ArrayList、LinkedList、 Vector、Map 用法比较

标签： vector hashmap iterator object equals 数据结构

2012-06-15 18:17 13196人阅读 评论(0) 收藏 举报

分类： Java/JSP（46）

ArrayList和Vector是采用数组方式存储数据，此数组元素总数大于实际存储的数据个数以便增加和插入元素，二者都允许直接序号索引元素，但是插入数据要移动数组元素等内存操作，所以它们索引数据快、插入数据慢。
ArrayList数组存储方式：

[java] view plain copy print ?

```java
private transient Object[] elementData;
public ArrayList(int initialCapacity) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: " + initialCapacity);
    this.elementData = new Object[initialCapacity];
}

// 空构造函数，默认容量大小为10
public ArrayList() {
    this(10);
}

public ArrayList(Collection<? extends E> c) {
    elementData = c.toArray();
    size = elementData.length;
    // c.toArray might (incorrectly) not return Object[] (see 6260652)
    if (elementData.getClass() != Object[].class)
        elementData = Arrays.copyOf(elementData, size, Object[].class);
}
```

Vector由于使用了synchronized同步方法（如add、insert、remove、set、equals、hashcode等操作），因此是线程安全，性能上比ArrayList要差。
Vector数组存储方式：

[java] view plain copy print ?

```java
protected Object[] elementData;
protected int capacityIncrement;
public Vector(int initialCapacity, int capacityIncrement) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
    this.elementData = new Object[initialCapacity];
    this.capacityIncrement = capacityIncrement;
}

// 设置容量大小为initialCapacity，默认增长个数为0
public Vector(int initialCapacity) {
    this(initialCapacity, 0);
}

// 空构造函数，默认容量大小为10
public Vector() {
    this(10);
}
```

LinkedList使用双向链表实现存储，按序号索引数据需要进行向前或向后遍历，但是插入数据时只需要记录本项的前后项即可，所以插入数度较快！

个人博客

mimvp.com

新博客：http://blog.mimvp.com

本CSDN博客，已全部移到了我的米扑博客； 开源出脚本转换工具，供大家参考。

博客专栏


Clojure 学习总结
文章：18篇
阅读：139826


Python 学习入门
文章：52篇
阅读：483076


设计模式
文章：3篇
阅读：42882


Android开发的点点滴滴
文章：32篇
阅读：1900504

博客公告

本博客内容，由本人精心整理

欢迎交流，欢迎转载，大家转载注明出处，禁止用于商业目的。

文章搜索

文章分类

文章存档

LinkedList双向链表，是指可以从first依次遍历至last（从头到尾），也可以从last遍历至first（从尾到头），但首尾没有构成环，不同于双向循环链表（注意区分）：

```
[java] view plain  copy  print  ?
01.  transient Node<E> first;
02.  transient Node<E> last;
03.
04.  public LinkedList() {
05.  }
06.
07.  private void linkFirst(E e) {
08.         final Node<E> f = first;
09.         final Node<E> newNode = new Node<>(null, e, f);       // 插入新节点，同时连接首、尾节点
10.         first = newNode;
11.         if (f == null)            // 起始节点为空（null），表示插入后有且只有一个节点，因此 first = last = newNode
12.             last = newNode;
13.         else
14.             f.prev = newNode;
15.         size++;
16.         modCount++;
17.     }
18.
19.
20.     void linkLast(E e) {
21.         final Node<E> l = last;
22.         final Node<E> newNode = new Node<>(l, e, null);       // 插入新节点，同时连接首、尾节点
23.         last = newNode;
24.         if (l == null)            // 末尾节点为空（null），表示插入后有且只有一个节点，因此 first = last = newNode
25.             first = newNode;
26.         else
27.             l.next = newNode;
28.         size++;
29.         modCount++;
30.     }
```

其中，Node类结构如下：

```
[java] view plain  copy  print  ?
01.  private static class Node<E> {
02.         E item;
03.         Node<E> next;
04.         Node<E> prev;
05.
06.         Node(Node<E> prev, E element, Node<E> next) {
07.             this.item = element; // 节点数据
08.             this.next = next;         // 连接下一节点
09.             this.prev = prev;         // 连接上一节点
10.         }
11.     }
```

线性表、链表、哈希表是常用的**数据结构**，在进行**Java**开发时，JDK已经为我们提供了一系列相应的类来实现基本的数据结构，这些类均在java.util包中。

Collection

- |--------List
  - |----------LinkedList
  - |----------ArrayList
  - |----------Vector
    - |-----Stack
- |--------Set
  - |----------HashSet.
    - |-----LinkedHashSet
  - |----------SortedSet
    - |-----TreeSet

Collection.

