# HashMap原理分析

## jdk1.7

HashMap是一个用于存储Key-Value键值对的集合，每一个键值对也叫做**Entry**。这些个键值对（Entry）分散存储在一个数组当中，这个数组就是HashMap的主干。

HashMap数组每一个元素的初始值都是Null。



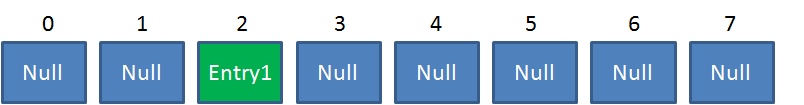
对于HashMap，我们最常使用的是两个方法：Get 和 Put。

## Put方法的原理

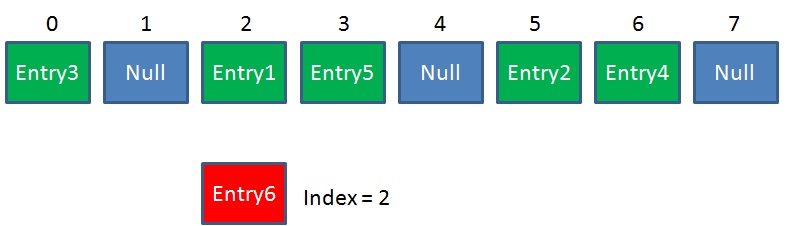
调用 hashMap.put("apple", 0) ，插入一个Key为“apple"的元素。这时候我们需要利用一个哈希函数来确定Entry的插入位置（index）：

index = Hash（“apple”）

假定最后计算出的index是2，那么结果如下：

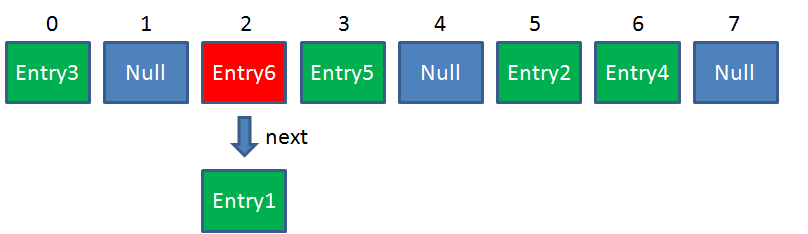


但是，因为HashMap的长度是有限的，当插入的Entry越来越多时，再完美的Hash函数也难免会出现index冲突的情况。比如下面这样：



这时候该怎么办呢？我们可以利用**链表**来解决。

HashMap数组的每一个元素不止是一个Entry对象，也是一个链表的头节点。每一个Entry对象通过Next指针指向它的下一个Entry节点。当新来的Entry映射到冲突的数组位置时，只需要插入到对应的链表即可：



需要注意的是，新来的Entry节点插入链表时，使用的是“头插法”。插入速度最快。

### 1.1.1、put深入

1. **put相同key，不同value时，返回原key值对应的value**

public static void main*(*String*[]* args*) {* HashMap*<*Object, Object*>* map = new HashMap*<>(*13*)*;  
 map.put*(*"1",2*)*;  
*}*



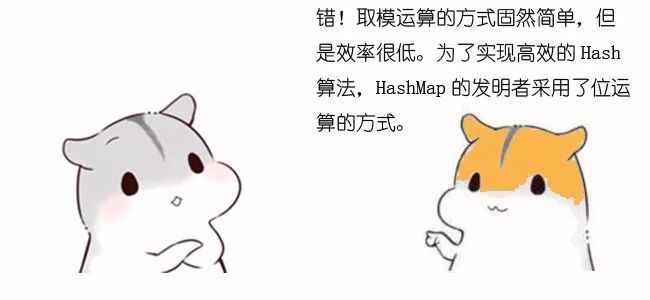
public HashMap*(*int initialCapacity*) {* this*(*initialCapacity, *DEFAULT\_LOAD\_FACTOR)*;  
*}*public HashMap*() {* this*(DEFAULT\_INITIAL\_CAPACITY*, *DEFAULT\_LOAD\_FACTOR)*;  
*}*

static final int *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4; // aka 16  
static final int *MAXIMUM\_CAPACITY* = 1 << 30;  
static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f;

Jdk1.7底层为数组+链表，在put时，要得到数组的下标



一个尽量均匀分布的Hash函数呢？我们通过利用Key的HashCode值来做某种运算。



//put的时候遍历链表

public V put*(*K key, V value*) {* if *(*table == *EMPTY\_TABLE) {* inflateTable*(*threshold*)*;  
 *}* if *(*key == null*)* return putForNullKey*(*value*)*;  
 int hash = hash*(*key*)*;  
 int i = *indexFor(*hash, table.length*)*;  
 for *(*Entry*<*K,V*>* e = table*[*i*]*; e != null; e = e.next*) {* Object k;  
 if *(*e.hash == hash && *((*k = e.key*)* == key || key.equals*(*k*))) {* V oldValue = e.value;  
 e.value = value;  
 e.recordAccess*(*this*)*;  
 return oldValue;  
 *}  
 }* modCount++;  
 addEntry*(*hash, key, value, i*)*;  
 return null;  
*}*

private void inflateTable*(*int toSize*) {* // Find a power of 2 >= toSize  
 int capacity = *roundUpToPowerOf2(*toSize*)*;  
  
 threshold = *(*int*)* Math.*min(*capacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1*)*;  
 table = new Entry*[*capacity*]*;  
 initHashSeedAsNeeded*(*capacity*)*;  
*}*

计算出2的指数次幂的容量值

private static int roundUpToPowerOf2*(*int number*) {* // assert number >= 0 : "number must be non-negative";  
 return number >= *MAXIMUM\_CAPACITY* ? *MAXIMUM\_CAPACITY* : *(*number > 1*)* ? Integer.*highestOneBit((*number - 1*)* << 1*)* : 1;  
*}*

*(*number - 1*)* << 1 数据扩大

public static int highestOneBit*(*int i*) {* // HD, Figure 3-1  
 i |= *(*i >> 1*)*;  
 i |= *(*i >> 2*)*;  
 i |= *(*i >> 4*)*;  
 i |= *(*i >> 8*)*;  
 i |= *(*i >> 16*)*;  
 return i - *(*i >>> 1*)*;

1.highestOneBit方法

传入一个int参数i，返回其二进制最高位1的权值，低位全部变为0，int 4字节，8位

若i为正整数，例如i=9，其二进制为1001，最高位1的权值为8，则返回值为8

若i为负整数，最高位1为符号位，返回值为-2147483648

若i为零，返回值为0

**或运算，有1则1，右移最边补0**

**Int 4个字节8共32位**

解析：

以129为例，二进制为10000001

i:

1 0 0 0 0 0 0 1

i >> 1:

0 1 0 0 0 0 0 0

二者位或：

**1 1 0 0 0 0 0 1**

同理，整个过程如下:

1 0 0 0 0 0 0 1

1 1 0 0 0 0 0 1

1 1 1 1 0 0 0 1

**1 1 1 1 1 1 1 1**

...

**i - (i >>> 1):**

1 1 1 1 1 1 1 1

- 0 1 1 1 1 1 1 1

**= 1 0 0 0 0 0 0 0**

返回值为128

整个过程可以理解为最高1位向右复制，然后将最高2位向右复制，再将最高4位向右复制……

这样最多经过五次迭代就能得到最终的结果。

roundUpToPowerOf2-->13--> Integer.*highestOneBit(11000)-->2^4=16*

//计算index

static int indexFor*(*int h, int length*) {* // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";  
 return h & *(*length-1*)*;  
*}*

index = HashCode（Key） & （Length - 1）

下面我们以值为“book”的Key来演示整个过程：

1.计算book的hashcode，结果为十进制的3029737，二进制的101110001110101110 1001。

2.假定HashMap长度是默认的16，计算Length-1的结果为十进制的15，二进制的1111。

3.把以上两个结果做与运算，101110001110101110 1001 & 1111 = 1001，十进制是9，所以 index=9。

**可以说，Hash算法最终得到的index结果，完全取决于Key的Hashcode值的最后几位。**

//检查是否扩容，**两个条件**，执行添加

void addEntry*(*int hash, K key, V value, int bucketIndex*) {* if *((*size >= threshold*)* && *(*null != table*[*bucketIndex*])) {* resize*(*2 \* table.length*)*;  
 hash = *(*null != key*)* ? hash*(*key*)* : 0;  
 bucketIndex = *indexFor(*hash, table.length*)*;  
 *}* createEntry*(*hash, key, value, bucketIndex*)*;  
*}*

//table即数组的

transient Entry*<*K,V*>[]* table = *(*Entry*<*K,V*>[]) EMPTY\_TABLE*;

//扩容

void resize*(*int newCapacity*) {* Entry*[]* oldTable = table;  
 int oldCapacity = oldTable.length;  
 if *(*oldCapacity == *MAXIMUM\_CAPACITY) {* threshold = Integer.*MAX\_VALUE*;  
 return;  
 *}* Entry*[]* newTable = new Entry*[*newCapacity*]*;  
 transfer*(*newTable, initHashSeedAsNeeded*(*newCapacity*))*;  
 table = newTable;  
 threshold = *(*int*)*Math.*min(*newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1*)*;  
*}*

//数据转移，不一定每次都rehash，遍历数组及其内部的链表

void transfer*(*Entry*[]* newTable, boolean rehash*) {* int newCapacity = newTable.length;  
 for *(*Entry*<*K,V*>* e : table*) {* while*(*null != e*) {* Entry*<*K,V*>* next = e.next;  
 if *(*rehash*) {* e.hash = null == e.key ? 0 : hash*(*e.key*)*;  
 *}* int i = *indexFor(*e.hash, newCapacity*)*;  
 e.next = newTable*[*i*]*;  
 newTable*[*i*]* = e;  
 e = next;  
 *}  
 }  
}*

//根据boolean rehash，判断是否计算新的hash,再依据新的容量newCapacity，计算出数组下标。

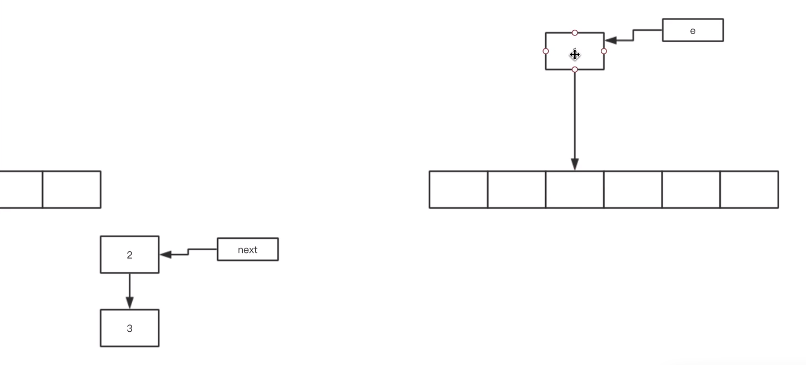


计算的扩容的数组的下标值=对应的扩容前数组下标值+原来数组的容量

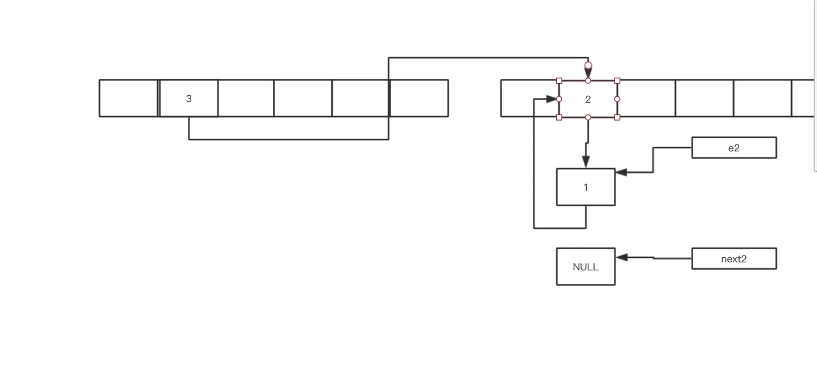
**元素的转移，造成链表数据逆序。**

newTable[i] = e;

e = next;



**多线程情况下扩容造成循环链表**



**减少扩容的方法：在明确知道要存储的元素的的大小时，可以设置集合的大小和加载因子，避免扩容。**



**Hashmap线程不安全，可以使用hashtable,hashtable的每个方法都有synchronized ，线程安全，效率低。**

**Hashtable---------------------------------------------------------------------------------------------------**

public synchronized V put*(*K key, V value*) {* // Make sure the value is not null  
 if *(*value == null*) {* throw new NullPointerException*()*;  
 *}* // Makes sure the key is not already in the hashtable.  
 Entry tab*[]* = table;  
 int hash = hash*(*key*)*;  
 int index = *(*hash & 0x7FFFFFFF*)* % tab.length;  
 for *(*Entry*<*K,V*>* e = tab*[*index*]* ; e != null ; e = e.next*) {* if *((*e.hash == hash*)* && e.key.equals*(*key*)) {* V old = e.value;  
 e.value = value;  
 return old;  
 *}  
 }* modCount++;  
 if *(*count >= threshold*) {* // Rehash the table if the threshold is exceeded  
 rehash*()*;  
  
 tab = table;  
 hash = hash*(*key*)*;  
 index = *(*hash & 0x7FFFFFFF*)* % tab.length;  
 *}* // Creates the new entry.  
 Entry*<*K,V*>* e = tab*[*index*]*;  
 tab*[*index*]* = new Entry*<>(*hash, key, value, e*)*;  
 count++;  
 return null;  
*}*

**Key为null的元素都存储在数组[0]的位置**

private V putForNullKey*(*V value*) {* for *(*Entry*<*K,V*>* e = table*[*0*]*; e != null; e = e.next*) {* if *(*e.key == null*) {* V oldValue = e.value;  
 e.value = value;  
 e.recordAccess*(*this*)*;  
 return oldValue;  
 *}  
 }* modCount++;  
 addEntry*(*0, null, value, 0*)*;  
 return null;  
*}*

//rehash Holder.*ALTERNATIVE\_HASHING\_THRESHOLD基本返回int最大值*

final boolean initHashSeedAsNeeded*(*int capacity*) {* boolean currentAltHashing = hashSeed != 0;  
 boolean useAltHashing = sun.misc.VM.*isBooted()* &&  
 *(*capacity >= Holder.*ALTERNATIVE\_HASHING\_THRESHOLD)*;  
 boolean switching = currentAltHashing ^ useAltHashing;  
 if *(*switching*) {* hashSeed = useAltHashing  
 ? sun.misc.Hashing.*randomHashSeed(*this*)* : 0;  
 *}* return switching;  
*}*

private static class Holder *{  
  
 /\*\*  
 \* Table capacity above which to switch to use alternative hashing.  
 \*/* static final int *ALTERNATIVE\_HASHING\_THRESHOLD*;  
  
 static *{* String altThreshold = java.security.AccessController.*doPrivileged(* new sun.security.action.GetPropertyAction*(* "jdk.map.althashing.threshold"*))*;  
  
 int threshold;  
 try *{* threshold = *(*null != altThreshold*)* ? Integer.*parseInt(*altThreshold*)* : *ALTERNATIVE\_HASHING\_THRESHOLD\_DEFAULT*;  
  
 // disable alternative hashing if -1  
 if *(*threshold == -1*) {* threshold = Integer.*MAX\_VALUE*;  
 *}* if *(*threshold < 0*) {* throw new IllegalArgumentException*(*"value must be positive integer."*)*;  
 *}  
 }* catch*(*IllegalArgumentException failed*) {* throw new Error*(*"Illegal value for 'jdk.map.althashing.threshold'", failed*)*;  
 *}  
  
 ALTERNATIVE\_HASHING\_THRESHOLD* = threshold;  
 *}  
}*

//头插法，插入到头部，size代表整个hashmap中所有元素的大小

void createEntry*(*int hash, K key, V value, int bucketIndex*) {* Entry*<*K,V*>* e = table*[*bucketIndex*]*;  
 table*[*bucketIndex*]* = new Entry*<>(*hash, key, value, e*)*;  
 size++;  
*}*

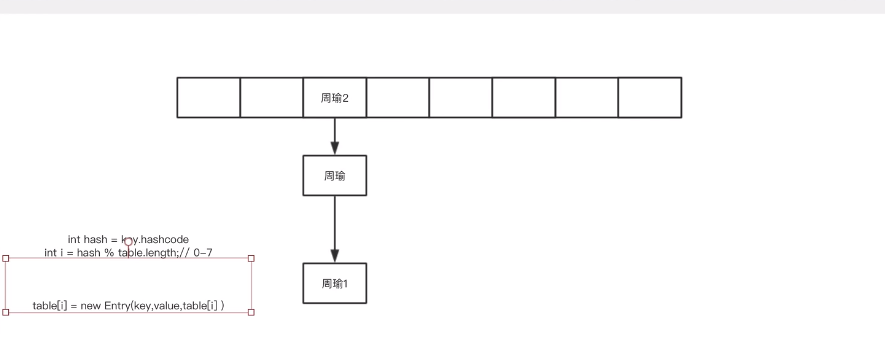
链表

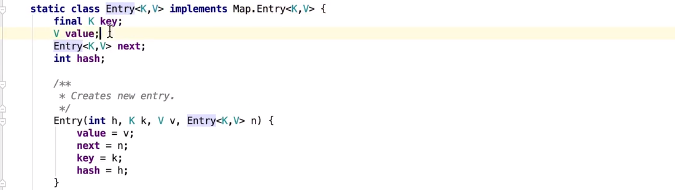
static class Entry*<*K,V*>* implements Map.Entry*<*K,V*> {* final K key;  
 V value;  
 Entry*<*K,V*>* next;  
 int hash;  
Entry*(*int h, K k, V v, Entry*<*K,V*>* n*) {* value = v;  
 next = n;  
 key = k;  
 hash = h;  
 *}* public final K getKey*() {* return key;  
 *}* public final V getValue*() {* return value;  
 *}* public final V setValue*(*V newValue*) {* V oldValue = value;  
 value = newValue;  
 return oldValue;  
 *}* public final boolean equals*(*Object o*) {* if *(*!*(*o instanceof Map.Entry*))* return false;  
 Map.Entry e = *(*Map.Entry*)*o;  
 Object k1 = getKey*()*;  
 Object k2 = e.getKey*()*;  
 if *(*k1 == k2 || *(*k1 != null && k1.equals*(*k2*))) {* Object v1 = getValue*()*;  
 Object v2 = e.getValue*()*;  
 if *(*v1 == v2 || *(*v1 != null && v1.equals*(*v2*)))* return true;  
 *}* return false;  
 *}* public final int hashCode*() {* return Objects.*hashCode(*getKey*())* ^ Objects.*hashCode(*getValue*())*;  
 *}* public final String toString*() {* return getKey*()* + "=" + getValue*()*;  
 *}* void recordAccess*(*HashMap*<*K,V*>* m*) {  
 }* void recordRemoval*(*HashMap*<*K,V*>* m*) {  
 }  
}*

final boolean initHashSeedAsNeeded*(*int capacity*) {* boolean currentAltHashing = hashSeed != 0;  
 boolean useAltHashing = sun.misc.VM.*isBooted()* &&  
 *(*capacity >= Holder.*ALTERNATIVE\_HASHING\_THRESHOLD)*;  
 boolean switching = currentAltHashing ^ useAltHashing;  
 if *(*switching*) {* hashSeed = useAltHashing  
 ? sun.misc.Hashing.*randomHashSeed(*this*)* : 0;  
 *}* return switching;  
*}*

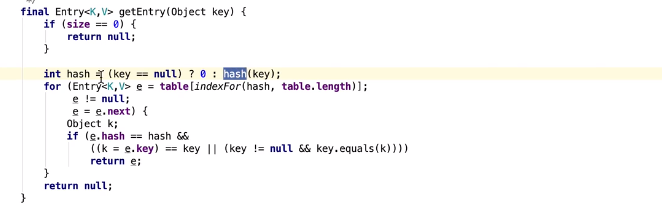
private V putForNullKey*(*V value*) {* for *(*Entry*<*K,V*>* e = table*[*0*]*; e != null; e = e.next*) {* if *(*e.key == null*) {* V oldValue = e.value;  
 e.value = value;  
 e.recordAccess*(*this*)*;  
 return oldValue;  
 *}  
 }* modCount++;  
 addEntry*(*0, null, value, 0*)*;  
 return null;  
*}*

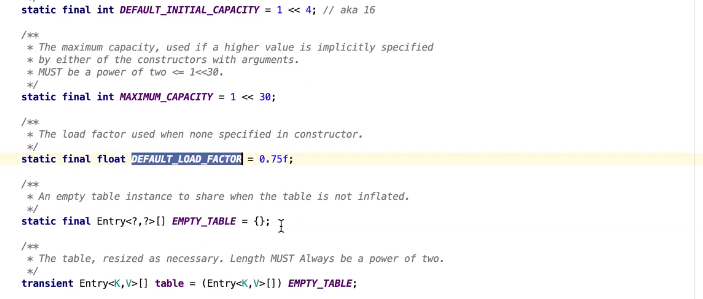
头插法，链表插入速度最快，向下移动链表，才能get到对应的值

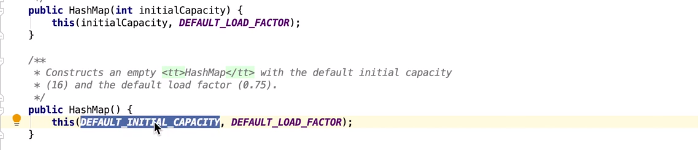




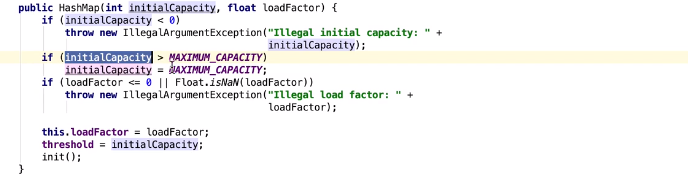
get方法

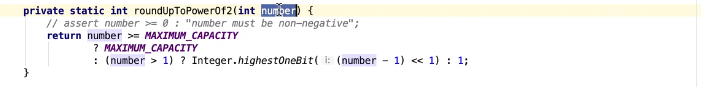
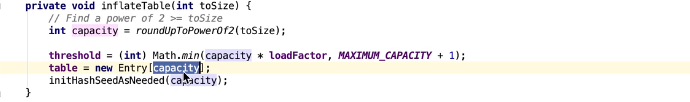


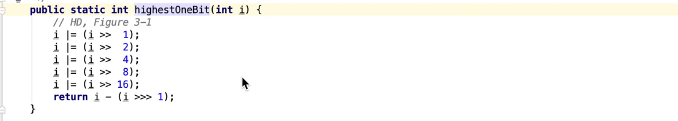




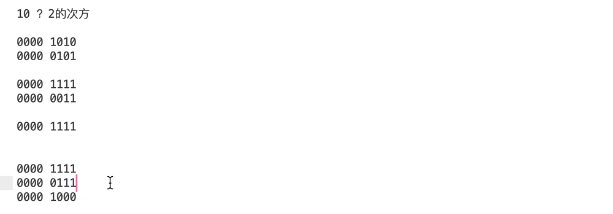
默认数组大小16，加载因子0.75







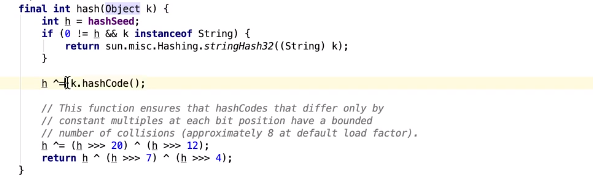
二进制数据，2的次方，只有一个byte位是2，其他都是不2



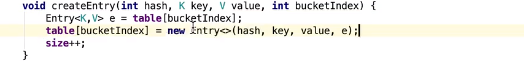
或运算，有1则1，右移最边补0

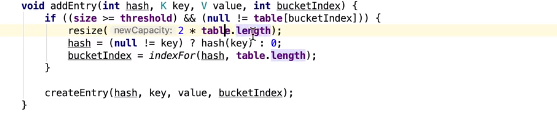
Int 4个字节8共32位

异或运算，相同为0，不同为1



与操作，都为1.则为1，其他为0





异或运算，不同为true,1.

final int hash*(*Object k*) {* int h = hashSeed;  
 if *(*0 != h && k instanceof String*) {* return sun.misc.Hashing.*stringHash32((*String*)* k*)*;  
 *}* h ^= k.hashCode*()*;  
 // This function ensures that hashCodes that differ only by  
 // constant multiples at each bit position have a bounded  
 // number of collisions (approximately 8 at default load factor).  
 h ^= *(*h >>> 20*)* ^ *(*h >>> 12*)*;  
 return h ^ *(*h >>> 7*)* ^ *(*h >>> 4*)*;  
*}*

Hash异或运算，提高key的散列程度，同时为复写检查容错。

static int indexFor*(*int h, int length*) {* // assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";  
 return h & *(*length-1*)*;  
*}*

**结果的取值范围是hash随机数，值低四位的取值范围 0000-1111**

**Lenth-1,保证数据有效，lenth必须为2的幂次方数才能保证结果在在hash取值的低四位。**

**Hash值高四位怎么变都不影响结果。**

Hash值为int数值，高位存在很多变化，但由于低位的限制，很多值的取值index会计算在一个数组脚标下，造成hash散列不均匀，一个链表很长。

//添加数据，扩容两个条件，扩容的目的是为了提高链表的get速度。

void addEntry*(*int hash, K key, V value, int bucketIndex*) {* if *((*size >= threshold*)* && *(*null != table*[*bucketIndex*])) {* resize*(*2 \* table.length*)*;  
 hash = *(*null != key*)* ? hash*(*key*)* : 0;  
 bucketIndex = *indexFor(*hash, table.length*)*;  
 *}* createEntry*(*hash, key, value, bucketIndex*)*;  
*}*

void createEntry*(*int hash, K key, V value, int bucketIndex*) {* Entry*<*K,V*>* e = table*[*bucketIndex*]*;  
 table*[*bucketIndex*]* = new Entry*<>(*hash, key, value, e*)*;  
 size++;  
*}*

Entry*(*int h, K k, V v, Entry*<*K,V*>* n*) {* value = v;  
 next = n;  
 key = k;  
 hash = h;  
*}*

扩容

void resize*(*int newCapacity*) {* Entry*[]* oldTable = table;  
 int oldCapacity = oldTable.length;  
 if *(*oldCapacity == *MAXIMUM\_CAPACITY) {* threshold = Integer.*MAX\_VALUE*;  
 return;  
 *}* Entry*[]* newTable = new Entry*[*newCapacity*]*;  
 transfer*(*newTable, initHashSeedAsNeeded*(*newCapacity*))*;  
 table = newTable;  
 threshold = *(*int*)*Math.*min(*newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1*)*;  
*}*

数据拷贝，可能会rehash

void transfer*(*Entry*[]* newTable, boolean rehash*) {* int newCapacity = newTable.length;  
 for *(*Entry*<*K,V*>* e : table*) {* while*(*null != e*) {* Entry*<*K,V*>* next = e.next;  
 if *(*rehash*) {* e.hash = null == e.key ? 0 : hash*(*e.key*)*;  
 *}* int i = *indexFor(*e.hash, newCapacity*)*;  
 e.next = newTable*[*i*]*;  
 newTable*[*i*]* = e;  
 e = next;  
 *}  
 }  
}*

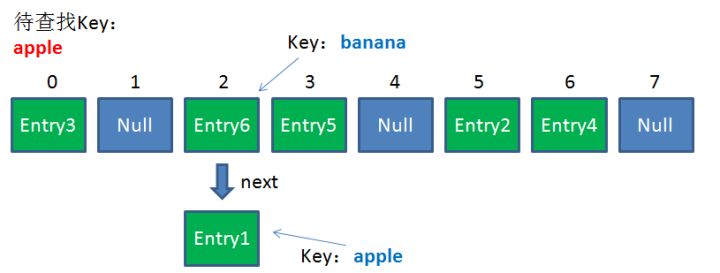
## 2.1、Get方法的原理

使用Get方法根据Key来查找Value的时候，发生了什么呢？

首先会把输入的Key做一次Hash映射，得到对应的index：

**index = Hash（“apple”）**

由于刚才所说的Hash冲突，同一个位置有可能匹配到多个Entry，这时候就需要顺着对应链表的头节点，一个一个向下来查找。假设我们要查找的Key是“apple”：

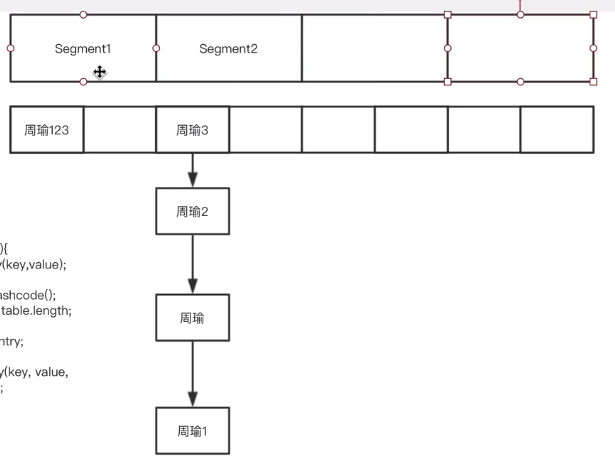


第一步，我们查看的是头节点Entry6，Entry6的Key是banana，显然不是我们要找的结果。

第二步，我们查看的是Next节点Entry1，Entry1的Key是apple，正是我们要找的结果。

之所以把Entry6放在头节点，是因为HashMap的发明者认为，**后插入的Entry被查找的可能性更大**。

## 2.1、ConcurrentMap，分段锁，线程安全



static final int *DEFAULT\_CONCURRENCY\_LEVEL* = 16;//并发级别，分段的数量

**//构造方法中多一个并发级别的默认参数**

public ConcurrentHashMap*() {* this*(DEFAULT\_INITIAL\_CAPACITY*, *DEFAULT\_LOAD\_FACTOR*, *DEFAULT\_CONCURRENCY\_LEVEL)*;  
*}*

public ConcurrentHashMap*(*int initialCapacity,  
 float loadFactor, int concurrencyLevel*) {* if *(*!*(*loadFactor > 0*)* || initialCapacity < 0 || concurrencyLevel <= 0*)* throw new IllegalArgumentException*()*;  
 if *(*concurrencyLevel > *MAX\_SEGMENTS)* concurrencyLevel = *MAX\_SEGMENTS*;  
 // Find power-of-two sizes best matching arguments  
 int sshift = 0;  
 int ssize = 1;  
 while *(*ssize < concurrencyLevel*) {* ++sshift;  
 ssize <<= 1;  
 *}* this.segmentShift = 32 - sshift;  
 this.segmentMask = ssize - 1;  
 if *(*initialCapacity > *MAXIMUM\_CAPACITY)* initialCapacity = *MAXIMUM\_CAPACITY*;  
 int c = initialCapacity / ssize;  
 if *(*c \* ssize < initialCapacity*)* ++c;  
 int cap = *MIN\_SEGMENT\_TABLE\_CAPACITY*;  
 while *(*cap < c*)* cap <<= 1;  
 // create segments and segments[0]  
 Segment*<*K,V*>* s0 =  
 new Segment*<*K,V*>(*loadFactor, *(*int*)(*cap \* loadFactor*)*,  
 *(*HashEntry*<*K,V*>[])*new HashEntry*[*cap*])*;  
 Segment*<*K,V*>[]* ss = *(*Segment*<*K,V*>[])*new Segment*[*ssize*]*;  
 *UNSAFE*.putOrderedObject*(*ss, *SBASE*, s0*)*; // ordered write of segments[0]  
 this.segments = ss;  
*}*

**//分段锁**

static final class Segment*<*K,V*>* extends ReentrantLock implements Serializable

**//转为接近size且大于size的2的次方数**

while *(*ssize < concurrencyLevel*) {* ++sshift;  
 ssize <<= 1;  
*}*

public V put*(*K key, V value*) {* Segment*<*K,V*>* s;  
 if *(*value == null*)* throw new NullPointerException*()*;  
 int hash = hash*(*key*)*;  
 int j = *(*hash >>> segmentShift*)* & segmentMask;  
 if *((*s = *(*Segment*<*K,V*>)UNSAFE*.getObject // nonvolatile; recheck  
 *(*segments, *(*j << *SSHIFT)* + *SBASE))* == null*)* // in ensureSegment  
 s = ensureSegment*(*j*)*;  
 return s.put*(*key, hash, value, false*)*;  
*}*