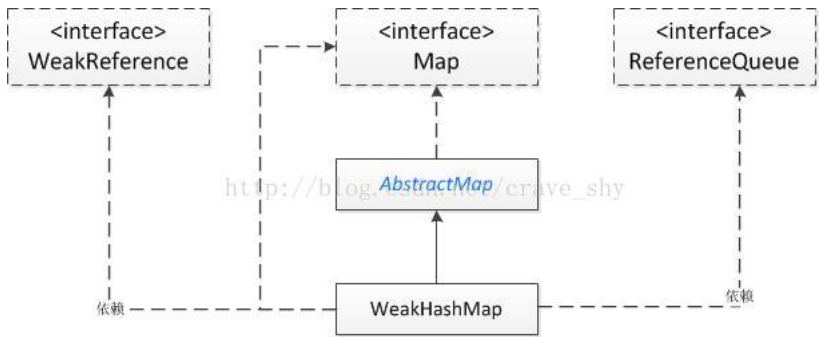


java_集合体系之WeakHashMap详解、源码及示例——11

原创 2013年12月27日 14:58:53 2054 0 10

java_集合体系之WeakHashMap详解、源码及示例——11

一：WeakHashMap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WeakReference接口、具有弱引用对象特性
- 6、实现ReferenceQueue接口、具有将引用向其注册引用功能、进而使用其提供的方法操作引用。
- 7、上面两个接口通常会一起使用、用来处理弱引用对象、比如本文中的WeakHashMap。

补充：弱引用（WeakReference）

如果一个对象只具有弱引用，那就类似于可有可无的生活用品。只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

二：WeakHashMap类简介：

- 1、基于哈希表的Map结构的实现
- 2、线程不安全
- 3、内部映射无序
- 4、允许值为null的key和value
- 5、当WeakHashMap中的键不再其他的强引用的时候、此键表示的键值对会被GC回收、这里可能有点难理解、后面会有更详细的说明。



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

+ 关注

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

原创 粉丝 喜欢
182 14 4

- > 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)

三：WeakHashMap API

1、构造方法

```
// 默认构造函数。
WeakHashMap()

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

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

// 包含“子Map”的构造函数
WeakHashMap(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：WeakHashMap 源码分析

说明：

- 1、对哈希表要有简单的认识。
 - 2、WeakHashMap是通过“拉链法”解决哈希冲突的。
 - 3、理解WeakHashMap源码中的关键部分、Entry实体类的行为、属性。Entry的存储方式、WeakHashMap的扩容方式、WeakHashMap内部关于获取新的hash code的算法。
 - 4、理解他如何实现弱引用的。
 - 5、理解三种视图的内部获取的方式。
 - 6、WeakHashMap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。
 - 7、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WeakHashMap 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。
 - 8、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）。
 - 9、如果很多映射关系要存储在 WeakHashMap 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。
- 当使用WeakHashMap对外提供的存入键值对的方法put()、putAll()时、WeakHashMap内部会检测WeakHashMap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）

```

package com.chy.collection.core;
import java.lang.ref.ReferenceQueue;
import java.lang.ref.WeakReference;
import java.util.Collections;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class WeakHashMap<K,V> extends AbstractMap<K,V> implements Map<K,V> {

    /** 初始化HashMap时默认的容量、必须是2的幂*/
    private 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*/
    private static final int MAXIMUM_CAPACITY = 1 << 30;

    /** 默认加载因子*/
    private 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*/
    private Entry[] table;

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

    /** 当前HashMap中键值对的总数*/
    private int threshold;

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

    /** 引用队列、垃圾回收器将已注册的引用对象添加到该队列中。
     * 在这里结合WeakReference使用、用于记录WeakHashMap中的弱引用键
     */
    private final ReferenceQueue<K> queue = new ReferenceQueue<K>();

    private volatile int modCount;

    /** 使用指定的容量、加载因子初始化WeakHashMap*/
    public WeakHashMap(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);

        int capacity = 1;
        while (capacity < initialCapacity)
            capacity <= 1;
        table = new Entry[capacity];
        this.loadFactor = loadFactor;
        threshold = (int)(capacity * loadFactor);
    }

    /** 使用指定初始容量、默认加载因子创建WeakHashMap*/
    public WeakHashMap(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WeakHashMap*/
    public WeakHashMap() {
        this.loadFactor = DEFAULT_LOAD_FACTOR;
        threshold = (int)(DEFAULT_INITIAL_CAPACITY);
        table = new Entry[DEFAULT_INITIAL_CAPACITY];
    }

    /** 创建包含指定传入Map的所有键值对创建WeakHashMap、使用默认加载因子、使用处理后的容量*/
    public WeakHashMap(Map<? extends K, ? extends V> m) {
        this(Math.max((int) (m.size() / DEFAULT_LOAD_FACTOR) + 1, 16),
            DEFAULT_LOAD_FACTOR);
        putAll(m);
    }

    // internal utilities

    /** 当key为null时使用的值
     * 因为WeakHashMap中允许“null的key”，若直接插入“null的key”，将其当作弱引用时，会被删除。
     */
    private static final Object NULL_KEY = new Object();

```

```

/** 当key为null时使用的值特殊处理、将其使用静态不可变量“new Object()”代替
 * 在put中会被调用、防止将null作为的key被当作“弱引用键”被GC回收。
 */
private static Object maskNull(Object key) {
    return (key == null ? NULL_KEY : key);
}

/**
 * 还原对“null的key”的特殊处理
 * 在get(key)中被调用、返回key为null的value。
 */
private static <K> K unmaskNull(Object key) {
    return (K) (key == NULL_KEY ? null : key);
}

/** 判断“x”和“y”是否相等*/
static boolean eq(Object x, Object y) {
    return x == y || x.equals(y);
}

//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);
}

/** 消除table中“弱引用键”对应的键值对
 * 1、当WeakHashMap中某个“弱引用的key”由于没有再被引用而被GC收回时、被回收的“弱引用key”也会被
被添加到“ReferenceQueue(queue)”中。
 */
private void expungeStaleEntries() {
    Entry<K,V> e;
    while ( (e = (Entry<K,V>) queue.poll()) != null) {
        int h = e.hash;
        int i = indexFor(h, table.length);

        Entry<K,V> prev = table[i];
        Entry<K,V> p = prev;
        while (p != null) {
            Entry<K,V> next = p.next;
            if (p == e) {
                if (prev == e)
                    table[i] = next;
                else
                    prev.next = next;
                e.next = null; // Help GC
                e.value = null; // " "
                size--;
                break;
            }
            prev = p;
            p = next;
        }
    }
}

/** 消除table中“弱引用键”对应的键值对、每次使用WeakHashMap时会先调用此方法*/
private Entry[] getTable() {
    expungeStaleEntries();
    return table;
}

/** 返回当前HashMap中键值对个数*/
public int size() {
    if (size == 0)
        return 0;
    expungeStaleEntries();
    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) {
    Object k = maskNull(key);
    int h = HashMap.hash(k.hashCode());
    Entry[] tab = getTable();
    int index = indexFor(h, tab.length);

```

```

    Entry<K,V> e = tab[index];
    while (e != null) {
        if (e.hash == h && eq(k, e.get()))
            return e.value;
        e = e.next;
    }
    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*/
Entry<K,V> getEntry(Object key) {
    Object k = maskNull(key);
    int h = HashMap.hash(k.hashCode());
    Entry[] tab = getTable();
    int index = indexFor(h, tab.length);
    Entry<K,V> e = tab[index];
    while (e != null && !(e.hash == h && eq(k, e.get())))
        e = e.next;
    return e;
}
/** 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) {
    K k = (K) maskNull(key);
    int h = HashMap.hash(k.hashCode());
    Entry[] tab = getTable();
    int i = indexFor(h, tab.length);

    for (Entry<K,V> e = tab[i]; e != null; e = e.next) {
        if (h == e.hash && eq(k, e.get())) {
            V oldValue = e.value;
            if (value != oldValue)
                e.value = value;
            return oldValue;
        }
    }

    modCount++;
    Entry<K,V> e = tab[i];
    tab[i] = new Entry<K,V>(k, value, queue, h, e);
    if (++size >= threshold)
        resize(tab.length * 2);
    return null;
}
/** rehash当前WeakHashMap、此方法会在WeakHashMap容量达到阈值的时候自动调用、*/
void resize(int newCapacity) {
    Entry[] oldTable = getTable();
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }

    Entry[] newTable = new Entry[newCapacity];
    transfer(oldTable, newTable);
    table = newTable;

    /*
     * If ignoring null elements and processing ref queue caused massive
     * shrinkage, then restore old table. This should be rare, but avoids
     * unbounded expansion of garbage-filled tables.
     */
    if (size >= threshold / 2) {
        threshold = (int)(newCapacity * loadFactor);
    } else {
        expungeStaleEntries();
        transfer(newTable, oldTable);
        table = oldTable;
    }
}

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

```

```

        Object key = e.get();
        if (key == null) {
            e.next = null; // Help GC
            e.value = null; // " "
            size--;
        } else {
            int i = indexOf(e.hash, dest.length);
            e.next = dest[i];
            dest[i] = e;
        }
        e = next;
    }
}

/** 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 (Map.Entry<? extends K, ? extends V> e : m.entrySet())
        put(e.getKey(), e.getValue());
}

/** 删除“键为key”的元素*/
public V remove(Object key) {
    Object k = maskNull(key);
    int h = HashMap.hash(k.hashCode());
    Entry[] tab = getTable();
    int i = indexOf(h, tab.length);
    Entry<K,V> prev = tab[i];
    Entry<K,V> e = prev;

    while (e != null) {
        Entry<K,V> next = e.next;
        if (h == e.hash && eq(k, e.get())) {
            modCount++;
            size--;
            if (prev == e)
                tab[i] = next;
            else
                prev.next = next;
            return e.value;
        }
        prev = e;
        e = next;
    }

    return null;
}

/** Special version of remove needed by Entry set */
Entry<K,V> removeMapping(Object o) {
    if (!(o instanceof Map.Entry))
        return null;
    Entry[] tab = getTable();
    Map.Entry entry = (Map.Entry)o;
    Object k = maskNull(entry.getKey());
    int h = HashMap.hash(k.hashCode());
    int i = indexOf(h, tab.length);
    Entry<K,V> prev = tab[i];
    Entry<K,V> e = prev;

    while (e != null) {

```

```

        Entry<K,V> next = e.next;
        if (h == e.hash && e.equals(entry)) {
            modCount++;
            size--;
            if (prev == e)
                tab[i] = next;
            else
                prev.next = next;
            return e;
        }
        prev = e;
        e = next;
    }

    return null;
}

/** 删除所有键值对*/
public void clear() {
    // clear out ref queue. We don't need to expunge entries
    // since table is getting cleared.
    while (queue.poll() != null)
        ;

    modCount++;
    Entry[] tab = table;
    for (int i = 0; i < tab.length; ++i)
        tab[i] = null;
    size = 0;

    // Allocation of array may have caused GC, which may have caused
    // additional entries to go stale. Removing these entries from the
    // reference queue will make them eligible for reclamation.
    while (queue.poll() != null)
        ;
}

/** 删除所有键值对*/
public void clear() {
    // clear out ref queue. We don't need to expunge entries
    // since table is getting cleared.
    while (queue.poll() != null)
        ;

    modCount++;
    Entry[] tab = table;
    for (int i = 0; i < tab.length; ++i)
        tab[i] = null;
    size = 0;

    // Allocation of array may have caused GC, which may have caused
    // additional entries to go stale. Removing these entries from the
    // reference queue will make them eligible for reclamation.
    while (queue.poll() != null)
        ;
}

/** 判断是否包含value*/
public boolean containsValue(Object value) {
    if (value==null)
        return containsNullValue();

    Entry[] tab = getTable();
    for (int i = tab.length ; i-- > 0 ; )
        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 = getTable();
    for (int i = tab.length ; i-- > 0 ; )
        for (Entry e = tab[i] ; e != null ; e = e.next)
            if (e.value==null)
                return true;
    return false;
}

/**
 * Entry是单向链表。
 * 他继承WeakReference、使得可以使用Entry的key作为弱引用、并且向ReferenceQueue（queue）中注册该

```

引用、以便后期检测WeakHashMap中key的引用类型、进而调整WeakHashMap

* 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(Object o), hashCode()这些函数

```
*/
private static class Entry<K,V> extends WeakReference<K> implements Map.Entry<K,V> {
    private V value;
    private final int hash;
    private Entry<K,V> next;

    /** 创建一个实体Entry、并将Entry的key以弱引用的形式向给定的ReferenceQueue注册*/
    Entry(K key, V value, ReferenceQueue<K> queue, int hash, Entry<K,V> next) {
        //创建引用给定对象的新的弱引用，并向给定队列注册该引用。
        super(key, queue);
        this.value = value;
        this.hash = hash;
        this.next = next;
    }

    public K getKey() {
        return WeakHashMap.<K>unmaskNull(get());
    }

    public V getValue() {
        return value;
    }

    public V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }

    public 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 int hashCode() {
        Object k = getKey();
        Object v = getValue();
        return ((k==null ? 0 : k.hashCode()) ^
            (v==null ? 0 : v.hashCode()));
    }

    public String toString() {
        return getKey() + "=" + getValue();
    }
}
}
```

```
/**
 * 抽象类、用于迭代WeakHashMap、
 * 包含三种视图的迭代“keySet”、“valueCollection”、“Entry<K, V>”三个迭代器
 */
```

```
private abstract class HashIterator<T> implements Iterator<T> {
    int index;
    Entry<K,V> entry = null;
    Entry<K,V> lastReturned = null;
    int expectedModCount = modCount;

    /** 下一个键（强引用、不会消失*/
    Object nextKey = null;

    /** 当前键（强引用、不会消失*/
    Object currentKey = null;

    HashIterator() {
        index = (size() != 0 ? table.length : 0);
    }
}
```

//查看是否有下一个


```

public boolean hasNext() {
    Entry[] t = table;

    while (nextKey == null) {
        Entry<K,V> e = entry;
        int i = index;
        while (e == null && i > 0)
            e = t[--i];
        entry = e;
        index = i;
        if (e == null) {
            currentKey = null;
            return false;
        }
        nextKey = e.get(); // hold on to key in strong ref
        if (nextKey == null)
            entry = entry.next;
    }
    return true;
}

//获取下一个元素
protected Entry<K,V> nextEntry() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
    if (nextKey == null && !hasNext())
        throw new NoSuchElementException();

    lastReturned = entry;
    entry = entry.next;
    currentKey = nextKey;
    nextKey = null;
    return lastReturned;
}

public void remove() {
    if (lastReturned == null)
        throw new IllegalStateException();
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();

    WeakHashMap.this.remove(currentKey);
    expectedModCount = modCount;
    lastReturned = null;
    currentKey = null;
}
}

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

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

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

// Views

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

public Set<K> keySet() {
    Set<K> ks = keySet;
    return (ks != null ? ks : (keySet = new KeySet()));
}

private class KeySet extends AbstractSet<K> {
    public Iterator<K> iterator() {
        return new KeyIterator();
    }
}

```

```

    public int size() {
        return WeakHashMap.this.size();
    }

    public boolean contains(Object o) {
        return containsKey(o);
    }

    public boolean remove(Object o) {
        if (containsKey(o)) {
            WeakHashMap.this.remove(o);
            return true;
        }
        else
            return false;
    }

    public void clear() {
        WeakHashMap.this.clear();
    }
}

public Collection<V> values() {
    Collection<V> vs = values;
    return (vs != null ? vs : (values = new Values()));
}

private class Values extends AbstractCollection<V> {
    public Iterator<V> iterator() {
        return new ValueIterator();
    }

    public int size() {
        return WeakHashMap.this.size();
    }

    public boolean contains(Object o) {
        return containsValue(o);
    }

    public void clear() {
        WeakHashMap.this.clear();
    }
}

public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> es = entrySet;
    return es != null ? es : (entrySet = new EntrySet());
}

private class EntrySet extends AbstractSet<Map.Entry<K,V>> {
    public Iterator<Map.Entry<K,V>> iterator() {
        return new EntryIterator();
    }

    public boolean contains(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry e = (Map.Entry)o;
        Object k = e.getKey();
        Entry candidate = getEntry(e.getKey());
        return candidate != null && candidate.equals(e);
    }

    public boolean remove(Object o) {
        return removeMapping(o) != null;
    }

    public int size() {
        return WeakHashMap.this.size();
    }

    public void clear() {
        WeakHashMap.this.clear();
    }
}

//深度克隆、提供toArray()、toArray(T[] a)方法
private List<Map.Entry<K,V>> deepCopy() {
    List<Map.Entry<K,V>> list = new ArrayList<Map.Entry<K,V>>(size());

```

```

        for (Map.Entry<K,V> e : this)
            list.add(new AbstractMap.SimpleEntry<K,V>(e));
        return list;
    }

    public Object[] toArray() {
        return deepCopy().toArray();
    }

    public <T> T[] toArray(T[] a) {
        return deepCopy().toArray(a);
    }
}
}

```

总结：

1、数据结构：WeakHashMap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、WeakHashMap中存储的Entry实现了Map.Entry<K, V>、WeakReference、并且借助WeakReference的构造方法将WeakReference与ReferenceQueue结合起来、使用Entry的key作为弱引用键注册到ReferenceQueue中。Entry源码：

```

/**
 * Entry是单向链表。
 * 他继承WeakReference、使得可以使用Entry的key作为弱引用、并且向ReferenceQueue（queue）中注册该引用、以便后期检测WeakHashMap中key的引用类型、进而调整WeakHashMap
 * 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(Object o), hashCode()这些函数
 */
private static class Entry<K,V> extends WeakReference<K> implements Map.Entry<K,V> {
    private V value;
    private final int hash;
    private Entry<K,V> next;

    /** 创建一个实体Entry、并将Entry的key以弱引用的形式向给定的ReferenceQueue注册*/
    Entry(K key, V value, ReferenceQueue<K> queue, int hash, Entry<K,V> next) {
        //调用WeakReference构造方法创建引用给定对象的新的弱引用，并向给定队列注册该引用。
        super(key, queue);
        this.value = value;
        this.hash = hash;
        this.next = next;
    }

    public K getKey() {
        return WeakHashMap.<K>unmaskNull(get());
    }

    public V getValue() {
        return value;
    }

    public V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }

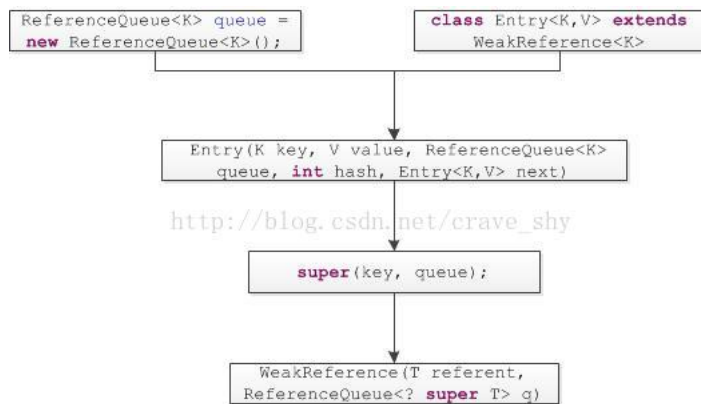
    public boolean equals(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry e = (Map.Entry)o;
        Object k1 = getKey();
        Object k2 = e.getKey();
        if (k1 == k2 || (k1 != null && k1.equals(k2))) {
            Object v1 = getValue();
            Object v2 = e.getValue();
            if (v1 == v2 || (v1 != null && v1.equals(v2)))
                return true;
        }
        return false;
    }

    public int hashCode() {
        Object k = getKey();
        Object v = getValue();
        return ((k==null ? 0 : k.hashCode()) ^
            (v==null ? 0 : v.hashCode()));
    }

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

```

关于key作为弱引用的实现流程图：

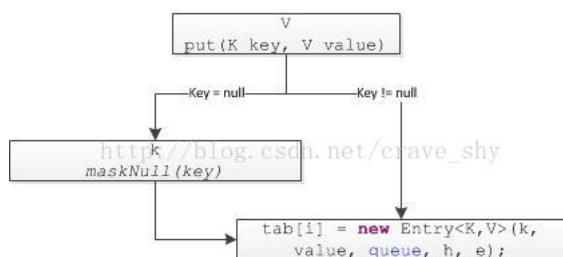


到此完成WeakHashMap的键的弱引用的构造。

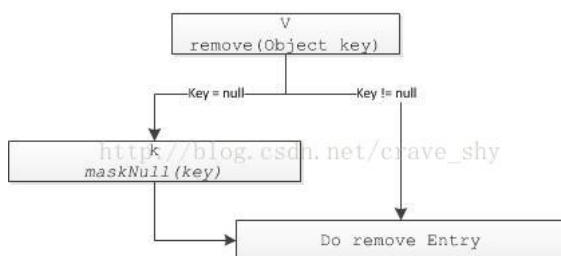
2、关于WeakHashMap的使用：

分成三部分来说明：添加、删除、其他、主要是WeakHashMap允许键为null的值、其内部对键为null进行了特殊处理、其他的则是每次使用WeakHashMap的时候都要将WeakHashMap中弱引用的键值对删除、即同步table和ReferenceQueue中存放的引用指向的键值对。

a) 添加：



b) 删除：



c) 其他：通过关键同步源码来说明

```

/** 消除table中“弱引用键”对应的键值对
 * 1、当WeakHashMap中某个“弱引用的key”由于没有再被引用而被GC收回时、被回收的“弱引用key”也会被会
被添加到"ReferenceQueue(queue)"中。
 */
private void expungeStaleEntries() {
    Entry<K,V> e;
    while ( (e = (Entry<K,V>) queue.poll()) != null) {
        int h = e.hash;
        int i = indexFor(h, table.length);

        Entry<K,V> prev = table[i];
        Entry<K,V> p = prev;
        while (p != null) {
            Entry<K,V> next = p.next;
            if (p == e) {
                if (prev == e)
                    table[i] = next;
                else
                    prev.next = next;
                e.next = null; // Help GC
                e.value = null; // " "
                size--;
                break;
            }
            prev = p;
            p = next;
        }
    }
}

/** 消除table中“弱引用键”对应的键值对、每次使用WeakHashMap时会先调用此方法*/
private Entry[] getTable() {
    expungeStaleEntries();
    return table;
}

```

当我们每次要使用WeakHashMap的一些方法时、都会事先调用此方法来处理WeakHashMap、也就是删除弱引用键指向的键值对。

3、与容量有关的内容WeakHashMap

```

/** rehash当前WeakHashMap、此方法会在WeakHashMap容量达到阈值的时候自动调用、*/
void resize(int newCapacity) {
    Entry[] oldTable = getTable();
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }

    Entry[] newTable = new Entry[newCapacity];
    transfer(oldTable, newTable);
    table = newTable;

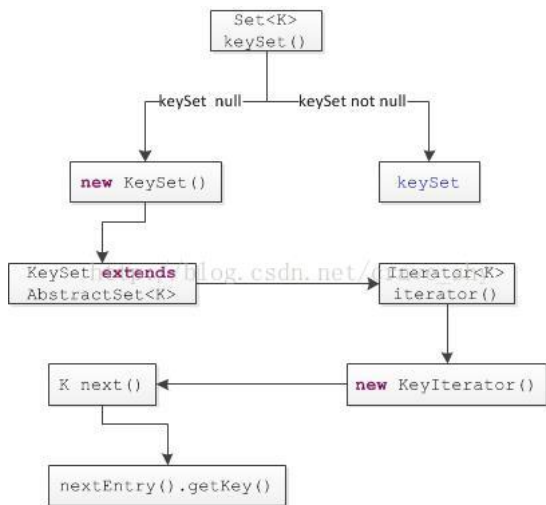
    /*
     * If ignoring null elements and processing ref queue caused massive
     * shrinkage, then restore old table. This should be rare, but avoids
     * unbounded expansion of garbage-filled tables.
     */
    if (size >= threshold / 2) {
        threshold = (int)(newCapacity * loadFactor);
    } else {
        expungeStaleEntries();
        transfer(newTable, oldTable);
        table = oldTable;
    }
}

/** 将原来table中所有元素转移到新的table中*/
private void transfer(Entry[] src, Entry[] dest) {
    for (int j = 0; j < src.length; ++j) {
        Entry<K,V> e = src[j];
        src[j] = null;
        while (e != null) {
            Entry<K,V> next = e.next;
            Object key = e.get();
            if (key == null) {
                e.next = null; // Help GC
                e.value = null; // " "
                size--;
            } else {
                int i = indexFor(e.hash, dest.length);
                e.next = dest[i];
                dest[i] = e;
            }
            e = next;
        }
    }
}
}

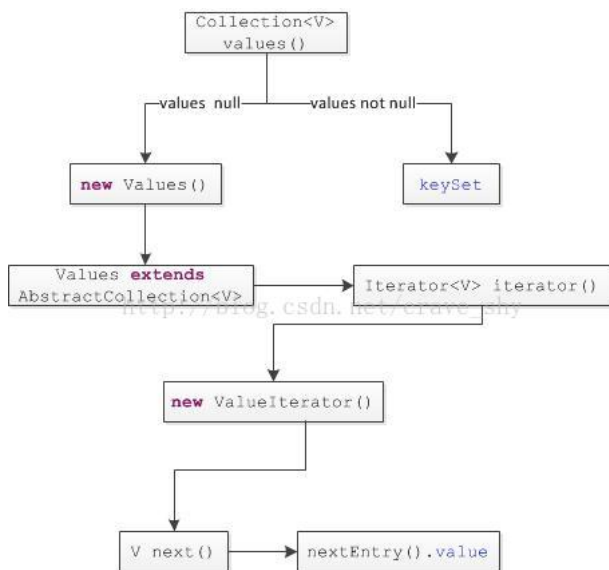
```

4、WeakHashMap迭代有关：从源码中可看出、WeakHashMap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、简化编程。WeakHashMap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、获取三个views的流程图：

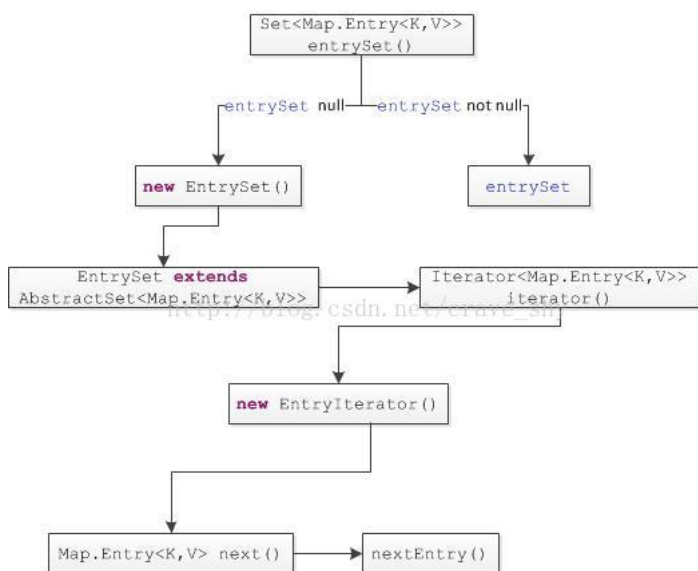
a) 获取key组成的set：Iterator内部要实现next()方法、hasNext()、remove()方法使用HashIterator提供的实现。



b) 获取values组成的Collection : Iterator内部要实现next()方法、hasNext()、remove()方法使用HashIterator提供的实现。



c) 获取entrySet组成的Set : Iterator内部要实现next()方法、hasNext()、remove()方法使用HashIterator提供的实现。



五：WeakHashMap 示例

1、遍历方式：

a) 使用由key组成的Set：

```
keySet = weakHashMap.keySet();  
Iterator<String> it = keySet.iterator();
```

b) 使用由value组成的Collection：

```
keySet = weakHashMap.keySet();  
Iterator<String> it = keySet.iterator();
```

c) 使用由entry组成的Set：

```
entrySet = weakHashMap.entrySet();  
Iterator<Entry<String, Integer>> it = entrySet.iterator();
```

2、迭代示例：

```

package com.chy.collection.example;

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

public class EragodicWeakHashMap {

    private static WeakHashMap<String, Integer> weakHashMap = new WeakHashMap<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++) {
            weakHashMap.put("" + i, i);
        }
    }

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

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

    /**
     * iterate weakHashMap by EntrySet
     */
    private static void testEntrySet(){
        entrySet = weakHashMap.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.WeakHashMap;

/**
 * WeakHashMap中所有的键如果不指向任何存在对象、则全都是弱引用对象、这样的对象会在下次调用WeakHashM
ap时被GC回收
 * 当一个值作为WeakHashMap的键的同时、也指向一个具体的对象、则这样的键为强引用、就暂时不会被回收。
 * 如果使用基本类型作为键、则下次调用WeakHashMap时、GC不会回收其所指定的键值对。
 */
@SuppressWarnings("unused")
public class WeakHashMapTest {
    /**
     * 测试使用强引用作为键的WeakHashMap
     */
    private static void testStrongReference(){
        Key key1 = new Key("1");
        Key key2 = new Key("2");
        Key key3 = new Key("3");

        WeakHashMap<Key, Value> whm = new WeakHashMap<Key, Value>();
        whm.put(key1, new Value("1"));
        whm.put(key2, new Value("2"));
        whm.put(key3, new Value("3"));
        System.gc();
        System.out.println("strong reference of WeakHashMap key : " + whm);
    }

    /**
     * 测试使用弱引用作为键的WeakHashMap
     */
    private static void testWeakReference(){
        WeakHashMap<Key, Value> whm = new WeakHashMap<Key, Value>();
        whm.put(new Key("1"), new Value("1"));
        whm.put(new Key("2"), new Value("2"));
        whm.put(new Key("3"), new Value("3"));
        System.gc();
        System.out.println("weak reference of WeakHashMap key : " + whm);
    }

    /**
     * 测试同时使用弱引用、强引用作为键的WeakHashMap
     * @throws InterruptedException
     */
    private static void testCompoundReference(){
        int size = 100; // 或者从命令行获得size的大小
        Key[] keys = new Key[size]; // 存放键对象的强引用
        WeakHashMap<Key, Value> whm = new WeakHashMap<Key, Value>();
        for (int i = 0; i < size; i++) {
            Key k = new Key(Integer.toString(i));
            Value v = new Value(Integer.toString(i));
            if (i % 3 == 0)
                keys[i] = k; // 使Key对象持有强引用
            whm.put(k, v); // 使Key对象持有弱引用
        }
        // 催促垃圾回收器工作
        System.gc(); // 把CPU让给垃圾回收器线程
    }

    /**
     * 测试将作为键的引用从强引用变为弱引用时的WeakHashMap
     */
    private static void testStrongToWeakReference(){
        Key key1 = new Key("1");
        Key key2 = new Key("2");
        Key key3 = new Key("3");

        WeakHashMap<Key, Value> whm = new WeakHashMap<Key, Value>();
        whm.put(key1, new Value("1"));
        whm.put(key2, new Value("2"));
        whm.put(key3, new Value("3"));

        key1 = null;

        // 放置一个key为null的键值对
        whm.put(null, new Value("1"));
        System.gc();
        System.out.println("strong to weak reference of WeakHashMap key : " + whm);
    }
}

```

```

/**
 * 测试使用基本类型作为键的WeakHashMap
 */
private static void testBasicReference(){
    WeakHashMap<Integer, Value> whm = new WeakHashMap<Integer, Value>();
    whm.put(1, new Value("1"));
    whm.put(2, new Value("2"));
    whm.put(3, new Value("3"));
    System.gc();
    System.out.println("use basic as reference of WeakHashMap key : " + whm);
}

public static void main(String[] args) {
    // testStrongReference();
    // testWeakReference();
    // testCompoundReference();
    // testBasicReference();
    // testStrongToWeakReference();
}

class Key {
    String id;

    public Key(String id) {
        this.id = id;
    }
    public String toString() {
        return id;
    }
    public int hashCode() {
        return id.hashCode();
    }
    public boolean equals(Object r) {
        return (r instanceof Key) && id.equals(((Key) r).id);
    }
    public void finalize() {
        System.out.println("Finalizing Key " + id);
    }
}

class Value {
    String id;
    public Value(String id) {
        this.id = id;
    }
    public String toString() {
        return id;
    }
    public void finalize() {
        System.out.println("Finalizing Value " + id);
    }
}

```


更多内容：[java_集合体系之总体目录——00](http://blog.csdn.net/crave_shy/article/details/174167)
http://blog.csdn.net/crave_shy/article/details/174167

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：WeakHashMap (<http://so.csdn.net/so/search/s.do?q=WeakHashMap&t=blog>) /
 Map框架图 (<http://so.csdn.net/so/search/s.do?q=Map框架图&t=blog>) /
 HashMap (<http://so.csdn.net/so/search/s.do?q=HashMap&t=blog>) /
 弱引用 (<http://so.csdn.net/so/search/s.do?q=弱引用&t=blog>) /
 源码 (<http://so.csdn.net/so/search/s.do?q=源码&t=blog>) /

0条评论

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

[发表评论](#)

暂无评论

相关文章推荐

Java微信二次开发之14-自定义菜单及菜单响应事件的推送 (/lanbuff/article/details/52450617)

2013年8月5日，伴随着微信5.0 iPhone版的发布，公众平台也进行了重要的更新，主要包括：1) 运营主体为组织，可选择成为服务号或者订阅号；2) 服务号可以申请自定义菜单；...



LanBuff 2016-09-06 15:45 3230

MySQL关于根据日期查询数据的sql语句 (/zhaitonghui/article/details/53080992)

查询往前7天的数据: select * from 数据表 where DATE_SUB(CURDATE(), INTERVAL 7 DAY) 查询往前30天的数据: select * from ...



zhaitonghui 2016-11-08 12:27 1505

Excel导入导出 (/crave_shy/article/details/21931685)

摘要：简单的基于Apache的POI的Excel的导入、导出。仅作基础操作、功能需要的可以自己根据自己的需求添加自己的实现。



chenghuaying 2014-03-24 10:08 2197

整理的oracle实现的基础sql语句查询 (/crave_shy/article/details/12068639)

/* --17、按平均成绩从高到低显示所有学生的所有课程的成绩以及平均成绩 --difficult point : --the use of "case field when condition ...



chenghuaying 2013-09-26 21:26 1424

求MySQL先按大于等于当前时间升序排序，再按小于当前时间降序排序 (/bulongwind/article/details/70156493)

现在有一个新闻表，表里面有一个时间字段。我现在想要一条select查出来的数据，在排序上是先按大于等于当前时间升序排序，再按小于当前时间降序排序。举个例子就是，比如今天是4月7...



bulongwind 2017-04-13 11:32 1387

C++闭包，一样很简单 (/goldenhawking/article/details/70589476)

闭包是指可以包含自由（未绑定到特定对象）变量的代码块；这些变量不是在这个代码块内或者任何全局上下文中定义的，而是在定义代码块的环境中定义（局部变量）。C++11以后，借助functional和lamb...



goldenhawking 2017-04-24 12:42 734

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

c++工程与java的互相调用 (/wcy6340/article/details/10445645)

内容详见我上传的资源文档，在此不再赘述！



wcy6340 2013-08-28 14:35 1043

【数据结构】链表的原理及java实现 (/jianyuerensheng/article/details/51200274)

一：单向链表基本介绍链表是一种数据结构，和数组同级。比如，Java中我们使用的ArrayList，其实现原理是数组。而LinkedList的实现原理就是链表了。链表在进行循环遍历时效率不高，但是插入和...



jianyuerensheng 2016-04-20 13:57 9359

基于51单片机的简易数字频率计 (/baidu_33836580/article/details/50578632)

要求使用定时/计数器1作定时用，定时1s；定时/计数器0作计数器用，被计数的外部脉冲从P3.4（T0）接入。单片机将在1s内对脉冲计数并送四位数码管显示，最大计数显示值为0FFFFH。求程序。悬赏分：...



baidu_33836580 2016-01-25 11:40 1641

Java_io体系之PipedWriter、PipedReader简介、走进源码及示例——14 (<http://810364804.iteye.com/blog/1992802>)

Java_io体系之PipedWriter、PipedReader简介、走进源码及示例——14 ——管道字符输出流、必须建立在管道输入流之上、所以先介绍管道字符输出流。可以先看示例或者总结、总结写的有点Q、不喜可无视、有误的地方指出则不胜感激。一：PipedWriter 1、类功能简介：管道字符输出流、用于将当前线程的指定字符写入到与此线程对应的管道字符输入流



810364804 2013-12-08 18:50 57

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

Java_io体系之FilterWriter、FilterReader简介、走进源码及示例——15 (<http://810364804.iteye.com/blog/1992801>)

Java_io体系之FilterWriter、FilterReader简介、走进源码及示例——15 一：FilterWriter 1、类功能简介：字符过滤输出流、与FilterOutputStream功能一样、只是简单重写了父类的方法、目的是为所有装饰类提供标准和基本的方法、要求子类必须实现核心方法、和拥有自己的特色。这里FilterWriter没有子类、可能其意义只是提供一个接口、留着以后的扩展。。。本身是一个抽象类、如同Wr



810364804 2013-12-08 20:50 62

java_集合体系之HashMap详解、源码及示例——09 (/crave_shy/article/details/17552679)

摘要：本文通过HashMap的结构图分析HashMap所具有的特性、通过源码深入了解HashMap实现原理、使用

方法、通过实例加深对HashMap的应用的理解。篇幅较长、慎入！



chenghuaying 2013-12-25 14:54 2542

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 一：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线程池 (/nanmuling/article/details/37881089)

Java线程池 线程池编程 java.util.concurrent多线程框架---线程池编程（一）一般的服务器都需要线程池，比如Web、FTP等服务器，不过它们一般都自己实现了线程池，比如以...



nanmuling 2014-07-16 16:44 2850

Java_io体系之PrintWriter简介、走进源码及示例——19 (<http://810364804.iteye.com/blog/1992795>)

Java_io体系之PrintWriter简介、走进源码及示例——19 PrintWriter 1、类功能简介：打印字符流、用于将各种Java数据一字符串的形式打印到底层字符输出流中、本身不会产生任何IOException、但是可以通过他的一个方法来查看是否抛出异常、可以指定autoFlush、若为true则当调用newLi



810364804 2013-12-10 14:33 121

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 一：

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

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