- 实现该接口及其子接口的所有类都可应用clone()方法，并是序列化类.

.....List.

.....● 可随机访问包含的元素
.....● 元素是有序的
.....● 可在任意位置增、删元素
.....● 不管访问多少次，元素位置不变
.....● 允许重复元素
.....● 用Iterator实现单向遍历，也可用ListIterator实现双向遍历

2016年03月 (1)
2016年02月 (1)
2016年01月 (1)
展开

阅读排行

Android APK反编译详解 (818221)
Git 常用命令详解（二） (176460)
SVN常用命令 (175638)
Android 获取屏幕尺寸与 (171602)
Android Service 服务（ (126128)
Ubuntu搭建Eclipse+JDK (120911)
Linux 抓取网页实例（sh (118815)
Windows搭建Eclipse+JD (98278)
各种基本算法实现小结（ (91419)
Android 创建与解析XML (90452)

.....ArrayList
.....● 用数组作为根本的数据结构来实现List
.....● 元素顺序存储
.....● 新增元素改变List大小时，内部会新建一个数组，在将添加元素前将所有数据拷贝到新数组中
.....● 随机访问很快，删除非头尾元素慢，新增元素慢而且费资源
.....● 较适用于无频繁增删的情况
.....● 比数组效率低，如果不是需要可变数组，可考虑使用数组
.....● 非线程安全

.....Vector.
.....● 另一种ArrayList，具备ArrayList的特性
.....● 所有方法都是线程安全的（双刃剑，和ArrayList的主要区别）
.....● 比ArrayList效率低

.....Stack
.....● LIFO的数据结构

.....LinkedList.
.....● 链接对象数据结构（类似链表）
.....● 随机访问很慢，增删操作很快，不耗费多余资源
.....● 非线程安全

.....Set.
.....● 不允许重复元素，可以有一个空元素
.....● 不可随机访问包含的元素
.....● 只能用Iterator实现单向遍历

.....HashSet
.....● 用HashMap作为根本数据结构来实现Set
.....● 元素是无序的
.....● 迭代访问元素的顺序和加入的顺序不同
.....● 多次迭代访问，元素的顺序可能不同
.....● 非线程安全

.....LinkedHashSet
.....● 基于HashMap和链表的Set实现
.....● 迭代访问元素的顺序和加入的顺序相同
.....● 多次迭代访问，元素的顺序不变
.....● 因此可说这是一种有序的数据结构
.....● 性能比HashSet差
.....● 非线程安全

.....SortedSet
.....● 加入SortedSet的所有元素必须实现Comparable接口
.....● 元素是有序的

.....TreeSet.
.....● 基于TreeMap实现的SortedSet
.....● 排序后按升序排列元素
.....● 非线程安全

Collection接口
Collection是最基本的集合接口，一个Collection代表一组Object，即Collection的元素（Elements）。一些Collection允许相同的元素而另一些不行，一些能排序而另一些不行。Java SDK不提供直接继承自Collection的类，Java SDK提供的类都是继承自Collection的"子接口"如List和Set。

所有实现Collection接口的类都必须提供两个标准的构造函数：
1）无参数的构造函数，用于创建一个空的Collection
2）有一个Collection参数的构造函数，用于创建一个新的Collection，这个新的Collection与传入的Collection有相同的元素。
后一个构造函数允许用户复制一个Collection。

如何遍历Collection中的每一个元素？
不论Collection的实际类型如何，它都支持一个iterator()的方法，该方法返回一个迭代子，使用该迭代子即可逐一访问Collection中每一个元素。典型的用法如下：

```
Iterator it = collection.iterator(); // 获得一个迭代子
while(it.hasNext()) {
    Object obj = it.next(); // 得到下一个元素
}
```

由Collection接口派生的两个接口是List和Set。

List接口
List是有序的Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在List中的位置，类似于数组下标）来访问List中的元素，这类似于Java的数组。和下面要提到的Set不同，List允许有相同的元素。

