概述

文章的内容基于JDK1.7进行分析，之所以选用这个版本，是因为1.8的有些类做了改动，增加了阅读的难度，虽然是1.7，但是对于1.8做了重大改动的内容，文章也会进行说明。

HashMap基于Map接口实现，元素以键值对的方式存储，并且允许使用null 建和null　值，　因为key不允许重复，因此只能有一个键为null,另外HashMap不能保证放入元素的顺序，它是无序的，和放入的顺序并不能相同。HashMap是线程不安全的。

数据结构

继承关系

public class HashMap<K,V>extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

实现接口

Serializable, Cloneable, Map<K,V>

基本属性

static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; //默认初始化大小 16

static final float DEFAULT\_LOAD\_FACTOR = 0.75f; //负载因子0.75

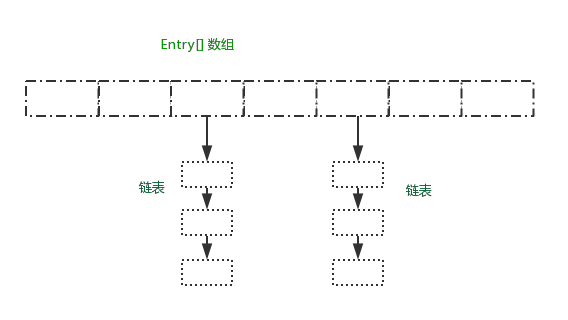
static final Entry<?,?>[] EMPTY\_TABLE = {}; //初始化的默认数组

transient int size; //HashMap中元素的数量

int threshold; //判断是否需要调整HashMap的容量

代码有移位运算，不理解请点击 [Java二进制位运算、移位运算>>、<<、>>>](https://my.oschina.net/90888/blog/edit/1627913)了解。

源码解析

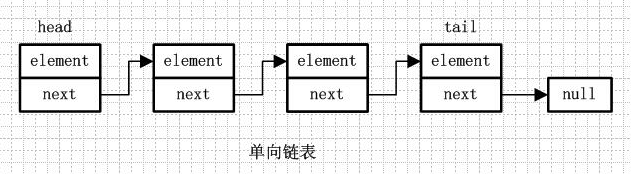


什么是链表

　　链表是由一系列非连续的节点组成的存储结构，简单分下类的话，链表又分为单向链表和双向链表，而单向/双向链表又可以分为循环链表和非循环链表，下面简单就这四种链表进行图解说明。

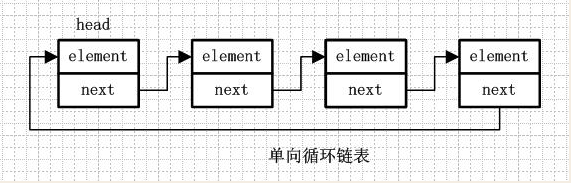
1.单向链表

　　　　单向链表就是通过每个结点的指针指向下一个结点从而链接起来的结构，最后一个节点的next指向null。



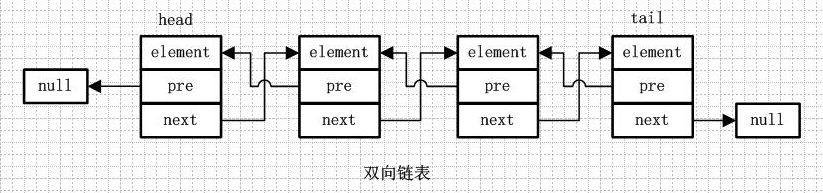
2.单向循环链表

　　　　单向循环链表和单向列表的不同是，最后一个节点的next不是指向null，而是指向head节点，形成一个“环”。



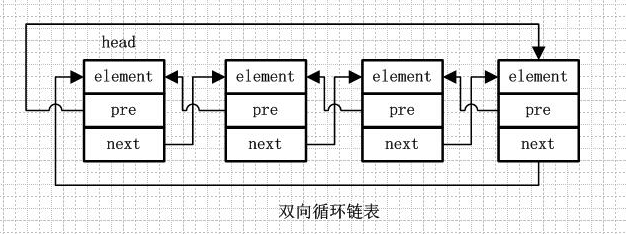
3.双向链表

　　　　从名字就可以看出，双向链表是包含两个指针的，pre指向前一个节点，next指向后一个节点，但是第一个节点head的pre指向null，最后一个节点的tail指向null。



