用领域：

1.字符串哈希

在数据存储领域，主要是数据的索引和对容器的结构化支持，比如哈希表。

2.加密哈希

用于数据/用户核查和验证。一个强大的加密哈希函数很难从结果再得到原始数据。加密哈希函数用于哈希用户的密码，用来代替密码本身存在某个服务器撒很难过。加密哈希函数也被视为不可逆的压缩功能，能够代表一个信号标识的大量数据，可以非常有用的判断当前的数据是否已经被篡改(比如MD5)，也可以作为一个数据标志使用，以证明了通过其他手段加密文件的真实性。

3.几何哈希

这个哈希表用于在计算机视觉领域，为在任意场景分类物体的探测。

*最初选择的过程涉及一个地区或感兴趣的对象。从那里使用，如Harris角检测器（HCD的），尺度不变特征变换（SIFT）或速成式的强大功能（冲浪），一组功能的仿射提取这被视为代表仿射不变特征检测算法表示对象或地区。这一套有时被称为宏观功能或功能的星座。对发现的功能的性质和类型的对象或地区被列为它可能仍然是可能的匹配两个星座的特点，即使可能有轻微的差异（如丢失或异常特征）两集。星座，然后说是功能分类设置。  
哈希值是计算从星座的特性。这通常是由最初定义一个地方的哈希值是为了居住空间中完成- 在这种情况下，散列值是一个多层面的价值，定义的空间正常化。再加上计算的哈希值另一个进程，决定了两个哈希值之间的距离是必要的过程-一个距离测量是必需的，而不是一个确定性的平等经营者由于对星座的哈希值计算到了可能的差距问题。也因为简单的欧氏距离度量的本质上是无效的，其结果是自动确定特定空间的距离度量已成为学术界研究的活跃领域处理这类空间的非线性性质。  
几何散列包括各种汽车分类的重新检测中任意场景的目的，典型的例子。检测水平可以多种多样，从刚检测是否是车辆，到特定型号的车辆，在特定的某个车辆。*

*4.布隆过滤器*

布隆过滤器允许一个非常大范围内的值被一个小很多的内存锁代表。在计算机科学，这是众所周知的关联查询，并在关联容器的核心理念。

Bloom Filter的实现通过多种不同的hash函数使用，也可通过允许一个特定值的存在有一定的误差概率会员查询结果的。布隆过滤器的保证提供的是，对于任何会员国的查询就永远不会再有假阴性，但有可能是假阳性。假阳性的概率可以通过改变控制为布隆过滤器，并通过不同的hash函数的数量所使用的表的大小。  
随后的研究工作集中在的散列函数和哈希表以及Mitzenmacher的布隆过滤器等领域。建议对这种结构，在数据被散列熵最实用的用法有助于哈希函数熵，这是理论成果上缔结一项最佳的布隆过滤器（一个提供给定一个最低的进一步导致假阳性的可能性表的大小或反之亦然）提供假阳性的概率定义用户可以建造最多也作为两种截然不同的两两独立的哈希散列函数已知功能，大大提高了查询效率的成员。  
布隆过滤器通常存在于诸如拼写检查器，字符串匹配算法，网络数据包分析工具和网络/ Internet缓存的应用程序。

**常用的哈希函数**

通用的哈希函数库有下面这些混合了加法和一位操作的字符串哈希算法。下面的这些算法在用法和功能方面各有不同，但是都可以作为学习哈希算法的实现的例子。(其他版本代码实现见[下载](http://blog.csdn.net/eaglex/article/details/6310727#%E4%B8%8B%E8%BD%BD)）

1.RS

从Robert Sedgwicks的 Algorithms in C一书中得到了。我(原文作者)已经添加了一些简单的优化的算法，以加快其散列过程。

**[java]** [view plain](http://blog.csdn.net/eaglex/article/details/6310727) [copy](http://blog.csdn.net/eaglex/article/details/6310727)