除了具有Collection接口必备的iterator()方法外，List还提供一个listIterator()方法，返回一个ListIterator接口，和标准的Iterator接口相比，ListIterator多了一些add()之类的方法，允许添加，删除，设定元素，还能向前或向后遍历。
实现List接口的常用类有LinkedList，ArrayList，Vector和Stack。

LinkedList类

LinkedList实现了List接口，允许null元素。此外在LinkedList的首部或尾部提供额外的get、remove、insert方法。这些操作使LinkedList可被用作堆栈（stack），队列（queue）或双向队列（deque）。
注意：LinkedList没有同步方法。如果多个线程同时访问一个List，则必须自己实现访问同步。一种解决方法是在创建List时构造一个同步的List：

```
List list = Collections.synchronizedList(new LinkedList(...));
```

ArrayList类
ArrayList实现了可变大小的数组，它允许所有元素，包括null，没有同步。
size、isEmpty、get、set方法运行时间为常数。但是add方法开销为分摊的常数，添加n个元素需要O(n)的时间，其他的方法运行时间为线性。

每个ArrayList实例都有一个容量（Capacity），即用于存储元素的数组的大小。这个容量可随着不断添加新元素而自动增加，但是增长算法并没有定义。当需要插入大量元素时，在插入前可以调用ensureCapacity方法来增加ArrayList的容量以提高插入效率。
ArrayList扩展容量：

```
[java] view plain  copy  print  ?
01.  public void ensureCapacity(int minCapacity) {       // minCapacity： 期望设置的最小容量大小
02.         if (minCapacity > 0)
03.             ensureCapacityInternal(minCapacity);     // 内部方法实现（private）
04.     }
05.
06.  private void ensureCapacityInternal(int minCapacity) {
07.         modCount++;
08.         // overflow-conscious code
09.         if (minCapacity - elementData.length > 0)     // 如果期望的最小容量minCapacity大于当前元素个数，则设置
10.             grow(minCapacity);
11.     }
12.
13.  private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
14.     private void grow(int minCapacity) {
15.         // overflow-conscious code
16.         int oldCapacity = elementData.length;
17.         int newCapacity = oldCapacity + (oldCapacity >> 1);         // 增长当前元素个数的1/2，即增长后为原先元素总量的3/2
18.
19.         if (newCapacity - minCapacity < 0)         // 设置的minCapacity大于自动增长的容量，则按minCapacity最大设置（即取最大的容量）
20.             newCapacity = minCapacity;
21.         if (newCapacity - MAX_ARRAY_SIZE > 0) // 超过了VMs最大的内存分配空间（注：Integer.MAX_VALUE - 8;其中减去8字节是因为Integer附加信息额外占用了8个字节）
22.             newCapacity = hugeCapacity(minCapacity);
23.         // minCapacity is usually close to size, so this is a win:
24.         elementData = Arrays.copyOf(elementData, newCapacity);
25.     }
26.
27.     private static int hugeCapacity(int minCapacity) {
28.         if (minCapacity < 0) // overflow
29.             throw new OutOfMemoryError();
30.         return (minCapacity > MAX_ARRAY_SIZE) ?
31.             Integer.MAX_VALUE :
32.             MAX_ARRAY_SIZE;
33.     }
```

和LinkedList一样，ArrayList也是非同步的（unsynchronized）。

Vector类
Vector非常类似ArrayList，但是Vector是同步的。
由Vector创建的Iterator，虽然和ArrayList创建的Iterator是同一接口，但是，因为Vector是同步的，当一个Iterator被创建而且正在被使用，另一个线程改变了Vector的状态（例如，添加或删除了一些元素），这时调用Iterator的方法时将抛出ConcurrentModificationException，因此必须捕获该异常。

Stack 类
Stack继承自Vector，也是同步的，实现一个后进先出的堆栈。Stack提供5个额外的方法使得Vector得以被当作堆栈使用。基本的push和pop方法，还有peek方法得到栈顶的元素，empty方法测试堆栈是否为空，search方法检测一个元素在堆栈中的位置。
Stack刚创建后是空栈。

Set接口
Set是一种不包含重复的元素的Collection，即任意的两个元素e1和e2都有e1.equals(e2)=false，Set最多有一个null元素。
很明显，Set的构造函数有一个约束条件，传入的Collection参数不能包含重复的元素。Set 没有同步方法。
注意：必须小心操作可变对象（Mutable Object）。如果一个Set中的可变元素改变了自身状态导致Object.equals(Object)=true将导致一些问题。

Map
|------Hashtable
|------Properties

|------HashMap

|------LinkedHashMap

|------WeakHashMap

|------SortedMap

|------TreeMap

Map

● 键值对，键和值一一对应

● 不允许重复的键.