4.双向循环链表

　　　　双向循环链表和双向链表的不同在于，第一个节点的pre指向最后一个节点，最后一个节点的next指向第一个节点，也形成一个“环”。**而LinkedList就是基于双向循环链表设计的。**



在进行源码解析之前，先从总体上对HashMap的数据存储结构进行一个大体上的说明。

存储结构如上图所示。

HashMap采用Entry数组来存储key-value对，每一个键值对组成了一个Entry实体，Entry类实际上是一个单向的链表结构，它具有Next指针，可以连接下一个Entry实体，依次来解决Hash冲突的问题，因为HashMap是按照Key的hash值来计算Entry在HashMap中存储的位置的，如果hash值相同，而key内容不相等，那么就用链表来解决这种hash冲突。

public class HashMap<K,V> extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable {

//默认初始化的容量

static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; // aka 16

//最大的容量

static final int MAXIMUM\_CAPACITY = 1 << 30;

//负载因子，当容量达到75%时就进行扩容操作

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

//当数组还没有进行扩容操作的时候，共享的一个空表对象

static final Entry<?,?>[] EMPTY\_TABLE = {};

//table,进行扩容操作，长度必须2的n次方

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

//Map中包含的元素数量

transient int size;

//阈值，用于判断是否需要扩容（threshold = 容量\*负载因子）

int threshold;

//加载因子实际的大小

final float loadFactor;

//HashMap改变的次数

transient int modCount;

static final int ALTERNATIVE\_HASHING\_THRESHOLD\_DEFAULT = Integer.MAX\_VALUE;

//内部类，通过vm来修改threshold的值

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")); //读取值

int threshold;

try {

threshold = (null != altThreshold) //修改值

? Integer.parseInt(altThreshold)

: ALTERNATIVE\_HASHING\_THRESHOLD\_DEFAULT;

// disable alternative hashing if -1

if (threshold == -1) {

threshold = Integer.MAX\_VALUE; //设置为Integer能表示的最大值

}

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; //返回

}

}

//HashCode的初始值为 0

transient int hashSeed = 0;

//构造方法，指定初始容量和负载因子

public HashMap(int initialCapacity, float loadFactor) {

if (initialCapacity < 0)

throw new IllegalArgumentException("Illegal initial capacity: " +

initialCapacity);

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

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

throw new IllegalArgumentException("Illegal load factor: " +

loadFactor);

this.loadFactor = loadFactor; //设置负载因子

threshold = initialCapacity; //初始容量

init(); //不做任何操作

}

//构造方法，指定了初始容量

public HashMap(int initialCapacity) {

this(initialCapacity, DEFAULT\_LOAD\_FACTOR);

}

//无参构造方法，使用默认的容量大小和负载因子，并调用其他的构造方法

public HashMap() {

this(DEFAULT\_INITIAL\_CAPACITY, DEFAULT\_LOAD\_FACTOR);

}

//构造函数，参数为指定的Map集合

public HashMap(Map<? extends K, ? extends V> m) {

this(Math.max((int) (m.size() / DEFAULT\_LOAD\_FACTOR) + 1,

DEFAULT\_INITIAL\_CAPACITY), DEFAULT\_LOAD\_FACTOR);

inflateTable(threshold);

putAllForCreate(m);

}

//选择合适的容量值，最好是number的2的幂数

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;

}

//扩充表，HashMap初始化时是一个空数组，此方法执行重新复制操作，创建一个新的Entry[]

private void inflateTable(int toSize) {

// Find a power of 2 >= toSize

int capacity = roundUpToPowerOf2(toSize); //capacity为2的幂数，大于等于toSize

threshold = (int) Math.min(capacity \* loadFactor, MAXIMUM\_CAPACITY + 1);

table = new Entry[capacity]; //新建数组，并重新赋值

initHashSeedAsNeeded(capacity); //修改hashSeed

}

// internal utilities

//初始化

void init() {

}

//与虚拟机设置有关，改变hashSeed的值

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;

