# 集合概述

为了保存数量不确定的数据，以及保存具有映射关系的数据，java提供了集合类，集合类主要负责保存数据，因此集合类也被称为容器类。所有的集合类都位于java.util包下，后来为了处理多线程环境下的并发安全问题，java 5还在java.util.concurrent包下提供了一些多线程支持的集合类。

# 集合中的接口

Java集合类主要由两个接口派生而出：Collection和Map接口。Collection和Map接口是Java集合的根接口，这两个接口包含了一些子接口或实现类。List和Set接口是Collection接口的子接口。

# 访问集合中的元素

如果要访问List集合中的元素，可以直接根据元素的索引来访问。如果访问Set集合中的元素，则只能根据元素本身来访问。如果访问Map集合中的元素，可以根据每项元素的key来访问value。

# Collection集合的用法

添加元素、删除元素、返回集合中元素的个数和清空整个集合。

# Collection接口的方法

## boolean add(E e)

//添加元素

## boolean addAll(Collection c)

//添加集合

## void clear()

//清空集合

## boolean contains(Object o)

//集合中包含指定的元素，返回true

## boolean equals(Object o)

//比较集合与指定对象是否相等

## int hashCode()

//返回集合的哈希码值

## boolean isEmpty()

//集合不包含元素，返回true

## Iterator iterator()

//返回集合元素上进行的迭代器

## boolean remove(Object o)

//删除元素

## boolean removeAll(Collection c)

//删除集合

## boolean retainAll(Collection c)

//从列表中移除未包含在指定collection中的所有元素

## int size()

//返回集合的大小

## Object[] toArray()

//返回包含集合中所有元素的数组（把集合转换为对象数组）

## T[] toArray(T[] a)

//返回包含此 collection 中所有元素的数组，返回数组的运行时类型与指定数组的运行时类型相同

# Iterator接口

## 基本概念

Iterator接口也是Java集合框架的成员，但是与Collection、Map接口不同，Collection、Map集合主要用于存储其他对象，而Iterator接口主要用于遍历Collection集合中的元素，Iterator也被称为迭代器。Iterator接口隐藏了各种Collection实现类的底层细节，向应用程序提供了遍历Collection集合元素的统一编程接口。

## 方法

### boolean hasNext()

//如果仍有元素可以迭代，则返回true。

### Object next()

//返回迭代的下一个元素。

### void remove()

//从迭代器指向的collection集合中移除迭代器返回的最后一个元素。

## 其他说明

Iterator仅用于遍历集合，Iterator本身并不提供存储对象的能力，如果需要创建Iterator对象，则必须有一个被迭代的集合，没有集合的Iterator对象没有任何价值。当使用Iterator对集合进行遍历时，Iterator并不是把集合元素本身传给了迭代变量，而是把集合元素的值传给了迭代变量，所以修改迭代变量的值对集合元素本身没有任何影响。当使用Iterator对集合进行遍历时，集合里的元素不能被改变，只有通过remove()方法删除上一次next()方法返回的集合元素才可以，否则引发程序异常。

Iterator迭代器采用的是快速失败机制，一旦在迭代过程中检测到该集合已经被修改（通常是程序中的其他线程修改），程序就会引发异常，而不是显示修改后的结果，这样可以避免共享资源而引发的潜在问题。

# List接口

## 基本概念

List接口代表一个有序、可重复的集合，集合中每个元素都有其对应的顺序索引，可以通过索引访问指定位置的集合元素，List集合默认按元素的添加顺序设置元素的索引。

所有的List实现类都可以调用方法来操作集合元素，与Set集合元素相比，List集合增加了根据索引来插入、替换、删除集合元素的方法。List判断两个对象相等只要通过equal()方法比较返回true即可。与Set提供了一个iterator()方法不同，List还额外提供了一个listIterator()方法，该方法返回一个ListIterator对象，ListIterator接口继承了Iterator接口，提供了专门操作List的方法，ListIterator接口在Iterator接口的基础上添加了如下方法：

（1）boolean hasPrevious()

（2）Object previous()

（3）void add()

## List接口的主要方法

### void add(int index,Object obj)

//向指定位置添加元素

### boolean add(Object obj)

//添加元素

### boolean addAll(Collection c)

//添加指定 collection 中的所有元素到此列表的结尾

### boolean addAll(int index,Collection c)

//将指定 collection 中的所有元素都插入到列表中的指定位置

### Object get(int index)

//查找列表中指定位置的元素

### int indexOf(Object obj)

//返回集合中第一次出现指定元素的索引，如果此列表不包含该元素，则返回-1。

### int lastindexOf(Object obj)

//返回集合中最后出现指定元素的索引，如果集合中不包含此元素，则返回-1.