.....Hashtable.

.....● 用作键的对象必须实现了hashcode()、equals()方法，也就是说只有Object及其子类可用作键

.....● 键、值都不能是空对象

.....● 多次访问，映射元素的顺序相同

.....● 线程安全的

.....Properties

.....● 键和值都是字符串

.....HashMap

.....● 键和值都可以是空对象

.....● 不保证映射的顺序

.....● 多次访问，映射元素的顺序可能不同

.....● 非线程安全

.....LinkedHashMap

.....● 多次访问，映射元素的顺序是相同的

.....● 性能比HashMap差

.....WeakHashMap..

.....● 当某个键不再正常使用时，垃圾收集器会移除它，即便有映射关系存在

.....● 非线程安全

.....SortedMap.

.....● 键按升序排列

.....● 所有键都必须实现.Comparable.接口.

.....TreeMap.

.....● 基于红黑树的SortedMap实现

.....● 非线程安全

Map接口

Map没有继承Collection接口，Map提供key到value的映射。一个Map中不能包含相同的key，每个key只能映射一个value。

Map接口提供3种集合的视图，Map的内容可以被当作一组key集合，一组value集合，或者一组key-value映射。

Map接口定义：

```
[java] view plain  copy  print  ?
01. public interface Map<K,V> {
02.     int size();
03.     boolean isEmpty();
04. 
05.     boolean containsKey(Object key);
06.     boolean containsValue(Object value);
07. 
08.     V get(Object key);
09.     V put(K key, V value);
10.     V remove(Object key);
11. 
12.     void putAll(Map<? extends K, ? extends V> m);
13.     void clear();
14. 
15.     Set<K> keySet();                        // key 集合
16.     Collection<V> values();                    // value 集合
17.     Set<Map.Entry<K, V>> entrySet();        // key-value 集合
18. 
19.     interface Entry<K,V> {
20.         K getKey();
21.         V getValue();
22.         V setValue(V value);
```

```
23.
24.         boolean equals(Object o);
25.         int hashCode();
26.     }
27.
28.     boolean equals(Object o);
29.     int hashCode();
30. }
```

Hashtable类

Hashtable继承于Dictionary字典，实现Map接口，完成一个key-value映射的哈希表。任何非空（non-null）的对象都可作为key或者value。

添加数据使用put(key, value)，取出数据使用get(key)，这两个基本操作的时间开销为常数。

Hashtable通过initial capacity和load factor两个参数调整性能。通常缺省的load factor 0.75较好地实现了时间和空间的均衡。增大load factor可以节省空间但相应的查找时间将增大，这会影响像get和put这样的操作。

Hashtable的构造实现：

```
[java] view plain  copy  print  ?
01. private transient Entry[] table;    // 数组实现
02. private float loadFactor;
03.
04. public Hashtable(int initialCapacity, float loadFactor) {
05.        if (initialCapacity < 0)
06.            throw new IllegalArgumentException("Illegal Capacity: " + initialCapacity);
07.        if (loadFactor <= 0 || Float.isNaN(loadFactor))
08.            throw new IllegalArgumentException("Illegal Load: " + loadFactor);
09.
10.        if (initialCapacity==0)       // 初始容量为0时，默认设置为1
11.            initialCapacity = 1;
12.        this.loadFactor = loadFactor;
13.        table = new Entry[initialCapacity];
14.        threshold = (int)(initialCapacity * loadFactor); // 实际阈值
15.    }
16.
17. public Hashtable(int initialCapacity) {
18.        this(initialCapacity, 0.75f);    // 默认 loadFactor = 0.75f
19.    }
20.
21. public Hashtable() {
22.        this(11, 0.75f);     // 默认 initialCapacity = 11 ， loadFactor = 0.75f
23.    }
24.
25. public Hashtable(Map<? extends K, ? extends V> t) {
26.        this(Math.max(2*t.size(), 11), 0.75f);        // 申请两倍Map大小，与默认11比较，取其
    大
27.        putAll(t);
28.    }
```

