

java_集合体系之HashMap详解、源码及示例——09

原创 2013年12月25日 14:54:12

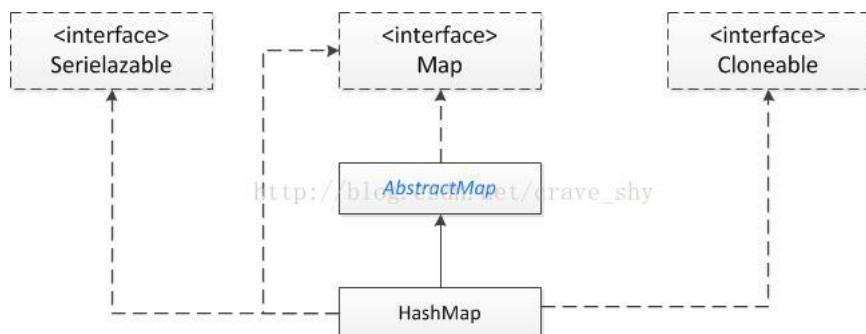
2541

0

12

java_集合体系之HashMap详解、源码及示例——09

一：HashMap结构图



简单说明：

- 1、上图中虚线且无依赖字样、说明是直接实现的接口
- 2、虚线但是有依赖字样、说明此类依赖与接口、但不是直接实现接口
- 3、实线是继承关系、类继承类、接口继承接口
- 4、继承AbstractMap、以键值对的形式存储、操作元素
- 5、实现Serialazable接口、允许使用ObjectInputStream/ObjectOutputStream读取/写入
- 6、实现Cloneable接口、允许克隆HashMap

二：HashMap类简介：

- a) 基于哈希表的Map结构的实现
- b) 线程不安全
- c) 内部映射无序
- d) 允许值为null的key和value

三：HashMap API

- 1、构造方法



Oscar Chen (<http://blog....>)

+关注

(<http://blog.csdn.net/chenghuaying>)

原创

粉丝

喜欢

182

14

2

- > CentOS 集群机器之间ssh免密
(/crave_shy/article/details/72964997)
- > JVM-内存管理-运行时数据区域
(/crave_shy/article/details/56675052)
- > JVM-Blog目录
(/crave_shy/article/details/56675032)
- > JVM-为什么要学JVM
(/crave_shy/article/details/56673439)

更多文章

(<http://blog.csdn.net/chenghuaying>)

在线课程



(http://edu.csdn.net/huiyiCourse/series_detail?utm_source=blog7)

【直播】机器学习&数据挖掘7周实训--韦玮

(http://edu.csdn.net/huiyiCourse/series_detail/54?utm_source=blog7)



(http://edu.csdn.net/combo/detail/471?utm_source=blog7)

【套餐】系统集成项目管理工程师顺利通关--徐朋

(http://edu.csdn.net/combo/detail/471?utm_source=blog7)

```
// 默认构造函数。
HashMap()

// 指定“容量大小”的构造函数
HashMap(int capacity)

// 指定“容量大小”和“加载因子”的构造函数
HashMap(int capacity, float loadFactor)

// 包含“子Map”的构造函数
HashMap(Map<? extends K, ? extends V> map)
```

2、一般方法

void	clear()
Object	clone()
boolean	containsKey(Object key)
boolean	containsValue(Object value)
Set<Entry<K, V>>	entrySet()
V	get(Object key)
boolean	isEmpty()
Set<K>	keySet()
V	put(K key, V value)
void	putAll(Map<? extends K, ? extends V> map)
V	remove(Object key)
int	size()
Collection<V>	values()

