经典数据结构教科书中，“表”是数据结构的一个大家族。其中，有顺序表（数组）、单向链表、双向链表、循环链表等等。我们今天聊的不是这些，而是“表”中的异类——哈希表(Hash Table)。

**为啥需要哈希表**

为什么会有[哈希表](http://www.nowamagic.net/academy/tag/%E5%93%88%E5%B8%8C%E8%A1%A8)这种数据结构呢？让我们用一个通俗的例子来理解：

大家一定都查过字典吧，我们知道，《新华字典》是按照读音排序的，可以理解为一个以读音为key，按升序排列的数据库。对于读音已知的字，可以通过“二分查找法”，很快地查找到要找的字，其时间复杂度为O(log2n)。但是，对于不知道读音的字怎么办呢？如果使用“顺序查找”，一页一页地翻字典，假设一本新华字典600页，每翻查一页的时间开销为0.5分钟，那么，每查到一个字耗费的时间t的数学期望值E(t) = 600 \* 0.5min / 2 =150min，也就是查一个字需要两个半小时！当然，这是难以接受的。

为了解决这个问题，《新华字典》的编辑们，很快就想出了解决办法，那就是在字典的前面加入一个“检字表”，如“四角号码检字表”“部首检字表”等，其特点是以每个字的字形为依据，计算出一个索引值，并映射到对应的页数。比如“法”字，按[四角号码检字法](http://zh.wikipedia.org/zh-cn/%E5%9B%9B%E8%A7%92%E5%8F%B7%E7%A0%81)，其索引值为34131，再根据这个数值，就可以找到相应的字了。在这种情况下，查找算法的时间复杂度接近于O(1)。换句话说，字典再厚，也不会明显地影响到查[字典](http://www.nowamagic.net/academy/tag/%E5%AD%97%E5%85%B8)的效率了。

* 在[Linux内核中的hash与bucket](http://localhost/NowaMagic/academy/detail/3008086)这个条目里，我们谈到了，哈希表能够实现高效的数据存储和查找，这是其它数据结构很难替代的。计算机里大数据比比皆是，比如说你每本书都像字典那么厚，你还能用传统的查找方法一页页翻吗？这种效率很难让人接受的。

**哈希表举例**

好，让我们回到计算机的世界中来。

哈希表的最大特点，是数据存储位置（偏移量）和数据记录的内容相关，存在着一个函数换算关系：

|  |  |
| --- | --- |
| 1 | Offset = Hash (Key) |

其中，Offset为数据存储的偏移量，Hash为散列函数(Hash Function)，Content为数据记录内容的关键字(Hash Key)。假设，我们要建立一张NowaMagic全球访问量统计表，每条记录包含下面的数据结构：

|  |  |
| --- | --- |
| 1 | struct access\_record\_t { |
| 2 | unsigned index\_i; /\* Index\*/ | |

|  |  |  |
| --- | --- | --- |
| 3 | charcountry\_name[MAX\_COUNTRY\_NAME\_LEN]; /\* 国家/地区名 \*/ | |
| 4 | unsigned long long access\_count;/\* 访问量 \*/ |

|  |  |
| --- | --- |
| 5 | }; |

我们可以用一个一维数组access\_record存储这张表，其中access\_record[index\_i]为编号为index\_i的国家的记录，也就是说，数据的存储位置由index\_i值唯一确定。例如，中国大陆的index\_i值为86，那么，access\_record[86].access\_count即为NowaMagic在中国大陆地区的访问量。

* 然而，我们知道，“China”比起数字86来，明显更接近自然语言，对于人脑的记忆来说更方便，所以，我们能不能想一个办法，做一个从国家/地区名到数字索引的映射呢？这就涉及到散列函数(Hash Function)了。

**哈希函数**

散列函数(Hash Function)又名哈希函数，是计算机科学中一个重要的课题。什么是散列函数呢？其实，这个概念并没有一个严格的定义。一般说来，散列函数满足以下的条件：

1. 对输入值运算，得到一个固定长度的摘要(Hash value)；
2. 不同的输入值可能对应同样的输出值；

以下的函数都可以认为是一个散列函数：

* f(x) = x mod 16;  (1)
* f(x) = (x2 + 10) \* x;  (2)
* f(x) = (x | 0×0000FFFF) XOR (x >> 16);  (3)

不过，仅仅满足上面这两条的函数，作为散列函数，还有不足的地方。我们还希望散列函数满足下面几点：

1. 散列函数的输出值尽量接近均匀分布；
2. x的微小变化可以使f(x)发生非常大的变化，即所谓“雪崩效应”([Avalanche effect](http://en.wikipedia.org/wiki/Avalanche_effect))；