1. **public** **long** RSHash(String str)
2. {
3. **int** b     = 378551;
4. **int** a     = 63689;
5. **long** hash = 0;
6. **for**(**int** i = 0; i < str.length(); i++)
7. {
8. hash = hash \* a + str.charAt(i);
9. a    = a \* b;
10. }
11. **return** hash;
12. }

2.JS

Justin Sobel写的一个位操作的哈希函数。

**[c-sharp]** [view plain](http://blog.csdn.net/eaglex/article/details/6310727) [copy](http://blog.csdn.net/eaglex/article/details/6310727)

1. **public** **long** JSHash(String str)
2. {
3. **long** hash = 1315423911;
4. **for**(**int** i = 0; i < str.length(); i++)
5. {
6. hash ^= ((hash << 5) + str.charAt(i) + (hash >> 2));
7. }
8. **return** hash;
9. }

3.PJW

该散列算法是基于贝尔实验室的彼得J温伯格的的研究。在Compilers一书中（原则，技术和工具），建议采用这个算法的散列函数的哈希方法。

**[java]** [view plain](http://blog.csdn.net/eaglex/article/details/6310727) [copy](http://blog.csdn.net/eaglex/article/details/6310727)

1. **public** **long** PJWHash(String str)
2. {
3. **long** BitsInUnsignedInt = (**long**)(4 \* 8);
4. **long** ThreeQuarters     = (**long**)((BitsInUnsignedInt  \* 3) / 4);
5. **long** OneEighth         = (**long**)(BitsInUnsignedInt / 8);
6. **long** HighBits          = (**long**)(0xFFFFFFFF) << (BitsInUnsignedInt - OneEighth);
7. **long** hash              = 0;
8. **long** test              = 0;
9. **for**(**int** i = 0; i < str.length(); i++)
10. {
11. hash = (hash << OneEighth) + str.charAt(i);
12. **if**((test = hash & HighBits)  != 0)
13. {
14. hash = (( hash ^ (test >> ThreeQuarters)) & (~HighBits));
15. }
16. }
17. **return** hash;
18. }

4.ELF

和PJW很相似，在Unix系统中使用的较多。

**[java]** [view plain](http://blog.csdn.net/eaglex/article/details/6310727) [copy](http://blog.csdn.net/eaglex/article/details/6310727)

1. **public** **long** ELFHash(String str)
2. {
3. **long** hash = 0;
4. **long** x    = 0;
5. **for**(**int** i = 0; i < str.length(); i++)
6. {
7. hash = (hash << 4) + str.charAt(i);
8. **if**((x = hash & 0xF0000000L) != 0)
9. {
10. hash ^= (x >> 24);
11. }
12. hash &= ~x;
13. }
14. **return** hash;
15. }

5.BKDR

这个算法来自Brian Kernighan 和 Dennis Ritchie的 The C Programming Language。这是一个很简单的哈希算法,使用了一系列奇怪的数字,形式如31,3131,31...31,看上去和DJB算法很相似。(参照我之前一篇博客，这个就是Java的字符串哈希函数)

**[java]** [view plain](http://blog.csdn.net/eaglex/article/details/6310727) [copy](http://blog.csdn.net/eaglex/article/details/6310727)

1. **public** **long** BKDRHash(String str)
2. {
3. **long** seed = 131; // 31 131 1313 13131 131313 etc..
4. **long** hash = 0;
5. **for**(**int** i = 0; i < str.length(); i++)
6. {
7. hash = (hash \* seed) + str.charAt(i);
8. }
9. **return** hash;
10. }

6.SDBM

这个算法在开源的SDBM中使用，似乎对很多不同类型的数据都能得到不错的分布。

**[java]** [view plain](http://blog.csdn.net/eaglex/article/details/6310727) [copy](http://blog.csdn.net/eaglex/article/details/6310727)

1. **public** **long** SDBMHash(String str)
2. {
3. **long** hash = 0;
4. **for**(**int** i = 0; i < str.length(); i++)
5. {
6. hash = str.charAt(i) + (hash << 6) + (hash << 16) - hash;
7. }
8. **return** hash;
9. }