四：HashMap 源码分析

简单说明：

- 1、对哈希表要有简单的认识、
- 2、HashMap是通过“拉链法”解决哈希冲突的
- 3、理解HashMap源码中的关键部分、Entry实体类的行为、属性。Entry的存储方式、HashMap的扩容方式、HashMap内部关于获取新的hash code的算法。

```

package com.chy.collection.core;
import java.io.IOException;
import java.io.Serializable;
import java.util.AbstractMap;
import java.util.Collections;
import java.util.ConcurrentModificationException;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;
import java.util.NoSuchElementException;
import java.util.TreeMap;

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable {

    /** 初始化HashMap时默认的容量、必须是2的幂*/
    static final int DEFAULT_INITIAL_CAPACITY = 16;

    /** HashMap容量最大值、必须是2幂、并且要小于2的30次方、如果容量超过这个值、将会被这个值代替*/
    static final int MAXIMUM_CAPACITY = 1 << 30;

    /** 默认加载因子*/
    static final float DEFAULT_LOAD_FACTOR = 0.75f;

    /** 存储数据的Entry数组，长度是2的幂。Entry的本质是一个单向链表*/
    transient Entry[] table;

    /** 当前HashMap中键值对的总数*/
    transient int size;

    /** HashMap容量的阈值、用于判断是否要rehash（threshold=容量*加载因子）*/
    int threshold;

    /** 加载因子的实际值*/
    final float loadFactor;

    /** HashMap被改变的次数*/
    transient volatile int modCount;

    /** 使用指定的容量、加载因子初始化HashMap*/
    public HashMap(int initialCapacity, float loadFactor) {
        //容量初始值是否合法
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " + initialCapacity);
        //容量初始值是否超过最大值
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        //判断加载因子是否合法
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " + loadFactor);

        // 查找一个大于初始化HashMap容量的2的幂的值
        int capacity = 1;
        while (capacity < initialCapacity)
            capacity <= 1;

        //初始化加载因子
        this.loadFactor = loadFactor;
        //初始化HashMap阈值
        threshold = (int)(capacity * loadFactor);
        //初始化HashMap用于存放键值对的数组table
        table = new Entry[capacity];
        init();
    }

    /** 使用指定初始容量、默认加载因子创建HashMap*/
    public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }

    /**使用默认初始容量 16、默认加载因子0.75创建HashMap*/
    public HashMap() {
        this.loadFactor = DEFAULT_LOAD_FACTOR;
        threshold = (int)(DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR);
        table = new Entry[DEFAULT_INITIAL_CAPACITY];
        init();
    }

    /** 创建包含指定传入Map的所有键值对创建HashMap、使用默认加载因子、使用处理后的容量*/
    public HashMap(Map<? extends K, ? extends V> m) {

```

```

        this(Math.max((int) (m.size() / DEFAULT_LOAD_FACTOR) + 1, DEFAULT_INITIAL_CAPACITY), DEFAULT_LOAD_FACTOR);
        //将Map中所有键值对放入到当前的HashMap中
        putAllForCreate(m);
    }
    // internal utilities

    void init() {
    }

    //使用指定的运算方式获取传入的hash值的新的hash值、key为null的值的hash值是0、index也是0、所以HashMap中只有一个值为null的key
    static int hash(int h) {
        h ^= (h >>> 20) ^ (h >>> 12);
        return h ^ (h >>> 7) ^ (h >>> 4);
    }

    //根据传入的hash值与数组长度获取hash值代表的键在table中的索引
    static int indexFor(int h, int length) {
        // 保证返回值的索引值小于length
        return h & (length-1);
    }

    /** 返回当前HashMap中键值对个数*/
    public int size() {
        return size;
    }

    /** 判断当前HashMap是否为空*/
    public boolean isEmpty() {
        return size == 0;
    }

    /** 获取指定key对应的value*/
    public V get(Object key) {
        if (key == null)
            return getForNullKey();
        //获取key的哈希值
        int hash = hash(key.hashCode());
        // 在“该hash值对应的链表”上查找“键值等于key”的元素
        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.equals(k)))
                return e.value;
        }
        return null;
    }

    /** 获取key为null的实体的value、key为null时只会放在table开头、即索引为0的位置
     * 只有两种结果、一种是不存在key的值为null、另一种就是table第一个实体的key是null
     */
    private V getForNullKey() {
        for (Entry<K,V> e = table[0]; e != null; e = e.next) {
            if (e.key == null)
                return e.value;
        }
        return null;
    }

    /** 是否包含传入的 key*/
    public boolean containsKey(Object key) {
        return getEntry(key) != null;
    }

    /** 获取指定key所代表的映射Entry*/
    final Entry<K,V> getEntry(Object key) {
        //获取key对应的hash值、再通过hash值找到其所在到table中的索引、获取Entry
        int hash = (key == null) ? 0 : hash(key.hashCode());
        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;
    }
}