上面两点用数学语言表示，就是：

1. 输出值y的分布函数F(y)=y/m, m为散列函数的最大值。或记为y~U[0, m]
2. |df(x)/dx| >> 1；

从上面两点，大家看看，前面举例的三个散列函数，哪个更好呢？对了，是第三个：f(x) = (x | 0×0000FFFF) XOR (x >> 16);

它很完美地满足“好的散列函数”的两个附加条件。

那么，为什么散列函数要带有这两个附加条件呢？原来，这是为了减少“哈希冲突”（Hashcollision），也就是两个不同输入产生了相同输出值的情况。根据抽屉原理，Hash算法不可能没有冲突(collision)，但是，由于冲突会造成一些问题，可能会影响到应用Hash函数的某些算法的效率，所以，我们需要尽量避免之。这样，对Hash算法的选择，就是很重要的了。密码学中的著名摘要算法的[MD5](http://en.wikipedia.org/wiki/MD5)和[SHA1](http://en.wikipedia.org/wiki/SHA_hash_functions)，以及另一种用于字符串摘要计算的[Jenkins Hash](http://en.wikipedia.org/wiki/Jenkins_hash_function)算法，都是很经典的Hash算法，有兴趣的同学可以阅读参考。

* 对计算机信息安全感兴趣的同学，一定听说过密码学家[王小云](http://zh.wikipedia.org/zh-cn/%E7%8E%8B%E5%B0%8F%E9%9B%B2)教授。王教授成名的贡献，就是发现了大大加速找出MD5和SHA1等Hash算法冲突的方法。譬如，根据“[生日攻击](http://songshuhui.net/archives/23737.html)”理论，对于Hash value为160bit的SHA1算法，找出一个Hash冲突需要280次运算，而王小云找出了一个269次运算就能找出冲突的算法，也就是提高了211=2048倍的效率！所以说，王教授的成果大大动摇了现代密码学的基础。

这时再来一个问题：Hash表中，数据存储的位置，是通过Hash函数计算得到的。那么，如果两条数据记录的Hash值发生冲突，应该怎么办呢？

**哈希冲突的处理**

在Hash表的建立时，会发生Hash值冲突的情况。实际上，如果记录Hash值的范围多于Hash表的条数，根据抽屉原理，一定会发生冲突。对于冲突的处理，我们一般有这几种方法：

1. 对Hash表中每个Hash值建立一个冲突表，即将冲突的几个记录以表的形式存储在其中；
2. 改变规则重新计算一次Hash值；
3. 建立一个公用的区域存放冲突的表项；

在工程上，考虑到实现算法的复杂度，方法1用得是最多的。对于方法1，又有两种不同的实现，一种方法是对每个Hash值，建立一个Hash桶(Bucket)，桶的容量是固定的，也就是只能处理固定次数的冲突，如1048576个Hash桶，每个桶中有4个表项(Entry)，总计4M个表项。另一种方法是，不限制Hash桶的容量，以链表形式将冲突的记录挂接在一个Hash桶中。

这两种实现各有什么利弊呢？首先，让我们看看第一种实现：

在这种情况下，由于Hash桶容量的限制，所以，有可能发生Hash表填不满的情况，也就是，虽然Hash表里面还有空位，但是新建的表项由于冲突过多，而不能装入Hash表中。不过，这样的实现也有其好处，就是查表的最大开销是可以确定的，因为最多处理的冲突数是确定的，所以算法的时间复杂度为O(1)+O(m)，其中m为Hash桶容量。

而另一种实现，由于Hash桶的容量是无限的，因此，只要没有超出Hash表的最大容量，就能够容纳新建的表项。但是，一旦发生了Hash冲突严重的情况，就会造成Hash桶的链表过长，大大降低查找效率。在最坏的情况下，时间复杂度退化为O(n)，其中n为Hash表的总容量。当然，这种情况的概率小之又小，几乎是可以忽略的。

Hash表的一个应用例子，是在网关(Gateway)中。以网络防火墙为例，它是根据源IP，目的IP，源端口，目的端口，协议号构成的五元组来标识一条网络数据流的，并且根据五元组来建立会话表项(session entry)。为了查找便捷，一般都使用Hash表来实现这个会话表，以提高转发的效率。事实上，对于大量表项的查找，逐项查找是不允许的，一般都使用Hash表来实现。不夸张的说，我们可以说，在你生活的每一天，都免不了同Hash表打交道，比如，查字典。

* 最后，我想说，数据结构的万紫千红中，我独爱Hash表这一种。