}

//计算k 的 hash值

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);

}

//根据hashcode,和表的长度，返回存放的索引

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);

}

//返回Map中键值对的数量

public int size() {

return size;

}

//判断集合是否为空

public boolean isEmpty() {

return size == 0;

}

//返回key ，对应的值

public V get(Object key) {

if (key == null)

return getForNullKey();

Entry<K,V> entry = getEntry(key);

return null == entry ? null : entry.getValue();

}

//返回null键的值

private V getForNullKey() {

if (size == 0) {

return null;

}

for (Entry<K,V> e = table[0]; e != null; e = e.next) {

if (e.key == null)

return e.value;

}

return null;

}

//是否包含键为key的元素

public boolean containsKey(Object key) {

return getEntry(key) != null;

}

//返回键为key 的entry实体，不存在返回null

final Entry<K,V> getEntry(Object key) {

if (size == 0) {

return null;

}

int hash = (key == null) ? 0 : hash(key); //计算key的 hash值

//定位到Entry[] 数组中的存储位置，开始遍历该位置是否有链表存在

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

e != null;

e = e.next) {

Object k;

//判断是否有键位key 的entry实体。有就返回。

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

return e;

}

return null;

}

//向map中添加key-value 键值对，如果可以包含了key的映射，则旧的value将被替换

public V put(K key, V value) {

if (table == EMPTY\_TABLE) { //table如果为空，进行初始化操作

inflateTable(threshold);

}

if (key == null) //key 为null ,放入数组的0号索引位置

return putForNullKey(value);

int hash = hash(key); //计算key的hash值

int i = indexFor(hash, table.length); //计算key在entry数组中存储的位置

//判断该位置是否已经有元素存在

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

Object k;

//判断key是否已经在map中存在，若存在用新的value替换掉旧的value,并返回旧的value

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

V oldValue = e.value;

e.value = value;

e.recordAccess(this); //空方法

return oldValue;

}

}

modCount++; //修改次数加1

addEntry(hash, key, value, i); //将key-value转化为Entry实体，添加到Map中

return null;

}

//key = null, 对应的操作，keyweinull ,存放在entry[]中的0号位置。并用新值替换旧值

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;

}

//私有方法，添加元素

private void putForCreate(K key, V value) {

int hash = null == key ? 0 : hash(key); //计算hash值

int i = indexFor(hash, table.length); //计算在HashMap中的存储位置

//遍历i号存储位置的链表

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

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k)))) {

e.value = value;

return;

}

}

//创建Entry实体，存放到i号位置中

createEntry(hash, key, value, i);

}

//将m中的元素添加到HashMap中

private void putAllForCreate(Map<? extends K, ? extends V> m) {

for (Map.Entry<? extends K, ? extends V> e : m.entrySet())

putForCreate(e.getKey(), e.getValue());

}

//扩容操作

void resize(int newCapacity) {

Entry[] oldTable = table; //将table赋值给新的引用

int oldCapacity = oldTable.length;

if (oldCapacity == MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return;

}

//创建一个长度为newCapacity的数组

Entry[] newTable = new Entry[newCapacity];

//将table中的元素复制到newTable中

transfer(newTable, initHashSeedAsNeeded(newCapacity));

table = newTable;

//更改阈值

threshold = (int)Math.min(newCapacity \* loadFactor, MAXIMUM\_CAPACITY + 1);

}

//将table中的数据复制到newTable中

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) { //是否需要重新计算Hash值

e.hash = null == e.key ? 0 : hash(e.key);

}

int i = indexFor(e.hash, newCapacity); //计算存储的位置

e.next = newTable[i];

newTable[i] = e;

e = next;

}

}

}

//将m中的元素全部添加到HashMap中

public void putAll(Map<? extends K, ? extends V> m) {

int numKeysToBeAdded = m.size();

if (numKeysToBeAdded == 0) //为空返回

return;

if (table == EMPTY\_TABLE) { //是否需要执行初始化操作

inflateTable((int) Math.max(numKeysToBeAdded \* loadFactor, threshold));

}

//判断是否需要扩容

if (numKeysToBeAdded > threshold) {

int targetCapacity = (int)(numKeysToBeAdded / loadFactor + 1);

if (targetCapacity > MAXIMUM\_CAPACITY)

targetCapacity = MAXIMUM\_CAPACITY;

int newCapacity = table.length;

while (newCapacity < targetCapacity)

newCapacity <<= 1;

if (newCapacity > table.length)

resize(newCapacity);

}

//执行添加操作

for (Map.Entry<? extends K, ? extends V> e : m.entrySet())

put(e.getKey(), e.getValue());

}

//删除key ,并返回key对应的value值

public V remove(Object key) {

Entry<K,V> e = removeEntryForKey(key);

return (e == null ? null : e.value);

}

//返回key对应的实体

final Entry<K,V> removeEntryForKey(Object key) {

if (size == 0) {

return null;

}

int hash = (key == null) ? 0 : hash(key); //计算key的hash值

int i = indexFor(hash, table.length); //计算存储位置

Entry<K,V> prev = table[i];

Entry<K,V> e = prev;

while (e != null) {

Entry<K,V> next = e.next;

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k)))) {

modCount++;

size--;

if (prev == e)

table[i] = next;

else

prev.next = next; //链表删除

e.recordRemoval(this);

return e;

}

prev = e;

e = next;

}

return e;

}

//删除一个指定的实体

final Entry<K,V> removeMapping(Object o) {

if (size == 0 || !(o instanceof Map.Entry))

return null;

Map.Entry<K,V> entry = (Map.Entry<K,V>) o;

Object key = entry.getKey();

int hash = (key == null) ? 0 : hash(key);

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

Entry<K,V> prev = table[i];

Entry<K,V> e = prev;

while (e != null) {

Entry<K,V> next = e.next;

if (e.hash == hash && e.equals(entry)) {

modCount++;

size--;

if (prev == e)

table[i] = next;

else

prev.next = next;

e.recordRemoval(this);

return e;

}

prev = e;

e = next;

}

return e;

}

//删除map

public void clear() {

modCount++;

Arrays.fill(table, null);

size = 0;

}

//判断是否包含指定value的实体

public boolean containsValue(Object value) {

if (value == null)

return containsNullValue();

Entry[] tab = table;

for (int i = 0; i < tab.length ; i++)

for (Entry e = tab[i] ; e != null ; e = e.next)

if (value.equals(e.value))

return true;

return false;

}

//是否包含value== null

private boolean containsNullValue() {

Entry[] tab = table;

for (int i = 0; i < tab.length ; i++)

for (Entry e = tab[i] ; e != null ; e = e.next)

if (e.value == null)

return true;

return false;

}

//重写克隆方法

public Object clone() {

HashMap<K,V> result = null;

try {

result = (HashMap<K,V>)super.clone();

} catch (CloneNotSupportedException e) {

// assert false;

}

if (result.table != EMPTY\_TABLE) {

result.inflateTable(Math.min(

(int) Math.min(

size \* Math.min(1 / loadFactor, 4.0f),

// we have limits...

HashMap.MAXIMUM\_CAPACITY),

table.length));

}

result.entrySet = null;

result.modCount = 0;

result.size = 0;

result.init();

result.putAllForCreate(this);

return result;

}

//静态内部类 ，Entry用来存储键值对，HashMap中的Entry[]用来存储entry

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

final K key; //键

V value; //值

Entry<K,V> next; //采用链表存储HashCode相同的键值对，next指向下一个entry

int hash; //entry的hash值

//构造方法， 负责初始化entry

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();

}

//当使用相同的key的value被覆盖时调用

void recordAccess(HashMap<K,V> m) {

}

//每移除一个entry就被调用一次

void recordRemoval(HashMap<K,V> m) {

}

}

//添加实体

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);

}

//创建实体

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++;

}

//内部类实现Iterator接口，进行遍历操作

private abstract class HashIterator<E> implements Iterator<E> {

Entry<K,V> next; // next entry to return

int expectedModCount; // For fast-fail

int index; // current slot

Entry<K,V> current; // current entry

HashIterator() {

expectedModCount = modCount;

if (size > 0) { // advance to first entry

Entry[] t = table;

while (index < t.length && (next = t[index++]) == null)

;

}

}

//是否有下一个元素

public final boolean hasNext() {

return next != null;

}

//返回下一个元素

final Entry<K,V> nextEntry() {

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

Entry<K,V> e = next;

if (e == null)

throw new NoSuchElementException();

if ((next = e.next) == null) {

Entry[] t = table;

while (index < t.length && (next = t[index++]) == null)

;

}

current = e;

return e;

}

//删除

public void remove() {

if (current == null)

throw new IllegalStateException();

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

Object k = current.key;

current = null;

HashMap.this.removeEntryForKey(k);

expectedModCount = modCount;

}

}

private final class ValueIterator extends HashIterator<V> {

public V next() {

return nextEntry().value;

}

}

private final class KeyIterator extends HashIterator<K> {

public K next() {

return nextEntry().getKey();

}

}

private final class EntryIterator extends HashIterator<Map.Entry<K,V>> {

public Map.Entry<K,V> next() {

return nextEntry();

}

}

// Subclass overrides these to alter behavior of views' iterator() method

Iterator<K> newKeyIterator() {

return new KeyIterator();

}

Iterator<V> newValueIterator() {

return new ValueIterator();

}

Iterator<Map.Entry<K,V>> newEntryIterator() {

return new EntryIterator();

}

// Views

private transient Set<Map.Entry<K,V>> entrySet = null;

//返回key组成的Set集合

public Set<K> keySet() {

Set<K> ks = keySet;

return (ks != null ? ks : (keySet = new KeySet()));

}

private final class KeySet extends AbstractSet<K> {

public Iterator<K> iterator() {

return newKeyIterator();

}

public int size() {

return size;

}

public boolean contains(Object o) {

return containsKey(o);

}

public boolean remove(Object o) {

return HashMap.this.removeEntryForKey(o) != null;

}

public void clear() {

HashMap.this.clear();

}

}

//返回Value组成的集合

public Collection<V> values() {

Collection<V> vs = values;

return (vs != null ? vs : (values = new Values()));

}

private final class Values extends AbstractCollection<V> {

public Iterator<V> iterator() {

return newValueIterator();

}

public int size() {

return size;

}

public boolean contains(Object o) {

return containsValue(o);

}

public void clear() {

HashMap.this.clear();

}

}

public Set<Map.Entry<K,V>> entrySet() {

return entrySet0();

}

private Set<Map.Entry<K,V>> entrySet0() {

Set<Map.Entry<K,V>> es = entrySet;

return es != null ? es : (entrySet = new EntrySet());

}

private final class EntrySet extends AbstractSet<Map.Entry<K,V>> {

public Iterator<Map.Entry<K,V>> iterator() {

return newEntryIterator();

}

public boolean contains(Object o) {

if (!(o instanceof Map.Entry))

return false;

Map.Entry<K,V> e = (Map.Entry<K,V>) o;

Entry<K,V> candidate = getEntry(e.getKey());

return candidate != null && candidate.equals(e);

}

public boolean remove(Object o) {

return removeMapping(o) != null;

}

public int size() {

return size;

}

public void clear() {

HashMap.this.clear();

}

}

//将对象写入到输出流中

private void writeObject(java.io.ObjectOutputStream s)

throws IOException

{

// Write out the threshold, loadfactor, and any hidden stuff

s.defaultWriteObject();

// Write out number of buckets

if (table==EMPTY\_TABLE) {

s.writeInt(roundUpToPowerOf2(threshold));

} else {

s.writeInt(table.length);

}

// Write out size (number of Mappings)

s.writeInt(size);

// Write out keys and values (alternating)

if (size > 0) {

for(Map.Entry<K,V> e : entrySet0()) {

s.writeObject(e.getKey());

s.writeObject(e.getValue());

}

}

}

private static final long serialVersionUID = 362498820763181265L;

//从输入流中读取对象

private void readObject(java.io.ObjectInputStream s)

throws IOException, ClassNotFoundException

{

// Read in the threshold (ignored), loadfactor, and any hidden stuff

s.defaultReadObject();

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

throw new InvalidObjectException("Illegal load factor: " +

loadFactor);

}

// set other fields that need values

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

// Read in number of buckets

s.readInt(); // ignored.

// Read number of mappings

int mappings = s.readInt();

if (mappings < 0)

throw new InvalidObjectException("Illegal mappings count: " +

mappings);

// capacity chosen by number of mappings and desired load (if >= 0.25)

int capacity = (int) Math.min(

mappings \* Math.min(1 / loadFactor, 4.0f),

// we have limits...

HashMap.MAXIMUM\_CAPACITY);

// allocate the bucket array;

if (mappings > 0) {

inflateTable(capacity);

} else {

threshold = capacity;

}

init(); // Give subclass a chance to do its thing.

// Read the keys and values, and put the mappings in the HashMap

for (int i = 0; i < mappings; i++) {

K key = (K) s.readObject();

V value = (V) s.readObject();

putForCreate(key, value);

}

}

// These methods are used when serializing HashSets

int capacity() { return table.length; }

float loadFactor() { return loadFactor; }

}

重要方法深度解析

构造方法

HashMap() //无参构造方法

HashMap(int initialCapacity) //指定初始容量的构造方法

HashMap(int initialCapacity, float loadFactor) //指定初始容量和负载因子

HashMap(Map<? extends K,? extends V> m) //指定集合，转化为HashMap

HashMap提供了四个构造方法，构造方法中 ，依靠第三个方法来执行的，但是前三个方法都没有进行数组的初始化操作，即使调用了构造方法，此时存放HaspMap中数组元素的table表长度依旧为0 。在第四个构造方法中调用了inflateTable()方法完成了table的初始化操作，并将m中的元素添加到HashMap中。

添加方法

public V put(K key, V value) {

if (table == EMPTY\_TABLE) { //是否初始化

inflateTable(threshold);

}

if (key == null) //放置在0号位置

return putForNullKey(value);

int hash = hash(key); //计算hash值

int i = indexFor(hash, table.length); //计算在Entry[]中的存储位置

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); //添加到Map中

return null;

}

在该方法中，添加键值对时，首先进行table是否初始化的判断，如果没有进行初始化（分配空间，Entry[]数组的长度）。然后进行key是否为null的判断，如果key==null ,放置在Entry[]的0号位置。计算在Entry[]数组的存储位置，判断该位置上是否已有元素，如果已经有元素存在，则遍历该Entry[]数组位置上的单链表。判断key是否存在，如果key已经存在，则用新的value值，替换点旧的value值，并将旧的value值返回。如果key不存在于HashMap中，程序继续向下执行。将key-vlaue, 生成Entry实体，添加到HashMap中的Entry[]数组中。

addEntry()

/\*

\* hash hash值

\* key 键值

\* value value值

\* bucketIndex Entry[]数组中的存储索引

\* /

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

if ((size >= threshold) && (null != table[bucketIndex])) {

resize(2 \* table.length); //扩容操作，将数据元素重新计算位置后放入newTable中，链表的顺序与之前的顺序相反

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++;

}

添加方法的具体操作，在添加之前先进行容量的判断，如果当前容量达到了阈值，并且需要存储到Entry[]数组中，先进行扩容操作，扩充的容量为table长度的2倍。重新计算hash值，和数组存储的位置，扩容后的链表顺序与扩容前的链表顺序相反。然后将新添加的Entry实体存放到当前Entry[]位置链表的头部。  
在1.8之前，新插入的元素都是放在了链表的头部位置，但是这种操作在高并发的环境下容易导致死锁，所以1.8之后，新插入的元素都放在了链表的尾部。