```

```

/** 将指定键值对放入HashMap中、如果HashMap中存在key、则替换key映射的value*/
public V put(K key, V value) {
    //如果key是null、特殊处理、即只操作table中第一个元素
    if (key == null)
        return putForNullKey(value);
    //使用新的算法得到key的hash值、进而得到key在table中存储的索引值
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        //如果key已存在、则使用新的value替换原来的value、并返回被替换的value
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    //如果key不存在、则将键值对添加到HashMap中、
    modCount++;
    addEntry(hash, key, value, i);
    return null;
}

/** 将key为null的键值对添加到table索引为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;
}

/** 供内部构造方法调用、用于创建HashMap、而put()是供外部调用、用于向HashMap中存入键值对*/
private void putForCreate(K key, V value) {
    int hash = (key == null) ? 0 : hash(key.hashCode());
    int i = indexFor(hash, table.length);

    /** 若该HashMap表中存在“键值等于key”的元素，则替换该元素的value值*/
    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;
        }
    }

    createEntry(hash, key, value, i);
}

/** 与上面方法相同、用于使用指定的m创建HashMap、用于clone()*/
private void putAllForCreate(Map<? extends K, ? extends V> m) {
    //使用迭代器迭代每一个键值对、然后调用putForCreate(K key, V value)构造HashMap
    for (Iterator<? extends Map.Entry<? extends K, ? extends V>> i = m.entrySet().iterator();
i.hasNext(); ) {
        Map.Entry<? extends K, ? extends V> e = i.next();
        putForCreate(e.getKey(), e.getValue());
    }
}

/** rehash当前HashMap、此方法会在HashMap容量达到阈值的时候自动调用、*/
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    //如果容量达到最大值、则修改阈值
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }

    //构造新的用于存储Entry的table、并且将原来table中所有元素转移到新的table中、修改HashMap的
    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable);
}

```

```

        table = newTable;
        threshold = (int)(newCapacity * loadFactor);
    }

    /** 将原来table中所有元素转移到新的table中*/
    void transfer(Entry[] newTable) {
        Entry[] src = table;
        int newCapacity = newTable.length;
        for (int j = 0; j < src.length; j++) {
            Entry<K,V> e = src[j];
            if (e != null) {
                src[j] = null;
                do {
                    Entry<K,V> next = e.next;
                    int i = indexFor(e.hash, newCapacity);
                    e.next = newTable[i];
                    newTable[i] = e;
                    e = next;
                } while (e != null);
            }
        }
    }

    /** 将m中所有键值对存储到HashMap中*/
    public void putAll(Map<? extends K, ? extends V> m) {
        int numKeysToBeAdded = m.size();
        if (numKeysToBeAdded == 0)
            return;

        /*
         * 计算容量是否满足添加元素条件
         * 若不够则将原来容量扩容2倍
         */
        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);
        }

        //使用迭代器迭代m中每个元素、然后添加到HashMap中
        for (Iterator<? extends Map.Entry<? extends K, ? extends V>> i = m.entrySet().iterator();
i.hasNext(); ) {
            Map.Entry<? extends K, ? extends V> e = i.next();
            put(e.getKey(), e.getValue());
        }
    }

    /** 删除“键为key”的元素*/
    public V remove(Object key) {
        Entry<K,V> e = removeEntryForKey(key);
        return (e == null ? null : e.value);
    }

    /** 删除“键为key”的元素*/
    final Entry<K,V> removeEntryForKey(Object key) {
        int hash = (key == null) ? 0 : hash(key.hashCode());
        int i = indexFor(hash, table.length);
        Entry<K,V> prev = table[i];
        Entry<K,V> e = prev;
        //本质是“删除单向链表中的节点”
        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;
}

/**
 * Special version of remove for EntrySet.
 */
final Entry<K,V> removeMapping(Object o) {
    if (!(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.hashCode());
    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;
}

/** 删除所有键值对*/
public void clear() {
    modCount++;
    Entry[] tab = table;
    for (int i = 0; i < tab.length; i++)
        tab[i] = null;
    size = 0;
}

/** 判断是否包含value*/
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;
}

/** 返回一个含有当前HashMap中所有键值对的Object*/
public Object clone() {
    HashMap<K,V> result = null;
    try {
        result = (HashMap<K,V>)super.clone();
    } catch (CloneNotSupportedException e) {
        // assert false;
    }
    result.table = new Entry[table.length];
    result.entrySet = null;
    result.modCount = 0;
    result.size = 0;
}

```

```

        result.init();
        result.putAllForCreate(this);
        return result;
    }
    /**
     * Entry是单向链表。
     * 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(Object o), hashCode()这些函数
     */
    static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
        //指向下一节点的引用
        Entry<K,V> next;
        final int hash;

        /** 创建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();
            //先比较键是否相同、键相同在比较键所对应的值是否相同、值也相同则返回true、否则返回false
            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;
        }

        //重写hashCode()方法的实现
        public final int hashCode() {
            return (key==null ? 0 : key.hashCode()) ^ (value==null ? 0 : value.hashCode());
        }

        //重写toString()
        public final String toString() {
            return getKey() + "=" + getValue();
        }

        /** 当向HashMap中添加元素时，此方法被调用*/
        void recordAccess(HashMap<K,V> m) {
        }

        /** 当从HashMap中删除元素时、此方法被调用*/
        void recordRemoval(HashMap<K,V> m) {
        }
    }

    /** 新增Entry。将“key-value”插入指定位置，bucketIndex是位置索引。 */
    void addEntry(int hash, K key, V value, int bucketIndex) {
        //获取bucketIndex处键值对
        Entry<K,V> e = table[bucketIndex];
        //将bucketIndex处的键值对设置成新的Entry、并且将e设置成下一个节点
        table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
        //若处理后的HashMap阈值小于HashMap实际大小、则扩容
    }

