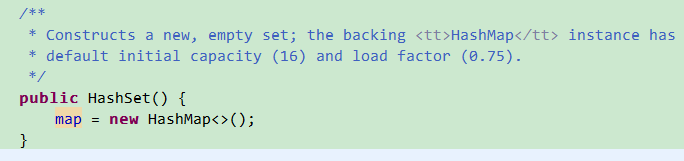
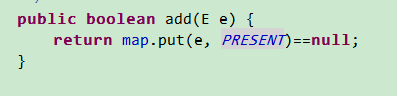
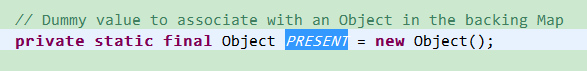
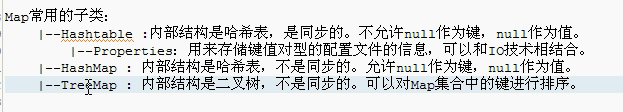
1. 实现Map接口的类用来存储键--值对。Set集合的底层代码就是采用Map来实现的

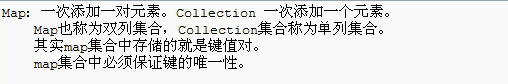


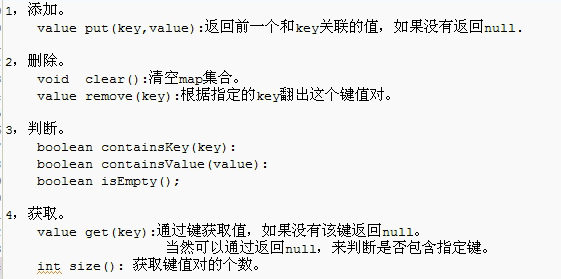
 

1. Map接口的实现类有HashMap、TreeMap(红黑树)、Hashtable、LinkedHashMap等。

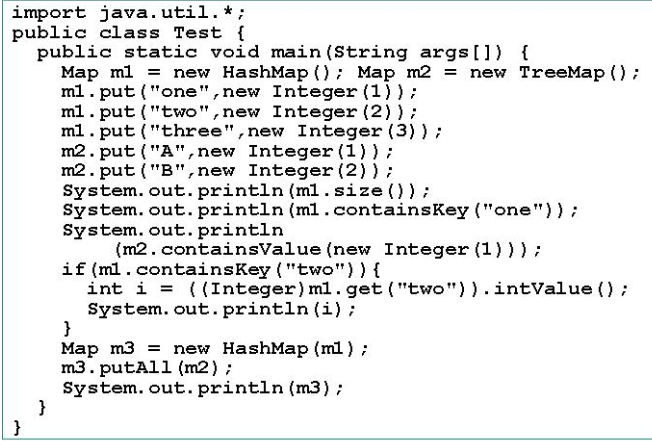


1. Map类中存储的键--值对通过键来标识，所以键值key不能重复(equals方法，对象作键的话要重写hashCode方法)。
2. 常用方法



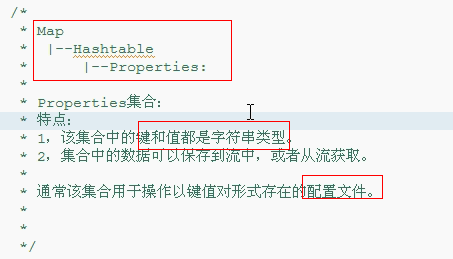


例子



输出结果：3;true;true;2;…

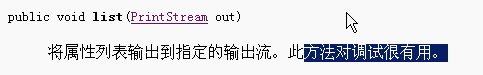
1. Properties集合



* 1. 存储、修改(覆盖)、取出元素



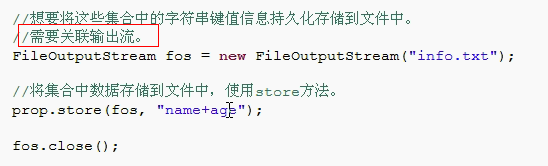
* 1. list方法(主要用于调试)



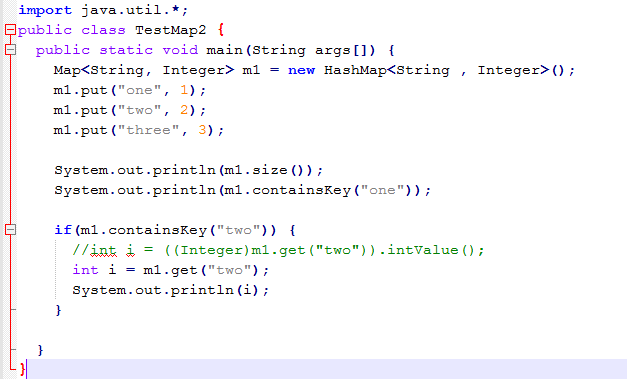




* 1. store方法(持久化)



1. Auto-boxing/unboxing：在合适的时机自动打包、解包(JDK1.5之后才有)
   1. 自动将基础类型转换为对象(基础类型包装类对象)
   2. 自动将对象转换为基础类型



1. 深入分析HashMap存储机制(http://zhaohe162.blog.163.com/blog/static/38216797201482333919581/)

**深入分析HashMap存储机制**

2014-09-23 15:39:19|  分类： [JDK](http://zhaohe162.blog.163.com/blog/#m=0&t=1&c=fks_084067087080085071084080084095087094083066082081095068) |  标签：[java](http://zhaohe162.blog.163.com/blog/" \l "m=0&t=3&c=java)  [jdk](http://zhaohe162.blog.163.com/blog/" \l "m=0&t=3&c=jdk)  [hashmap](http://zhaohe162.blog.163.com/blog/" \l "m=0&t=3&c=hashmap)  [hash](http://zhaohe162.blog.163.com/blog/#m=0&t=3&c=hash)  |举报|字号 订阅

 HashMap也是我们使用非常多的Collection，它是基于哈希表的 Map 接口的实现，以key-value的形式存在。在HashMap中，key-value总是会当做一个整体来处理，系统会根据hash算法来来计算key-value的存储位置，我们总是可以通过key快速地存、取value。下面就来分析HashMap的存取。

**一、定义**

      HashMap实现了Map接口，继承AbstractMap。其中Map接口定义了键映射到值的规则，而AbstractMap类提供 Map 接口的骨干实现，以最大限度地减少实现此接口所需的工作，其实AbstractMap类已经实现了Map，这里标注Map LZ觉得应该是更加清晰吧！

public class HashMap<K,V>

extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

**二、构造函数**

      HashMap提供了三个构造函数：

      HashMap()：构造一个具有默认初始容量 (16) 和默认加载因子 (0.75) 的空 HashMap。

      HashMap(int initialCapacity)：构造一个带指定初始容量和默认加载因子 (0.75) 的空 HashMap。

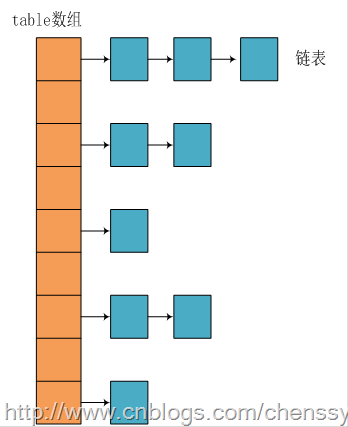
      HashMap(int initialCapacity, float loadFactor)：构造一个带指定初始容量和加载因子的空 HashMap。

      在这里提到了两个参数：初始容量，加载因子。这两个参数是影响HashMap性能的重要参数，其中容量表示哈希表中桶的数量，初始容量是创建哈希表时的容量，加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，它衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间是O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。系统默认负载因子为0.75，一般情况下我们是无需修改的。

      HashMap是一种支持快速存取的数据结构，要了解它的性能必须要了解它的数据结构。

**三、数据结构**

      我们知道在Java中最常用的两种结构是数组和模拟指针(引用)，几乎所有的数据结构都可以利用这两种来组合实现，HashMap也是如此。实际上HashMap是一个“链表散列”，如下是它数据结构：

[](http://images.cnitblog.com/blog/381060/201401/152128347367.png)

      从上图我们可以看出HashMap底层实现还是数组，只是数组的每一项都是一条链。其中参数initialCapacity就代表了该数组的长度。下面为HashMap构造函数的源码：

复制代码

public HashMap(int initialCapacity, float loadFactor) {

//初始容量不能<0

if (initialCapacity < 0)

throw new IllegalArgumentException("Illegal initial capacity: "

+ initialCapacity);

//初始容量不能 > 最大容量值，HashMap的最大容量值为2^30

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

//负载因子不能 < 0

if (loadFactor <= 0 || Float.isNaN(loadFactor))

throw new IllegalArgumentException("Illegal load factor: "

+ loadFactor);

// 计算出大于 initialCapacity 的最小的 2 的 n 次方值。

int capacity = 1;

while (capacity < initialCapacity)

capacity <<= 1;

this.loadFactor = loadFactor;

//设置HashMap的容量极限，当HashMap的容量达到该极限时就会进行扩容操作

threshold = (int) (capacity \* loadFactor);

//初始化table数组

table = new Entry[capacity];

init();

}

复制代码

      从源码中可以看出，每次新建一个HashMap时，都会初始化一个table数组。table数组的元素为Entry节点。

复制代码

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;

final int hash;

/\*\*

\* Creates new entry.

\*/

Entry(int h, K k, V v, Entry<K,V> n) {

value = v;

next = n;

key = k;

hash = h;

}

.......

}

复制代码

      其中Entry为HashMap的内部类，它包含了键key、值value、下一个节点next，以及hash值，这是非常重要的，正是由于Entry才构成了table数组的项为链表。

      上面简单分析了HashMap的数据结构，下面将探讨HashMap是如何实现快速存取的。

**四、存储实现：put(key,vlaue)**

      首先我们先看源码

复制代码

public V put(K key, V value) {

//当key为null，调用putForNullKey方法，保存null与table第一个位置中，这是HashMap允许为null的原因

if (key == null)

return putForNullKey(value);

//计算key的hash值

int hash = hash(key.hashCode()); ------(1)

//计算key hash 值在 table 数组中的位置

int i = indexFor(hash, table.length); ------(2)

//从i出开始迭代 e,找到 key 保存的位置

for (Entry<K, V> e = table[i]; e != null; e = e.next) {

Object k;

//判断该条链上是否有hash值相同的(key相同)

//若存在相同，则直接覆盖value，返回旧value

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value; //旧值 = 新值

e.value = value;

e.recordAccess(this);

return oldValue; //返回旧值

}

}

//修改次数增加1

modCount++;

//将key、value添加至i位置处

addEntry(hash, key, value, i);

return null;

}

复制代码

      通过源码我们可以清晰看到HashMap保存数据的过程为：首先判断key是否为null，若为null，则直接调用putForNullKey方法。若不为空则先计算key的hash值，然后根据hash值搜索在table数组中的索引位置，如果table数组在该位置处有元素，则通过比较是否存在相同的key，若存在则覆盖原来key的value，否则将该元素保存在链头（最先保存的元素放在链尾）。若table在该处没有元素，则直接保存。这个过程看似比较简单，其实深有内幕。有如下几点：

      1、 先看迭代处。此处迭代原因就是为了防止存在相同的key值，若发现两个hash值（key）相同时，HashMap的处理方式是用新value替换旧value，这里并没有处理key，这就解释了HashMap中没有两个相同的key。

      2、 在看（1）、（2）处。这里是HashMap的精华所在。首先是hash方法，该方法为一个纯粹的数学计算，就是计算h的hash值。

static int hash(int h) {

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

      我们知道对于HashMap的table而言，数据分布需要均匀（最好每项都只有一个元素，这样就可以直接找到），不能太紧也不能太松，太紧会导致查询速度慢，太松则浪费空间。计算hash值后，怎么才能保证table元素分布均与呢？我们会想到取模，但是由于取模的消耗较大，HashMap是这样处理的：调用indexFor方法。

static int indexFor(int h, int length) {

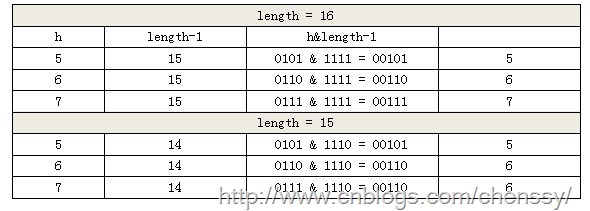
return h & (length-1);

}

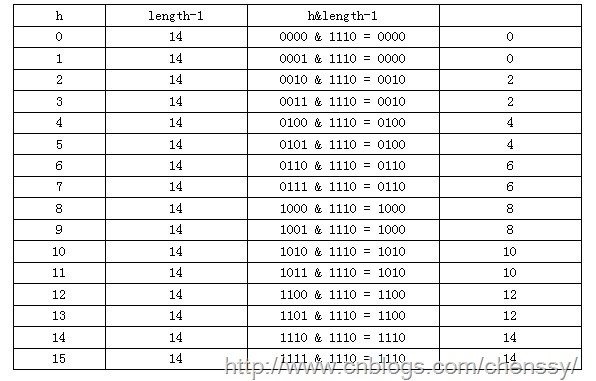
      HashMap的底层数组长度总是2的n次方，在构造函数中存在：capacity <<= 1;这样做总是能够保证HashMap的底层数组长度为2的n次方。当length为2的n次方时，h&(length - 1)就相当于对length取模，而且速度比直接取模快得多，这是HashMap在速度上的一个优化。至于为什么是2的n次方下面解释。