使用Hashtable的简单示例如下，将1，2，3放到Hashtable中，他们的key分别是”one”，”two”，”three”：

```
Hashtable numbers = new Hashtable();
numbers.put(“one”, new Integer(1));
numbers.put(“two”, new Integer(2));
numbers.put(“three”, new Integer(3));
```

要取出一个数，比如2，用相应的key：

```
Integer n = (Integer)numbers.get(“two”);
System.out.println(“two = ” + n);
```

由于作为key的对象将通过计算其散列函数来确定与之对应的value的位置，因此任何作为key的对象都必须实现hashCode和equals方法。

hashCode和equals方法继承自根类Object，如果你用自定义的类当作key的话，要相当小心，按照散列函数的定义，如果两个对象相同，即obj1.equals(obj2)=true，则它们的hashCode必须相同，但如果两个对象不同，则它们的hashCode不一定不同，如果两个不同对象的hashCode相同，这种现象称为冲突，冲突会导致操作哈希表的时间开销增大，所以尽量定义好的hashCode()方法，能加快哈希表的操作。

如果相同的对象有不同的hashCode，对哈希表的操作会出现意想不到的结果（期待的get方法返回null），要避免这种问题，只需要牢记一条：要同时复写equals方法和hashCode方法，而不要只写其中一个。

Hashtable是同步的（函数体由synchronized修饰）。

HashMap类

HashMap和Hashtable类似，不同之处在于HashMap是非同步的，并且允许null，即null value和null key。

但是将HashMap视为Collection时（values()方法可返回Collection），其迭代子操作时间开销和HashMap的容量成比例。因此，如果迭代操作的性能相当重要的话，不要将HashMap的初始化容量设得过高，或者load factor过低。

HashMap的构造实现：

```
[java] view plain  copy  print  ?
01. static final int DEFAULT_INITIAL_CAPACITY = 16;      // 默认 initialCapacity = 16 (2指数
    的整倍数)
02. static final int MAXIMUM_CAPACITY = 1 << 30;       // 最大容量 (向左位移30位而不是31位，是因
    为int最高位为符号位)
03. static final float DEFAULT_LOAD_FACTOR = 0.75f;      // 默认 loadFactor = 0.75f
```

```
04.
05.  transient Entry[] table;
06.  int threshold;
07.  final float loadFactor;
08.
09.  public HashMap(int initialCapacity, float loadFactor) {
10.         if (initialCapacity < 0)
11.             throw new IllegalArgumentException("Illegal initial capacity: " + initialCap
12.         if (initialCapacity > MAXIMUM_CAPACITY)       // 超过最大容量时，则重置为最大容量
13.             initialCapacity = MAXIMUM_CAPACITY;
14.         if (loadFactor <= 0 || Float.isNaN(loadFactor))
15.             throw new IllegalArgumentException("Illegal load factor: " + loadFactor);
16.
17.         // Find a power of 2 >= initialCapacity
18.         int capacity = 1;
19.         while (capacity < initialCapacity)
20.             capacity <<= 1;         // 容量大小以2的指数级增长
21.
22.         this.loadFactor = loadFactor;
23.         threshold = (int)(capacity * loadFactor);
24.         table = new Entry[capacity];
25.         init();
26.     }
27.
28.  void init() {
29.     }
30.
31.  public HashMap(int initialCapacity) {
32.         this(initialCapacity, DEFAULT_LOAD_FACTOR);
33.     }
34.
35.  public HashMap() {
36.         this.loadFactor = DEFAULT_LOAD_FACTOR;
37.         threshold = (int)(DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR);
38.         table = new Entry[DEFAULT_INITIAL_CAPACITY];
39.         init();
40.     }
41.
42.  public HashMap(Map<? extends K, ? extends V> m) {
43.         this(Math.max((int) (m.size() / DEFAULT_LOAD_FACTOR) + 1, DEFAULT_INITIAL_CAPACI
44.         putAllForCreate(m);
45.     }
```