7.DJB

这个算法是Daniel J.Bernstein 教授发明的，是目前公布的最有效的哈希函数。

**[java]** [view plain](http://blog.csdn.net/eaglex/article/details/6310727) [copy](http://blog.csdn.net/eaglex/article/details/6310727)

1. **public** **long** DJBHash(String str)
2. {
3. **long** hash = 5381;
4. **for**(**int** i = 0; i < str.length(); i++)
5. {
6. hash = ((hash << 5) + hash) + str.charAt(i);
7. }
8. **return** hash;
9. }

8.DEK

由伟大的Knuth在《编程的艺术 第三卷》的第六章排序和搜索中给出。

**[java]** [view plain](http://blog.csdn.net/eaglex/article/details/6310727) [copy](http://blog.csdn.net/eaglex/article/details/6310727)

1. **public** **long** DEKHash(String str)
2. {
3. **long** hash = str.length();
4. **for**(**int** i = 0; i < str.length(); i++)
5. {
6. hash = ((hash << 5) ^ (hash >> 27)) ^ str.charAt(i);
7. }
8. **return** hash;
9. }

9.AP

这是本文作者Arash Partow贡献的一个哈希函数，继承了上面以旋转以为和加操作。代数描述：

**[java]** [view plain](http://blog.csdn.net/eaglex/article/details/6310727) [copy](http://blog.csdn.net/eaglex/article/details/6310727)

1. **public** **long** APHash(String str)
2. {
3. **long** hash = 0xAAAAAAAA;
4. **for**(**int** i = 0; i < str.length(); i++)
5. {
6. **if** ((i & 1) == 0)
7. {
8. hash ^= ((hash << 7) ^ str.charAt(i) \* (hash >> 3));
9. }
10. **else**
11. {
12. hash ^= (~((hash << 11) + str.charAt(i) ^ (hash >> 5)));
13. }
14. }
15. **return** hash;
16. }

[这里](http://blog.csai.cn/user3/50125/archives/2009/35638.html)有一个关于这些算法的评测，可以稍微看看，自己也可以简单测试下，我在VSM试验中的测试，这些算法没有太大的性能差异，可能是数据量较小的缘故。

**各版本哈希代码下载**

#### [General Hash Function Source Code (C)](http://www.partow.net/downloads/GeneralHashFunctions_-_C.zip" \t "_blank)

#### [General Hash Function Source Code (C++)](http://www.partow.net/downloads/GeneralHashFunctions_-_CPP.zip" \t "_blank)

#### [General Hash Function Source Code (Pascal & Object Pascal)](http://www.partow.net/downloads/GeneralHashFunctions_-_Object_Pascal.zip" \t "_blank)

#### [General Hash Function Source Code (Java)](http://www.partow.net/downloads/GeneralHashFunctions_-_Java.zip" \t "_blank)

#### [General Hash Function Source Code (Ruby)](http://www.partow.net/downloads/GeneralHashFunctions_-_Ruby.zip" \t "_blank)

#### [General Hash Function Source Code (Python)](http://www.partow.net/downloads/GeneralHashFunctions_-_Python.zip" \t "_blank)

#### [General Hash Function Source Code (All Languages)](http://www.partow.net/downloads/GeneralHashFunctions.zip" \t "_blank)

#### [Open Bloom Filter Source Code (C++)](http://www.partow.net/downloads/OpenBloomFilter.zip" \t "_blank)

#### [Bloom Filter Source Code (Object Pascal)](http://www.partow.net/downloads/BloomFilter_-_Object_Pascal.zip" \t "_blank)

#### [Selecting a Hashing Algorithm (Bruce J. McKenzie, R. Harries, Timothy C. Bell)](http://www.partow.net/downloads/SelectingAHashingAlgorithm.zip" \t "_blank)

* [**Cryptographic Hash Functions : A Survey (S. Bakhtiari, R. Safavi-Naini, J. Pieprzyk)**](http://www.partow.net/downloads/CryptographicHashFunctions-ASurvey.zip)