      我们回到indexFor方法，该方法仅有一条语句：h&(length - 1)，这句话除了上面的取模运算外还有一个非常重要的责任：均匀分布table数据和充分利用空间。

      这里我们假设length为16(2^n)和15，h为5、6、7。

[](http://images.cnitblog.com/blog/381060/201401/152128356118.jpg)

      当n=15时，6和7的结果一样，这样表示他们在table存储的位置是相同的，也就是产生了碰撞，6、7就会在一个位置形成链表，这样就会导致查询速度降低。诚然这里只分析三个数字不是很多，那么我们就看0-15。

[](http://images.cnitblog.com/blog/381060/201401/152128366276.jpg)

      从上面的图表中我们看到总共发生了8此碰撞，同时发现浪费的空间非常大，有1、3、5、7、9、11、13、15处没有记录，也就是没有存放数据。这是因为他们在与14进行&运算时，得到的结果最后一位永远都是0，即0001、0011、0101、0111、1001、1011、1101、1111位置处是不可能存储数据的，空间减少，进一步增加碰撞几率，这样就会导致查询速度慢。而当length = 16时，length – 1 = 15 即1111，那么进行低位&运算时，值总是与原来hash值相同，而进行高位运算时，其值等于其低位值。所以说当length = 2^n时，不同的hash值发生碰撞的概率比较小，这样就会使得数据在table数组中分布较均匀，查询速度也较快。

      这里我们再来复习put的流程：当我们想一个HashMap中添加一对key-value时，系统首先会计算key的hash值，然后根据hash值确认在table中存储的位置。若该位置没有元素，则直接插入。否则迭代该处元素链表并依此比较其key的hash值。如果两个hash值相等(e.hash == hash && ((k = e.key) == key || key.equals(k))),则用新的Entry的value覆盖原来节点的value。如果两个hash值不等 ，则将该节点插入该链表的链头。具体的实现过程见addEntry方法，如下：

复制代码

void addEntry(int hash, K key, V value, int bucketIndex) {

//获取bucketIndex处的Entry

Entry<K, V> e = table[bucketIndex];

//将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry

table[bucketIndex] = new Entry<K, V>(hash, key, value, e);

//若HashMap中元素的个数超过极限了，则容量扩大两倍

if (size++ >= threshold)

resize(2 \* table.length);

}

复制代码

      这个方法中有两点需要注意：

**一是链的产生。**这是一个非常优雅的设计。系统总是将新的Entry对象添加到bucketIndex处。如果bucketIndex处已经有了对象，那么新添加的Entry对象将指向原有的Entry对象，形成一条Entry链，但是若bucketIndex处没有Entry对象，也就是e==null,那么新添加的Entry对象指向null，也就不会产生Entry链了。

**二、扩容问题。**

      随着HashMap中元素的数量越来越多，发生碰撞的概率就越来越大，所产生的链表长度就会越来越长，这样势必会影响HashMap的速度，为了保证HashMap的效率，系统必须要在某个临界点进行扩容处理。该临界点在当HashMap中元素的数量等于table数组长度\*加载因子。但是扩容是一个非常耗时的过程，因为它需要重新计算这些数据在新table数组中的位置并进行复制处理。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。

**五、读取实现：get(key)**

      相对于HashMap的存而言，取就显得比较简单了。通过key的hash值找到在table数组中的索引处的Entry，然后返回该key对应的value即可。

复制代码

public V get(Object key) {

// 若为null，调用getForNullKey方法返回相对应的value

if (key == null)

return getForNullKey();

// 根据该 key 的 hashCode 值计算它的 hash 码

int hash = hash(key.hashCode());

// 取出 table 数组中指定索引处的值

for (Entry<K, V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {

Object k;

//若搜索的key与查找的key相同，则返回相对应的value

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

return null;

}

复制代码

      在这里能够根据key快速的取到value除了和HashMap的数据结构密不可分外，还和Entry有莫大的关系，在前面就提到过，HashMap在存储过程中并没有将key，value分开来存储，而是当做一个整体key-value来处理的，这个整体就是Entry对象。同时value也只相当于key的附属而已。在存储的过程中，系统根据key的hashcode来决定Entry在table数组中的存储位置，在取的过程中同样根据key的hashcode取出相对应的Entry对象。