WeakHashMap类

WeakHashMap是一种改进的HashMap，也是非同步的，它对key实行“弱引用”，如果一个key不再被外部所引用，那么该key可以被GC回收。

使用场景比较

1） 同步性

Vector是同步的。这个类中的一些方法保证了Vector中的对象是线程安全的。

ArrayList则是异步的，因此ArrayList中的对象并不是线程安全的。

因为同步的要求会影响执行的效率，所以如果你不需要线程安全的集合那么使用ArrayList是一个很好的选择，这样可以避免由于同步带来的不必要的性能开销。

2） 数据增长

从内部实现机制来讲ArrayList和Vector都是使用数组(Array)来控制集合中的对象。

当你向这两种类型中增加（插入）元素的时候，如果元素的数目超出了内部数组目前的长度，它们都需要扩展内部数组的长度，Vector缺省情况下自动增长原来一倍的数组长度，ArrayList是原来的50%，所以最后你获得的这个集合所占的空间总是比你实际需要的要大。所以如果你要在集合中保存大量的数据那么使用Vector有一些优势，因为你可以通过设置集合的初始化大小来避免不必要的资源开销。

3） 使用模式

在ArrayList和Vector中，从一个指定的位置（通过索引）查找数据或是在集合的末尾增加、移除一个元素所花费的时间是一样的，这个时间我们用O(1)表示。

但是，如果在集合的其他位置增加或移除元素那么花费的时间会呈线形增长：O(n-i)，其中n代表集合中元素的个数，i代表元素增加或移除元素的索引位置。为什么会这样呢？以为在进行上述操作的时候集合中第i和第i个元素之后的所有元素都要执行位移的操作。这一切意味着什么呢？

这意味着，你只是查找特定位置的元素或只在集合的末端增加、移除元素，那么使用Vector或ArrayList都可以。如果是其他操作，你最好选择其他的集合操作类。比如，LinkList集合类在增加或移除集合中任何位置的元素所花费的时间都是一样的?O(1)，但它在查询索引一个元素的使用时却比较慢O(i)，其中i是索引的位置。

使用ArrayList也很容易，因为你可以简单的使用索引来代替创建iterator对象的操作。

LinkList也会为每个插入的元素创建对象，所有你要明白它也会带来额外的开销。

最后，在《Practical Java》一书中Peter Haggar建议使用一个简单的数组（Array）来代替Vector或ArrayList。尤其是对于执行效率要求高的程序更应如此。因为使用数组(Array)避免了同步、额外的方法调用和不必要的重新分配空间的操作。

总结

如果涉及到堆栈、队列等操作，应该考虑用List；

对于需要快速插入，删除元素，应该使用LinkedList；

如果需要快速随机访问元素，应该使用ArrayList。

如果程序在单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步的类，其效率较高，
如果多个线程可能同时操作一个类，应该使用同步的类。

要特别注意对哈希表的操作，作为key的对象要正确复写equals和hashCode方法。
尽量返回接口而非实际的类型，如返回List而非ArrayList，这样如果以后需要将ArrayList换成LinkedList时，客户端代码不用改变，这就是针对抽象编程。

参考推荐：

ARRAYLIST VECTOR LINKEDLIST区别

ArrayList、LinkedList、Vector的关系和区别

Java 集合类Array、List、Map区别和联系

C++ hash_map 与 Java HashMap 的区别

Java util之常用数据类型特性盘点 （推荐）

cplusplus.com

上一篇 Android 学习小结

下一篇 String、StringBuilder、StringBuffer 用法比较

我的同类文章

Java/JSP（46）

- Log4j 日志详细用法 2014-07-23 阅读 6438
- Fat jar打包工具 2014-03-09 阅读 16409
- MongoDB Java 连接 2014-01-15 阅读 11850
- tomcat OutOfMemoryError 2013-10-02 阅读 4247
- Spring分布式事务实现 2013-09-01 阅读 7639
- js处理json和字符串示例 2013-09-07 阅读 3303
- Spring事务属性详解 2013-09-01 阅读 9859
- JVM 参数调优 2013-08-23 阅读 1782
- JVM 基础知识 2013-08-17 阅读 5164
- JConsole 使用总结 2013-08-24 阅读 1905
- eclipse.ini 内存设置 2013-07-23 阅读 3132

参考知识库

猜你在找

Docker入门到实践
iOS进阶开发-调试程序
数据结构（C版）
Java之路
C++语言基础
ArrayListLinkedList VectorMap 用法比较
转载 ArrayList Vector LinkedList 区别与用法 以及对
Java容器使用总结
java基础知识
Java学习笔记

查看评论

暂无评论

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack
FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap