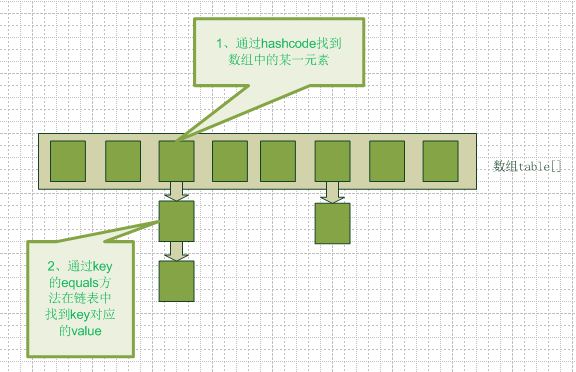
**[深入理解HashMap（及hash函数的真正巧妙之处）](http://pengranxiang.iteye.com/blog/543893)**

Hashmap是一种非常常用的、应用广泛的数据类型，最近研究到相关的内容，就正好复习一下。网上关于hashmap的文章很多，但到底是自己学习的总结，就发出来跟大家一起分享，一起讨论。

# hashmap的数据结构

要知道hashmap是什么，首先要搞清楚它的数据结构，在java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，hashmap也不例外。Hashmap实际上是一个数组和链表的结合体（在数据结构中，一般称之为“链表散列“），请看下图（横排表示数组，纵排表示数组元素【实际上是一个链表】）。



从图中我们可以看到一个hashmap就是一个数组结构，当新建一个hashmap的时候，就会初始化一个数组。我们来看看java代码：

**Java代码  [收藏代码](javascript:void())**

1. **/\*\***
2. **\* The table, resized as necessary. Length MUST Always be a power of two.**
3. **\*  FIXME 这里需要注意这句话，至于原因后面会讲到**
4. **\*/**
5. **transient Entry[] table;**

**Java代码  [收藏代码](javascript:void())**

1. **static class Entry<K,V> implements Map.Entry<K,V> {**
2. **final K key;**
3. **V value;**
4. **final int hash;**
5. **Entry<K,V> next;**
6. **..........**
7. **}**

上面的Entry就是数组中的元素，它持有一个指向下一个元素的引用，这就构成了链表。

     当我们往hashmap中put元素的时候，先根据key的hash值得到这个元素在数组中的位置（即下标），然后就可以把这个元素放到对应的位置中了。如果这个元素所在的位子上已经存放有其他元素了，那么在同一个位子上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。从hashmap中get元素时，首先计算key的hashcode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。从这里我们可以想象得到，如果每个位置上的链表只有一个元素，那么hashmap的get效率将是最高的，但是理想总是美好的，现实总是有困难需要我们去克服，哈哈~

# hash算法

我们可以看到在hashmap中要找到某个元素，需要根据key的hash值来求得对应数组中的位置。如何计算这个位置就是hash算法。前面说过hashmap的数据结构是数组和链表的结合，所以我们当然希望这个hashmap里面的元素位置尽量的分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用hash算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，而不用再去遍历链表。

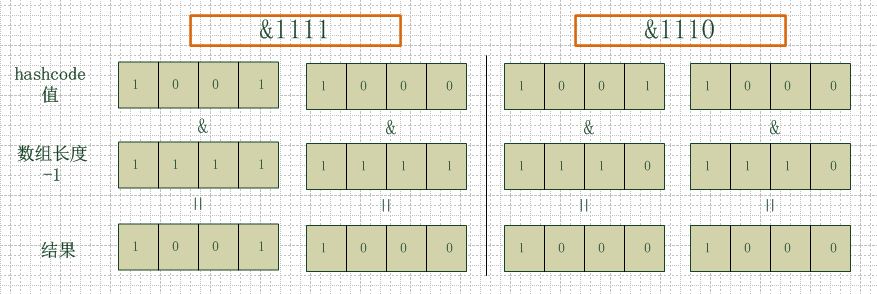
所以我们首先想到的就是把hashcode对数组长度取模运算，这样一来，元素的分布相对来说是比较均匀的。但是，“模”运算的消耗还是比较大的，能不能找一种更快速，消耗更小的方式那？java中时这样做的，

**Java代码  [收藏代码](javascript:void())**

1. **static** **int** indexFor(**int** h, **int** length) {
2. **return** h & (length-1);
3. }

首先算得key得hashcode值，然后跟数组的长度-1做一次“与”运算（&）。看上去很简单，其实比较有玄机。比如数组的长度是2的4次方，那么hashcode就会和2的4次方-1做“与”运算。很多人都有这个疑问，为什么hashmap的数组初始化大小都是2的次方大小时，hashmap的效率最高，我以2的4次方举例，来解释一下为什么数组大小为2的幂时hashmap访问的性能最高。

         看下图，左边两组是数组长度为16（2的4次方），右边两组是数组长度为15。两组的hashcode均为8和9，但是很明显，当它们和1110“与”的时候，产生了相同的结果，也就是说它们会定位到数组中的同一个位置上去，这就产生了碰撞，8和9会被放到同一个链表上，那么查询的时候就需要遍历这个链表，得到8或者9，这样就降低了查询的效率。同时，我们也可以发现，当数组长度为15的时候，hashcode的值会与14（1110）进行“与”，那么最后一位永远是0，而0001，0011，0101，1001，1011，0111，1101这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！



          所以说，当数组长度为2的n次幂的时候，不同的key算得得index相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。

          说到这里，我们再回头看一下hashmap中默认的数组大小是多少，查看源代码可以得知是16，为什么是16，而不是15，也不是20呢，看到上面annegu的解释之后我们就清楚了吧，显然是因为16是2的整数次幂的原因，在小数据量的情况下16比15和20更能减少key之间的碰撞，而加快查询的效率。

所以，在存储大容量数据的时候，最好预先指定hashmap的size为2的整数次幂次方。就算不指定的话，也会以大于且最接近指定值大小的2次幂来初始化的，代码如下(HashMap的构造方法中)：

**Java代码  [收藏代码](javascript:void())**

1. // Find a power of 2 >= initialCapacity
2. **int** capacity = 1;
3. **while** (capacity < initialCapacity)
4. capacity <<= 1;

总结：

        本文主要描述了HashMap的结构，和hashmap中hash函数的实现，以及该实现的特性，同时描述了hashmap中resize带来性能消耗的根本原因，以及将普通的域模型对象作为key的基本要求。尤其是hash函数的实现，可以说是整个HashMap的精髓所在，只有真正理解了这个hash函数，才可以说对HashMap有了一定的理解。

# hashmap的resize

       当hashmap中的元素越来越多的时候，碰撞的几率也就越来越高（因为数组的长度是固定的），所以为了提高查询的效率，就要对hashmap的数组进行扩容，数组扩容这个操作也会出现在ArrayList中，所以这是一个通用的操作，很多人对它的性能表示过怀疑，不过想想我们的“均摊”原理，就释然了，而在hashmap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。

         那么hashmap什么时候进行扩容呢？当hashmap中的元素个数超过数组大小\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，也就是说，默认情况下，数组大小为16，那么当hashmap中元素个数超过16\*0.75=12的时候，就把数组的大小扩展为2\*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知hashmap中元素的个数，那么预设元素的个数能够有效的提高hashmap的性能。比如说，我们有1000个元素new HashMap(1000), 但是理论上来讲new HashMap(1024)更合适，不过上面annegu已经说过，即使是1000，hashmap也自动会将其设置为1024。 但是new HashMap(1024)还不是更合适的，因为0.75\*1000 < 1000, 也就是说为了让0.75 \* size > 1000, 我们必须这样new HashMap(2048)才最合适，既考虑了&的问题，也避免了resize的问题。

# key的hashcode与equals方法改写

在第一部分hashmap的数据结构中，annegu就写了get方法的过程：首先计算key的hashcode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。所以，hashcode与equals方法对于找到对应元素是两个关键方法。

Hashmap的key可以是任何类型的对象，例如User这种对象，为了保证两个具有相同属性的user的hashcode相同，我们就需要改写hashcode方法，比方把hashcode值的计算与User对象的id关联起来，那么只要user对象拥有相同id，那么他们的hashcode也能保持一致了，这样就可以找到在hashmap数组中的位置了。如果这个位置上有多个元素，还需要用key的equals方法在对应位置的链表中找到需要的元素，所以只改写了hashcode方法是不够的，equals方法也是需要改写滴~当然啦，按正常思维逻辑，equals方法一般都会根据实际的业务内容来定义，例如根据user对象的id来判断两个user是否相等。

在改写equals方法的时候，需要满足以下三点：

(1) 自反性：就是说a.equals(a)必须为true。

(2) 对称性：就是说a.equals(b)=true的话，b.equals(a)也必须为true。

(3) 传递性：就是说a.equals(b)=true，并且b.equals(c)=true的话，a.equals(c)也必须为true。

通过改写key对象的equals和hashcode方法，我们可以将任意的业务对象作为map的key(前提是你确实有这样的需要)。

# 总结：

        本文主要描述了HashMap的结构，和hashmap中hash函数的实现，以及该实现的特性，同时描述了hashmap中resize带来性能消耗的根本原因，以及将普通的域模型对象作为key的基本要求。尤其是hash函数的实现，可以说是整个HashMap的精髓所在，只有真正理解了这个hash函数，才可以说对HashMap有了一定的理解。