```



```

        if (size++ >= threshold)
            resize(2 * table.length);
    }

    /** 相对于上面的方法少了关于容量的处理、这也间接说明两个方法的用法的区别
     * 1、addEntry()是在HashMap创建好之后、向HashMap中添加键值对的时候、比如调用put()时、进而调用addEntry()、这种不知道添加之后HashMap原来容量是否够用
     * 所以要对HashMap容量进行判断处理、进而决定是否扩容
     * 2、createEntry()此方法使用在已经知道向HashMap中添加元素之后、HashMap的容量仍然不会达到阈值的情况、比如使用Map构造HashMap、或者克隆HashMap时、
     * HashMap容量足以存放下Map中所有元素、或者克隆的所有元素、因此不必在扩容。
     */
    void createEntry(int hash, K key, V value, int bucketIndex) {
        //获取bucketIndex处键值对
        Entry<K,V> e = table[bucketIndex];
        //将bucketIndex处的键值对设置成新的Entry、并且将e设置成下一个节点
        table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
        size++;
    }
    /**
     * 抽象类、用于迭代HashMap、
     * 包含三种视图的迭代“keySet”、“valueCollection”、“Entry<K, V>”三个迭代器
     */
    private abstract class HashIterator<E> implements Iterator<E> {
        Entry<K,V> next;           // 下一个元素
        int expectedModCount;      // 用于实现fail-fast机制
        int index;                 // 当前索引
        Entry<K,V> current;        // 当前元素

        HashIterator() {
            expectedModCount = modCount;
            if (size > 0) { // advance to first entry
                Entry[] t = table;
                //将next指向第一个不为null的元素、table的索引是从0开始的、下面这句是index依次从0开始遍历直到下个元素不为null则结束循环
                while (index < t.length && (next = t[index++]) == null) ;
            }
        }

        //查看是否有下一个
        public final boolean hasNext() {
            return next != null;
        }

        //获取下一个元素
        final Entry<K,V> nextEntry() {
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
            Entry<K,V> e = next;
            if (e == null)
                throw new NoSuchElementException();
            /*
             * 将原来当前元素的下一个元素引用“next”指向下一个元素
             * Entry是单向链表结构、当下一个节点不为null的时候、将next指向下一个节点、否则继续寻找单向链表中下一个不是null的节点
             */
            if ((next = e.next) == null) {
                Entry[] t = table;
                while (index < t.length && (next = t[index++]) == null)
                    ;
            }
            //将原来的下一个节点设置成当前节点、并作为结果返回
            current = e;
            return e;
        }
    }

    //删除当前节点、
    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;
    }
}

```

```

// value的迭代器
private final class ValueIterator extends HashIterator<V> {
    public V next() {
        return nextEntry().value;
    }
}

// key的迭代器
private final class KeyIterator extends HashIterator<K> {
    public K next() {
        return nextEntry().getKey();
    }
}

// Entry的迭代器
private final class EntryIterator extends HashIterator<Map.Entry<K,V>> {
    public Map.Entry<K,V> next() {
        return nextEntry();
    }
}

//返回一个"key迭代器"
Iterator<K> newKeyIterator() {
    return new KeyIterator();
}

//返回一个"Value迭代器"
Iterator<V> newValueIterator() {
    return new ValueIterator();
}

//返回一个"Entry迭代器"
Iterator<Map.Entry<K,V>> newEntryIterator() {
    return new EntryIterator();
}

// Views

//当前HashMap的Entry对应的Set
private transient Set<Map.Entry<K,V>> entrySet = null;

/** "key"对应的set*/
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();
    }
}

/** value对用的Collection*/
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();
        }
    }

    /** Map.Entry<K, V>对应的Set*/
    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HashMap的“总的容量，实际容量，所有的Entry”都写入到输出流中*/
    private void writeObject(java.io.ObjectOutputStream s)
        throws IOException
    {
        Iterator<Map.Entry<K,V>> i =
            (size > 0) ? entrySet0().iterator() : null;

        // Write out the threshold, loadfactor, and any hidden stuff
        s.defaultWriteObject();

        // Write out number of buckets
        s.writeInt(table.length);

        // Write out size (number of Mappings)
        s.writeInt(size);

        // Write out keys and values (alternating)
        if (i != null) {
            while (i.hasNext()) {
                Map.Entry<K,V> e = i.next();
                s.writeObject(e.getKey());
                s.writeObject(e.getValue());
            }
        }
    }

    private static final long serialVersionUID = 362498820763181265L;

    /** 将HashMap的“总的容量，实际容量，所有的Entry”都写入到输出流中*/
    private void readObject(java.io.ObjectInputStream s)
        throws IOException, ClassNotFoundException
    {
        // Read in the threshold, loadfactor, and any hidden stuff
        s.defaultReadObject();

        // Read in number of buckets and allocate the bucket array;
        int numBuckets = s.readInt();
        table = new Entry[numBuckets];

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

        // Read in size (number of Mappings)
        int size = s.readInt();
    }

```

```

        // Read the keys and values, and put the mappings in the HashMap
        for (int i=0; i<size; i++) {
            K key = (K) s.readObject();
            V value = (V) s.readObject();
            putForCreate(key, value);
        }
    }

    // These methods are used when serializing HashSets
    //返回HashMap的总容量
    int capacity() { return table.length; }
    //返回HashMap的加载因子
    float loadFactor() { return loadFactor; }
}

```

总结：

1、数据结构：HashMap是以哈希表的形式存储数据的、并且是通过“拉链法”解决冲突、HashMap中存储的Entry实现了Map.Entry<K,V>关于HashMap存储元素的结构及hash值的获取算法：

```

//使用指定的运算方式获取传入的hash值的新的hash值、key为null的值的hash值是0、index也是0、所以HashMap中只有一个值为null的key
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}

//根据传入的hash值与数组长度获取hash值代表的键在table中的索引
static int indexFor(int h, int length) {
    // 保证返回值的索引值小于length
    return h & (length-1);
}

```

2、与容量有关的内容HashMap

a) 的实例有两个参数影响其性能：初始容量 和加载因子。容量 是哈希表中桶的数量，初始容量只是哈希表在创建时的容量。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。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行rehash 操作（即重建内部数据结构），从而哈希表将具有大约两倍的桶数。

b) 默认加载因子 (0.75) 在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查询成本（在大多数HashMap 类的操作中，包括 get 和put 操作，都反映了这一点）。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少 rehash 操作次数。如果初始容量大于最大条目数除以加载因子，则不会发生 rehash 操作。

c) 如果迭代性能很重要，则不要将初始容量设置得太高（或将加载因子设置得太低）。

d) 如果很多映射关系要存储在 HashMap 实例中，则相对于按需执行自动的 rehash 操作以增大表的容量来说，使用足够大的初始容量创建它将使得映射关系能更有效地存储。

e) 当使用HashMap对外提供的存入键值对的方法put()、putAll()时、HashMap内部会检测HashMap容量是否达到阈值、进而判断是否需要扩容。与此有关的一系列方法源码汇总（包括内部方法）

```

/** rehash当前HashMap、此方法会在HashMap容量达到阈值的时候自动调用、*/
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    //如果容量达到最大值、则修改阈值
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }

    //构造新的用于存储Entry的table、并且将原来table中所有元素转移到新的table中、修改HashMap的
    阈值

    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable);
    table = newTable;
    threshold = (int)(newCapacity * loadFactor);
}

/** 新增Entry。将“key-value”插入指定位置，bucketIndex是位置索引。 */
void addEntry(int hash, K key, V value, int bucketIndex) {
    //获取bucketIndex处键值对
    Entry<K,V> e = table[bucketIndex];
    //将bucketIndex处的键值对设置成新的Entry、并且将e设置成下一个节点
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
    //若处理后的HashMap阈值小于HashMap实际大小、则扩容
    if (size++ >= threshold)
        resize(2 * table.length);
}

/** 将指定键值对放入HashMap中、如果HashMap中存在key、则替换key映射的value*/
public V put(K key, V value) {
    //如果key是null、特殊处理、即只操作table中第一个元素
    if (key == null)
        return putForNullKey(value);
    //使用新的算法得到key的hash值、进而得到key在table中存储的索引值
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        //如果key已存在、则使用新的value替换原来的value、并返回被替换的value
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    //如果key不存在、则将键值对添加到HashMap中、
    modCount++;
    addEntry(hash, key, value, i);
    return null;
}

/** 将key为null的键值对添加到table索引为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;
}

/** 将m中所有键值对存储到HashMap中*/
public void putAll(Map<? extends K, ? extends V> m) {
    int numKeysToBeAdded = m.size();
    if (numKeysToBeAdded == 0)
        return;

    /**
     * 计算容量是否满足添加元素条件
     * 若不够则将原来容量扩容2倍
     */
}

```

```

        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);
        }

        //使用迭代器迭代m中每个元素、然后添加到HashMap中
        for (Iterator<? extends Map.Entry<? extends K, ? extends V>> i = m.entrySet().iterator();
i.hasNext(); ) {
            Map.Entry<? extends K, ? extends V> e = i.next();
            put(e.getKey(), e.getValue());
        }
    }
}

```

f) addEntry()与createEntry()区别

addEntry()是在HashMap创建好之后、向HashMap中添加键值对的时候、比如调用put()时、进而调用addEntry()、这种不知道添加之后HashMap原来容量是否够用所以要对HashMap容量进行判断处理、进而决定是否扩容

createEntry()此方法使用在已经知道向HashMap中添加元素之后、HashMap的容量仍然不会达到阈值的情况、比如使用Map构造HashMap、或者克隆HashMap时、HashMap容量足以存放下Map中所有元素、或者克隆的所有元素、因此不必在扩容。

3、HashMap迭代有关：从源码中可看出、HashMap内部提供一个抽象类HashIterator、此类实现Iterator接口、并且提供了Iterator接口必须实现的next() hasNext() remove()方法、HashMap有三种迭代视图、一个是所有键的集合Set、一个是所有值的集合、一个是所有键值对的集合、获取这三个迭代器的方法是分别构造KeyIterator、ValueIterator、EntryIterator、而这三个类都继承自HashIterator、只是分别提供自己特有的next()方法、而剩下的hasNext()、remove()方法都是使用HashIterator的实现、HashIterator内部在构造方法中会将其一个指向下一个节点的引用next初始化、根据next的指向是否为空实现hasNext()、remove()的实现就是调用内部方法将当前节点删除（做的是关于删除单向链表的一个节点的操作）、具体可以参见源码中关于HashIterator、KeyIterator、ValueIterator部分。

五：HashMap 示例

1、遍历方式：

a) 使用keySet()遍历

```

Set<K> keySet = hashMap.keySet();
Iterator<K> it = keySet.iterator();

```

b) 使用values()遍历

```

Collection<K> values = hashMap.values();
Iterator<K> it = values.iterator();

```

c) 使用entrySet()遍历

```

Set<Entry<K, V>> entrySet = hashMap.entrySet();
Iterator<K> it = entrySet.iterator();

```

2、迭代示例：

```

package com.chy.collection.example;

import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Set;
import java.util.Map.Entry;

public class EragodicHashMap {

    private static HashMap<String, Integer> hashMap = new HashMap<String, Integer>();
    private static Set<String> keySet;
    private static Collection<Integer> values;
    private static Set<Entry<String, Integer>> entrySet;
    static{
        for (int i = 0; i < 10; i++) {
            hashMap.put("" + i, i);
        }
    }

    /**
     * iterate HashMap by keySet
     */
    private static void testKeySet(){
        keySet = hashMap.keySet();
        Iterator<String> it = keySet.iterator();
        while(it.hasNext()){
            System.out.println("key " + it.next());
        }
        System.out.println("=====");
    }

    /**
     * iterate HashMap by values
     */
    private static void testValues(){
        values = hashMap.values();
        Iterator<Integer> it = values.iterator();
        while(it.hasNext()){
            System.out.println("value : " + it.next());
        }
        System.out.println("=====");
    }

    /**
     * iterate HashMap by EntrySet
     */
    private static void testEntrySet(){
        entrySet = hashMap.entrySet();
        Iterator<Entry<String, Integer>> it = entrySet.iterator();
        while(it.hasNext()){
            System.out.println("entry : " + it.next());
        }
        System.out.println("=====");
    }

    /**
     * common methods of three views
     */
    private static void testViewsCommonMethods(){
        System.out.println("keySet size: " + keySet.size() + " values size: " + values.size() + " entrySet size: " + entrySet.size());
        System.out.println("keySet contains 1 ? " + keySet.contains("1") + " values contain s 1 ? " + values.contains("1") + " entrySet contains 1 ? " + entrySet.contains("1"));
        System.out.println("keySet remove 1 ? " + keySet.remove("1") + " values remove 1 ? " + values.remove("1") + " entrySet remove 1 ? " + entrySet.remove("1"));
        keySet.clear();
        values.clear();
        entrySet.clear();
        System.out.println("keySet size: " + keySet.size() + " values size: " + values.size() + " entrySet size: " + entrySet.size());
    }

    public static void main(String[] args) {
        testEntrySet();
        testKeySet();
        testValues();
        testViewsCommonMethods();
    }
}

```

3、API示例：

```
package com.chy.collection.example;

import java.util.HashMap;

@SuppressWarnings("all")
public class HashMapTest {

    /**
     * 测试构造方法、下面四个方法效果相同、
     */
    private static void testConstructor(){
        //use default construct
        HashMap<String, String> hashMap1 = new HashMap<String, String>();
        //use specified initCapacity
        HashMap<String, String> hashMap2 = new HashMap<String, String>(16);
        //use specified initCapacity and loadFactor
        HashMap<String, String> hashMap3 = new HashMap<String, String>(16, 0.75f);
        //use specified Map
        HashMap<String, String> hashMap4 = new HashMap<String, String>(hashMap1);
    }

    /**
     * 测试API方法
     */
    private static void testAPI(){
        //初始化、键-值都为字符串"1"的hashMap
        HashMap<String, String> hashMap = new HashMap<String, String>();
        for (int i = 0; i < 10; i++) {
            hashMap.put(""+i, ""+i);
        }
        /**
         * 向hashMap中添加键为null的键值对
         * 只会在HashMap的index为0处、保存一个键为null的键值对、键为null、值为最后一次添加
         的键值对的值。
         */
        hashMap.put(null, null);
        hashMap.put(null, "n");
        System.out.println(hashMap.size());
        System.out.println(hashMap);

        //是否包含键"1"
        System.out.println("hashMap contains key ? " + hashMap.containsKey("1"));
        //是否包含值"1"
        System.out.println("hashMap contains value ? " + hashMap.containsValue("1"));
        //获取键为"1"的值
        System.out.println("the value of key=1 " + hashMap.get("1"));
        //将键为"1"的值修改成"11"
        hashMap.put("1", "11");

        //将hashMap复制到hashMap1中
        HashMap<String, String> hashMap1 = (HashMap<String, String>)hashMap.clone();
        //将hashMap1所有键值对复制到hashMap中
        hashMap.putAll(hashMap1);
        System.out.println(hashMap);
        //如果有二十个元素、因为他不会再添加重复元素
        //如果hashMap非空、则清空
        if(!hashMap.isEmpty()){
            hashMap.clear();
        }

        System.out.println(hashMap.size());
    }

    public static void main(String[] args) {
        testAPI();
    }
}
```


更多内容：[java_集合体系之总体目录——00](http://blog.csdn.net/crave_shy/article/details/1741679)
(http://blog.csdn.net/crave_shy/article/details/1741679)

版权声明：本文为博主原创文章，未经博主允许不得转载。



标签：HashMap (<http://so.csdn.net/so/search/s.do?q=HashMap&t=blog>) /
Map (<http://so.csdn.net/so/search/s.do?q=Map&t=blog>) /
哈希表 (<http://so.csdn.net/so/search/s.do?q=哈希表&t=blog>) /
Iterator (<http://so.csdn.net/so/search/s.do?q=Iterator&t=blog>) /
java集合 (<http://so.csdn.net/so/search/s.do?q=java集合&t=blog>) /

0条评论

 qq_36596145 (http://my.csdn.net/qq_36596145)





发表评论

暂无评论

相关文章推荐



SpringMVC组件及配置详解 (/account090909/article/details/60766397)

上传文件解析器 MultipartResolver 加载该组件时查找名为multipartResolver类型为MultipartResolver的Bean。作为该类型组件。该组件没有默认...

 account090909 2017-03-08 09:04  254



java_集合体系之总体目录——00 (/crave_shy/article/details/17416791)

摘要：java集合系列目录、不断更新中、、、水平有限、总有不足、误解的地方、请多包涵、也希望多提意见、多多讨论 ^_^

 chenghuaying 2013-12-19 15:41  3210



HashMap实现详解 (/u011225629/article/details/48735331)

1. HashMap概述： HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久...

 u011225629 2015-09-25 17:22  520

Java 集合系列10之 HashMap详细介绍(源码解析)和使用示例 (/wangtaomtk/article/details/51815552)

概要 这一章，我们对HashMap进行学习。我们先对HashMap有个整体认识，然后再学习它的源码，最后再通过实例来学会使用HashMap。内容包括：第1部分 HashMap介绍 第2部分 Ha...

 wangtaomtk 2016-07-03 16:05  268

关于hashmap (/benjinglin/article/details/18843453)

HashMap是基于哈希表的map接口的非同步实现。允许使用null值和null键。不保证映射的顺序，特别是它不保证该顺序恒久不变。HashMap实际上是一个链表散列的数据结构，即数组和链表的结合...



u013282523 2014-01-28 16:38 421

JavaAPI之ConcurrentHashMap (/u010142437/article/details/51628169)

结构 java.util.concurrent 类 ConcurrentHashMap java.lang.Object java.util.AbstractMap java.util...



u010142437 2016-06-10 18:35 1716

HashMap笔记 (/u013919103/article/details/52303508)

今天的项目中看别人的代码使用treeMap(), 不知道为什么使用treeMap而不使用hashMap, 因此搜了一下treeMap和hashMap。这篇文章写的很详细 在此处记录一下自己的理解, 现...



u013919103 2016-08-24 19:55 451

hashMap的原理 深入理解 (/jamebing/article/details/73824319)

首先再次强调hashCode (==) 和equals的真正含义 (我记得以前有人会说, equals是判断对象内容, hashCode是判断是否相等之类): equals: 是否同一个对象实例。注意, 是...