获取方法

public V get(Object key) {

if (key == null)

//返回table[0] 的value值

return getForNullKey();

Entry<K,V> entry = getEntry(key);

return null == entry ? null : entry.getValue();

}

final Entry<K,V> getEntry(Object key) {

if (size == 0) {

return null;

}

int hash = (key == null) ? 0 : hash(key);

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

e != null;

e = e.next) {

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

return e;

}

return null;

}

在get方法中，首先计算hash值，然后调用indexFor()方法得到该key在table中的存储位置，得到该位置的单链表，遍历列表找到key和指定key内容相等的Entry，返回entry.value值。

删除方法

public V remove(Object key) {

Entry<K,V> e = removeEntryForKey(key);

return (e == null ? null : e.value);

}

final Entry<K,V> removeEntryForKey(Object key) {

if (size == 0) {

return null;

}

int hash = (key == null) ? 0 : hash(key);

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

Entry<K,V> prev = table[i];

Entry<K,V> e = prev;

while (e != null) {

Entry<K,V> next = e.next;

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k)))) {

modCount++;

size--;

if (prev == e)

table[i] = next;

else

prev.next = next;

e.recordRemoval(this);

return e;

}

prev = e;

e = next;

}

return e;

}

删除操作，先计算指定key的hash值，然后计算出table中的存储位置，判断当前位置是否Entry实体存在，如果没有直接返回，若当前位置有Entry实体存在，则开始遍历列表。定义了三个Entry引用，分别为pre, e ,next。 在循环遍历的过程中，首先判断pre 和 e 是否相等，若相等表明，table的当前位置只有一个元素，直接将table[i] = next = null 。若形成了pre -> e -> next 的连接关系，判断e的key是否和指定的key 相等，若相等则让pre -> next ,e 失去引用。

总结：

**1、实现原理**

* HashMap是基于hashing的原理，我们使用put(key, value)存储对象到HashMap中，使用get(key)从HashMap中获取对象。
* 当我们给put(key, value)方法传递键和值时，它先调用key.hashCode()方法，返回的hashCode值，用于找到bucket位置，来储存Entry对象。
* Map提供了一些常用方法，如keySet()、entrySet()等方法。  
  keySet()方法返回值是Map中key值的集合；entrySet()的返回值也是返回一个Set集合，此集合的类型为Map.Entry。
* “如果两个key的hashcode相同，你如何获取值对象？”答案：当我们调用get(key)方法，HashMap会使用key的hashcode值，找到bucket位置，然后获取值对象。
* “如果有两个值对象，储存在同一个bucket ？”答案：将会遍历链表直到找到值对象。
* “这时会问因为你并没有值对象去比较，你是如何确定确定找到值对象的？”答案：找到bucket位置之后，会调用keys.equals()方法，去找到链表中正确的节点，最终找到要找的值对象。

完美的回答：

* HashMap基于hashing原理，我们通过put(key,value)和get(key)方法储存和获取对象。
* 当储存对象时，我们将键值对传递给put(key,value)方法时，它调用键对象key的hashCode()方法来计算hashcode，然后找到bucket位置，来储存值对象value。
* 当获取对象时，通过key的equals()方法找到正确的键值对key-value，然后返回值对象value。
* HashMap使用链表来解决碰撞问题，当发生碰撞了，对象将会储存在链表的下一个节点中。
* HashMap在每个链表节点中，储存 键值对key-value 对象。
* 当两个不同的键对象key的hashcode相同时，会发生什么？它们会储存在同一个bucket位置的链表中，并通过键对象key的equals()方法用来找到键值对key-value。

因为HashMap的好处非常多，我曾经在我的应用中使用HashMap作为缓存。因为金融领域非常多的运用Java，也出于性能的考虑，我们会经常用到HashMap和ConcurrentHashMap。

