Atitit hash的实现原理以及性能更高的解决方案attilax总结

[1. Hash函数实现原理。实际中的Hash函数是指把一个大范围映射到一个小范围 2](#_Toc9363)

[1.1. Attilax总结的一个原理，就是想办法吧对象转化为一个数字，然后数字分区存储即可。。或者字串纯粹按照字符集也不错啊 2](#_Toc23787)

[1.2. 哈希法主要包括以下两方面的内容：  1）如何构造哈希函数 2）如何处理冲突。 2](#_Toc6285)

[1.3.  哈希函数的构造方法 原则是：①函数本身便于计算；②计算出来的地址分布均匀 2](#_Toc8469)

[1.4. 原理的比喻 2](#_Toc22758)

[1.5. Hash桶 buckets。。。就是分区单位 3](#_Toc29226)

[1.6. 哈希冲突冲突 3](#_Toc1385)

[1.7.   哈希函数的构造方法 数字分析法 平方取中法 分段叠加法 除留余数法 伪随机数法 3](#_Toc2806)

[1.8.](#_Toc26401) **[处理冲突的方法](#_Toc26401)** [。常用的解决冲突方法有以下四种：](#_Toc26401)**[开放定址法](#_Toc26401)**[这种方法也称](#_Toc26401)**[再散列法 再哈希法 链地址法 4、建立公共溢出区](#_Toc26401)** [4](#_Toc26401)

[2. 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； 4](#_Toc19490)

[2.2. 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 5](#_Toc12487)

[2.3. 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）。 5](#_Toc1868)

[2.4. Attilax的hashcode实现方法 字符ascii码连接法 5](#_Toc30736)

[3. 如何提升hash的性能 5](#_Toc957)

[4. 参考资料 5](#_Toc22007)

# Hash函数实现原理。实际中的Hash函数是指把一个大范围映射到一个小范围

## Attilax总结的一个原理，就是想办法吧对象转化为一个数字，然后数字分区存储即可。。或者字串纯粹按照字符集也不错啊

## 哈希法主要包括以下两方面的内容：  1）如何构造哈希函数 2）如何处理冲突。

## 哈希函数的构造方法 原则是：①函数本身便于计算；②计算出来的地址分布均匀

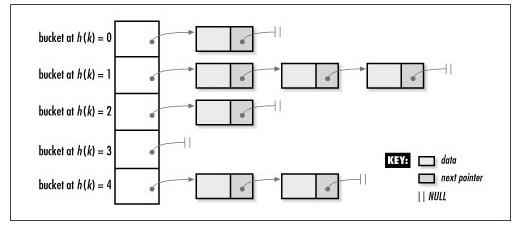
    构造哈希函数的原则是：①函数本身便于计算；②计算出来的地址分布均匀，即对任一关键字k，f(k) 对应不同地址的概率相等，目的是尽可能减少冲突。

## 原理的比喻

这个问题有点难度，不是很好说清楚。 我来做一个比喻吧。   
我们有很多的小猪，每个的体重都不一样，假设体重分布比较平均(我们考虑到公斤级别)，我们按照体重来分,划分成100个小猪圈。   
然后把每个小猪，按照体重赶进各自的猪圈里，记录档案。 好了，如果我们要找某个小猪怎么办呢？我们需要每个猪圈，每个小猪的比对吗？   
当然不需要了。 我们先看看要找的这个小猪的体重，然后就找到了对应的猪圈了。   
在这个猪圈里的小猪的数量就相对很少了。   
我们在这个猪圈里就可以相对快的找到我们要找到的那个小猪了。 对应于hash算法。   
就是按照hashcode分配不同的猪圈，将hashcode相同的猪放到一个猪圈里。   
查找的时候，先找到hashcode对应的猪圈，然后在逐个比较里面的小猪。 所以问题的关键就是建造多少个猪圈比较合适。 如果每个小猪的体重全部不同（考虑到毫克级别)，每个都建一个猪圈，那么我们可以最快速度的找到这头猪。缺点就是，建造那么多猪圈的费用有点太高了。 如果我们按照10公斤级别进行划分，那么建造的猪圈只有几个吧，那么每个圈里的小猪就很多了。我们虽然可以很快的找到猪圈，但从这个猪圈里逐个确定那头小猪也是很累的。 所以，好的hashcode，可以根据实际情况，根据具体的需求，在时间成本(更多的猪圈，更快的速度)和空间本(更少的猪圈，更低的空间需求)之间平衡。

## Hash桶 buckets。。。就是分区单位

一个由5个buckets组成的[哈希表](http://www.nowamagic.net/academy/tag/%E5%93%88%E5%B8%8C%E8%A1%A8" \t "http://www.nowamagic.net/academy/detail/_blank)，里面有7个元素：



## 哈希冲突冲突

这时又计算出1这个  
位置，那么数组1这个位置，就必须储存两个数了。这时，就叫  
哈希冲突。冲突之后就要按照顺序来存放了。  
如果数据的分布比较广泛，而且储存数据的数组长度比较大。  
那么哈希冲突就比较少。否则冲突是很高的。

## 哈希函数的构造方法 数字分析法 平方取中法 分段叠加法 除留余数法 伪随机数法

## **处理冲突的方法** 。常用的解决冲突方法有以下四种：**开放定址法**这种方法也称**再散列法 再哈希法 链地址法 4、建立公共溢出区**

# 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；

#### 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：

 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);  
 }

#### 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，

 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。

#### 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。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。

## 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

## 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。

## Attilax的hashcode实现方法 字符ascii码连接法

对于大部分字符串很好。。但是开头相同的比较麻烦。。发现冲突大于一个法治，则把list换成map，可以再内接传统的乘法子hash集合吧。。

# 如何提升hash的性能

使用更加简单的hash算法。

根据特定类型，使用特殊的hash法，可以大力提升性能。。

# 参考资料

几种经典的hash算法.html

Hash算法原理.xhtml

哈希表及处理冲突的方法\_你若盛开-清风自来\_新浪博客.html