JameBing 2017-06-27 22:15 91

HashMap的实现原理-博客总结 (/guozhao265/article/details/73348355)

hashmap实现原理是面试时的常见问题, 现在找了一篇写的比较好的相关博客引用过来精读一下, 并稍作修改和做笔记。



guozhao265 2017-06-16 18:11 56

深入学习集合之HashMap实现原理 (/lzxyzq/article/details/51393965)

深入学习集合之HashMap实现原理1. HashMap概述: HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作, 并允许使用null值和null键。此类不保证映射的...



lzxyzq 2016-05-13 11:35 368

java_集合体系之Hashtable详解、源码及示例——10 (/crave_shy/article/details/17583001)

摘要: 本文通过Hashtable的结构图来说明Hashtable的结构、以及所具有的功能。根据源码给出Hashtable所具有的特性、结合源码对其特性深入理解、给出示例体会使用方式。



chenghuaying 2013-12-26 15:29 1840

Java_io体系之PrintStream简介、走进源码及示例——09 (http://810364804.iteye.com/blog/1992806)

Java_io体系之PrintStream简介、走进源码及示例——09 PrintStream 1、类功能简介: 字节打印流、功能很强大的一个装饰流、作为FilterInputStream的一个子类、他为底层输出流提供的装饰是可以打印各种java类型的数据、包括对象、这里首次接触的时候会有个误解、觉得PrintStream是将结果打印到控制台的、当然、Print



810364804 2013-11-27 21:28 174

java_集合体系之:LinkedList详解、源码及示例——04 (/crave_shy/article/details/17440835)

摘要：本文通过对LinkedList内部存储数据的结构、LinkedList的结构图、示例、源码、多方面深入分析LinkedList的特性和使用方法。



chenghuaying 2013-12-20 15:11 6349

JNI书籍特供 (<http://473687880.iteye.com/blog/1964909>)

<p style="margin-top: 0px; margin-bottom: 0px; padding-top: 0px; padding-



473687880 2013-10-14 14:18 510

java_集合体系之ArrayList详解、源码及示例——03 (<http://810364804.iteye.com/blog/1992789>)

java_集合体系之ArrayList详解、源码及示例——03 — : ArrayList结构图 <img

src="http://img.blog.csdn.net/20131220102938781?

watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvY3JhdmVfc2h5/font/5a6L5L2T/fontsize/400/fill/10JBQkFCMA==/dissolve/



810364804 2013-12-20 10:56 163

java_集合体系之总体目录——00 (crave_shy/article/details/17416791)

摘要：java集合系列目录、不断更新中、、、水平有限、总有不足、误解的地方、请多包涵、也希望多提意见、多多讨论 ^_^



chenghuaying 2013-12-19 15:41 3210

JNI_编程技术__网文整理 (<http://xiaoruanjian.iteye.com/blog/1367476>)

<!--[if supportFields]><span style='mso-element:field-
begin'> TOC /o
 1-2 /h /z /u <spa



wapysun 2010-10-26 15:24 795

java_集合体系之Vector详解、源码及示例——05 (crave_shy/article/details/17504279)

摘要：本文通过对Vector的结构图中涉及到的类、接口来说明Vector的特性、通过源码来深入了解Vector各种功能的实现原理、通过示例加深对Vector的理解。



chenghuaying 2013-12-23 14:40 2129

Java_io体系之OutputStreamWriter、InputStreamReader简介、走进源码及示例——17 (<http://810364804.iteye.com/blog/1992797>)

Java_io体系之OutputStreamWriter、InputStreamReader简介、走进源码及示例——17 — :

OutputStreamWriter 1、类功能简介：输入字符转换流、是输入字节流转向输入字符流的桥梁、用于将输入字节流转换成输入



810364804 2013-12-10 09:51 60

计算机科学精彩帖子收集 (xuxu120/article/details/51254870)

转载自：<http://blog.csdn.net/unix21/article/details/8492617> linux源码 LXR 源自 “the Linux Cross R...



xuxu120 2016-04-26 22:26 2116