2、底层的数据结构

HashMap的底层主要是基于数组和链表来实现的，它之所以有相当快的查询速度主要是因为它是通过计算散列码来决定存储的位置。

* HashMap中主要是通过key的hashCode来计算hash值的，只要hashCode相同，计算出来的hash值就一样。
* 如果存储的对象对多了，就有可能不同的对象所算出来的hash值是相同的，这就出现了所谓的hash冲突。
* 学过数据结构的同学都知道，解决hash冲突的方法有很多，HashMap底层是通过链表来解决hash冲突的。

**补充知识：**

* **HashMap是基于哈希表的 Map 接口的实现。**
* 此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了非同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）
* 此类不保证映射的顺序，特别是它不保证该顺序恒久不变。
* 值得注意的是HashMap不是线程安全的，如果想要线程安全的HashMap，可以通过Collections类的静态方法**synchronizedMap**获得线程安全的HashMap。

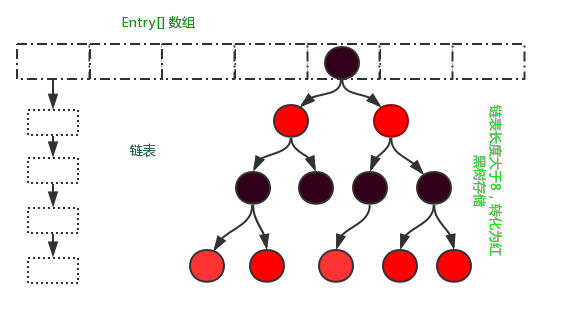
Map map = Collections.synchronizedMap(new HashMap());

* HashMap结合了ArrayList与LinkedList两个实现的优点，虽然HashMap并不会向List的两种实现那样，在某项操作上性能较高，但是在基本操作（get 和 put）上具有稳定的性能。

JDK 1.8的 改变

在Jdk1.8中HashMap的实现方式做了一些改变，但是基本思想还是没有变得，只是在一些地方做了优化，下面来看一下这些改变的地方,数据结构的存储由数组+链表的方式，变化为数组+链表+红黑树的存储方式，在性能上进一步得到提升。

数据存储方式



put方法简单解析

public V put(K key, V value) {

//调用putVal()方法完成

return putVal(hash(key), key, value, false, true);

}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,

boolean evict) {

Node<K,V>[] tab; Node<K,V> p; int n, i;

//判断table是否初始化，否则初始化操作

if ((tab = table) == null || (n = tab.length) == 0)

n = (tab = resize()).length;

//计算存储的索引位置，如果没有元素，直接赋值

if ((p = tab[i = (n - 1) & hash]) == null)

tab[i] = newNode(hash, key, value, null);

else {

Node<K,V> e; K k;

//节点若已经存在，执行赋值操作

if (p.hash == hash &&

((k = p.key) == key || (key != null && key.equals(k))))

e = p;

//判断链表是否是红黑树

else if (p instanceof TreeNode)

//红黑树对象操作

e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

else {

//为链表，

for (int binCount = 0; ; ++binCount) {

if ((e = p.next) == null) {

p.next = newNode(hash, key, value, null);

//链表长度8，将链表转化为红黑树存储

if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st

treeifyBin(tab, hash);

break;

}

//key存在，直接覆盖

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

break;

p = e;

}

}

if (e != null) { // existing mapping for key

V oldValue = e.value;

if (!onlyIfAbsent || oldValue == null)

e.value = value;

afterNodeAccess(e);

return oldValue;

}

}

//记录修改次数

++modCount;

//判断是否需要扩容

if (++size > threshold)

resize();

//空操作

afterNodeInsertion(evict);

return null;

}

总结

* HashMap采用hash算法来决定Map中key的存储，并通过hash算法来增加集合的大小。
* hash表里可以存储元素的位置称为**桶（bucket）**，如果通过key计算hash值发生冲突时，那么将采用链表的形式，来存储元素。
* HashMap的扩容操作是一项很耗时的任务，所以如果能估算Map的容量，最好给它一个默认初始值，避免进行多次扩容。
* HashMap的线程是不安全的，多线程环境中推荐是ConcurrentHashMap。