### ListIterator listIterator()

//返回集合元素的迭代器

### Object remove(int index)

//删除列表中指定位置的元素

### boolean remove(Object o)

//删除列表中的元素

### boolean removeAll(Collection<?> c)

//删除包含的集合

### Object set(int index,Object obj)

//用指定元素替换列表中指定位置的元素

### List subList(int fromIndex,int toIndex)

//返回列表中指定的 fromIndex（包括 ）和 toIndex（不包括）之间的部分列表

## List集合的实现类

ArrayList、LinkedList、Vector

ArrayList和Vector类都是基于数组实现的List实现类，所以ArrayList和Vector类封装了一个动态的、允许再分配的Object[]数组，ArrayList或Vector对象使用initialCapacity参数来设置该数组的长度，当向ArrayList或Vector类中添加元素超出数组的长度时，它们的initialCapacity参数会自动增加。如果开始就知道ArrayList或Vector类中保存多少个元素，则可以在创建它们时就指定initialCapacity大小，如果创建空的ArrayList或Vector集合时不指定initialCapacity参数，则Object[]数组的长度默认为10。

# ArrayList实现类

## 基本概念

ArrayList就是传说中的动态数组，就是Array的复杂版本，它提供了如下一些好处： 动态的增加和减少元素 ，实现了Collection和List接口，灵活的设置数组的大小。

# ArrayList和Vector类的区别

ArrayList类是线程不安全的，当多个线程访问同一个ArrayList时，如果有超过一个线程修改了ArrayList集合，则程序必须手动保证该集合的同步性，但是Vector集合无须程序保证该集合的同步性，因为Vector是线程安全的，所以Vector的性能比ArrayList的性能要低，实际上即使需要保证List集合线程安全，也同样不推荐使用Vector实现类，使用Collections工具类可以将一个ArrayList变成线程安全的。

# LinkedList

## 基本概念

LinkedList是List接口的实现类，可以根据索引随机访问集合中的元素，LinkedList实现了Deque接口，因此它可以被当成双端队列来使用，也可以被当成栈来使用。LinkedList不是同步的，如果多个线程同时访问一个链接列表，而其中至少一个线程从结构上修改了该列表，则必须保持它同步。

## 特有的方法

### boolean offer(E e)

//将指定元素添加到此列表的末尾（最后一个元素）

### boolean offerFirst(E e)

//在此列表的开头插入指定的元素

**boolean offerLast(E e)**

//在此列表末尾插入指定的元素

### E peek()

//获取但不移除此列表的第一个元素

## 示例代码

LinkedList li = new LinkedList();

li.offer("a");//将字符串元素加入队列的尾部

li.push("b");//将字符串元素加入栈的顶部

li.offerFirst("c");//将字符串元素加入栈的顶部

System.out.println(li);//输出集合

System.out.println(li.peek());//输出集合中的栈的顶部

System.out.println(li.peekFirst());//输出集合中的栈的顶部

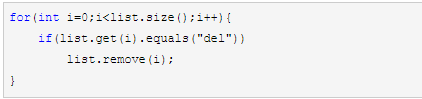
System.out.println(li.peekLast());//输出集合中的栈的底部

# LinkedList与ArrayList的区别

LinkedList与ArrayList的实现机制不同，ArrayList内部以数组的形式保存集合中的元素，因此随机访问集合元素时有较好的性能，而LinkedList内部以链表的形式来保存集合中的元素，因此随机访问集合元素的性能较差，但在插入、删除元素时性能非常出色（只需改变指针所指的地址即可）。对于所有内部基于数组的集合实现，例如ArrayList，使用随机访问的性能比使用Iterator迭代访问的性能要好，因为随机访问会被映射成对数组元素的访问。

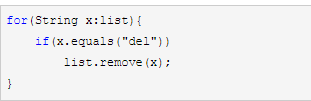
# List集合中删除指定的元素

## 使用for循环去删除



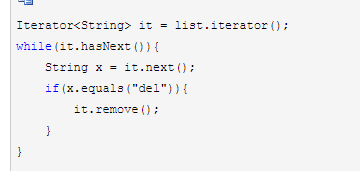
这种方式的问题在于删除某个元素后，list的大小发生了变化，而你的索引也在变化，所以会导致你在遍历的时候漏掉某些元素。比如当你删除第1个元素后，继续根据索引访问第2个元素时，因为删除的关系后面的元素都往前移动了一位，所以实际访问的是第3个元素。因此这种方式可以用在删除特定的一个元素时使用，但不适合循环删除多个元素时使用。

## 增强for循环



这种方式的问题在于，删除元素后继续循环会报错误信息ConcurrentModificationException，因为元素在使用的时候发生了并发的修改，导致异常抛出。但是删除完毕马上使用break跳出，则不会触发报错。

## iterator遍历



这种方式可以正常的循环及删除。但要注意的是，使用iterator的remove方法，如果用list的remove方法同样会报上面提到的ConcurrentModificationException错误。

## 总结

（1）循环删除list中特定一个元素的，可以使用三种方式中的任意一种，但在使用中要注意上面分析的各个问题。

（2）循环删除list中的多个元素时，应该使用迭代器iterator方式。

# 各种线性表的性能分析

Java提供的List是一个线性表接口，而ArrayList、LinkedList又是线性表的两种典型实现：基于数组的线性表和基于链的线性表。

Queue代表了队列，Deque代表了双端队列（既可以作为队列使用，也可以作为栈来使用）。

数组以一块连续内存区来保存所有的数组元素，所有数组在随机访问时性能最好，所有内部以数组作为底层实现的集合在随机访问时性能较好；而内部以链表作为底层实现的集合在执行插入、删除操作时有很好的性能；进行迭代操作时，以链表作为底层实现的集合比以数组作为底层实现的集合性能好。

# 关于使用List集合的建议

如果需要遍历List集合元素，对于ArrayList、Vector集合，应该使用随机访问方法（get）来遍历集合元素，这样性能更好。对于LinkedList集合，则应该采用迭代器（Iterator）来遍历集合元素。如果需要经常执行插入、删除操作来改变List集合的大小，则应该采用LinkedList集合而不是ArrayList。使用ArrayList、Vector集合需要经常重新分配内部数组的大小，其时间开销常常是使用LinkedList的时间开销的几十倍，效果很差。如果有多个线程需要同时访问List集合中的元素，开发者可考虑使用Collections类将集合包装成线程安全的集合。

# Set接口

## 基本概念

Set集合存储无序、不能重复的数据，Set集合判断两个对象相同不是使用==运算符，而是使用equal()方法进行判断的，也就是说只要两个对象用equal方法进行判断返回true时，Set集合就不会接受这两个对象，反之，只要两个对象用equal方法比较返回false，Set就会接受这两个对象。

## Set接口的实现类

HashSet、LinkedHashSet、TreeSet。

## 方法

### boolean add(E e)

//添加元素

### void clear()

//清空集合中的所有元素

### boolean contains(Object o)

//如果此 set 包含指定元素，则返回 true

### boolean equals(Object o)

//比较指定对象与Set的相等性

### boolean isEmpty()

//集合不包含任何元素，返回true

### Iterator iterator()

//返回集合元素的迭代器

### boolean remove(Object o)

//删除元素

### int size()

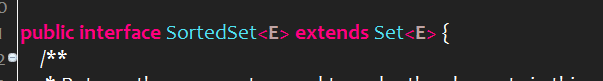
//返回集合的大小

### Object[] toArray()

//返回Set集合中所有元素的数组

# SortedSet接口

## 源码定义



# HashSet实现类

## 基本概念

HashSet是Set接口的典型实现，大多数时候使用Set集合时就是使用这个实现类，HashSet按Hash算法来存储集合中的元素，因此具有很好的存取和查找性能。HashSet集合判断两个元素相等的标准是两个对象通过equal()方法比较相等，并且两个对象的hashCode()方法返回值也相等。如果需要把某个类的对象保存到HashSet集合中，重写这个类的equal()方法和hashCode()方法时，应该尽量保证两个对象通过equal()方法比较返回true时，它们的hashCode()方法返回值也相等。

## HashSet的特点

（1）不能保证元素的排列顺序，顺序有可能发生变化。（无序）

（2）HashSet不是同步的，如果多个线程同时访问一个HashSet，必须通过代码来保证其同步。

（3）集合元素值可以是null。（List集合元素值也可以存储null）

# LinkedHashSet实现类

LinkedHashSet是HashSet的子类，也是根据元素的hashCode值来决定元素的存储位置，但它同时使用链表维护元素的次序，这样使得元素看起来是以插入的顺序保存的，也就是说，当遍历LinkedHashSet集合里的元素时，LinkedHashSet将会按元素的添加顺序来访问集合里的元素。（有序的）

LinkedHashSet需要维护元素的插入顺序，因此性能低于HashSet的性能，但在迭代访问Set里全部元素时将有很好的性能，因为它以链表来维护内部顺序。LinkedHashSet使用了链表来记录集合元素的添加顺序，但LinkedHashSet依然是HashSet，因此它依然不允许集合元素重复。LinkedHashSet是有序不重复的。

# TreeSet实现类

## 基本概念

TreeSet是SortSet接口的实现类，TreeSet可以确保集合元素处于排序状态。

TreeSet中的元素是有序的，所以增加了访问第一个、前一个、后一个、最后一个元素的方法，并提供了三个从TreeSet中截取子TreeSet的方法。

TreeSet并不是根据元素的插入顺序进行排序的，而是根据元素实际值的大小来进行排序的，TreeSet采用红黑树的数据结构来存储集合元素，TreeSet支持两种排序方法：自然排序和定制排序，在默认情况下，TreeSet采用自然排序。 自然排序：TreeSet会调用集合元素的compareTo(Object obj)方法来比较元素之间的大小关系，然后将集合元素按升序排列。

说明：如果试图把一个对象添加到TreeSet时，则该对象的类必须实现Comparable接口，否则程序将会抛出异常，还有向TreeSet中添加的应该是同一个类的对象，否则也会引发ClassCastException异常。

## 方法

### （1）Comparator comparator()

如果TreeSet采用了定制排序，则该方法返回定制排序所使用的Comparator，如果TreeSet采用了自然排序，则返回null。默认采用自然排序。

### （2）Object first()

返回集合中的第一个元素。

### （3）Object last()

返回集合中的最后一个元素。

### （4）Object lower(Object e)

返回集合中位于指定元素之前的元素。

### （5）Object higher(Object e)

返回集合中位于指定元素之后的元素。

### （6）SortedSet subSet(fromElement,toElement)

返回此Set的子集合，范围从fromElement(包含)到toElement(不包含)。

### （7）SortedSet headSet(toElement)

返回此Set的子集，由小于toElement的元素组成。

### （8）SortedSet tailSet(fromElement)

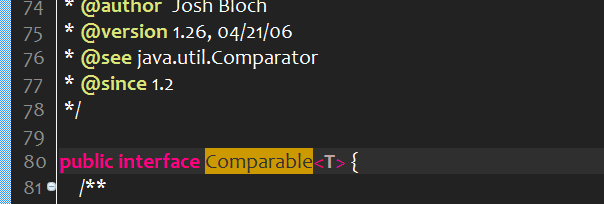
返回此Set的子集，由大于或等于fromElement的元素组成。

# Comparable接口

## 基本概念

Java提供了一个Comparable接口，该接口里定义了一个compareTo(Object obj)方法，该方法返回一个数据值，实现该接口的类必须实现该方法，实现了该接口的类的对象就可以比较大小，当一个对象调用该方法与另一个对象进行比较时，例如：obj.compareTo(obj)，如果该方法返回0，则表明这两个对象相等，如果该方法返回一个正整数，则表明obj1大于obj2，如果该方法返回一个负整数，则表明obj1小于obj2。Java的一些常用类已经实现了Comparable接口，并提供了比较大小的标准。这个接口称为内比较器，就是在这个对象的内部实现排序算法，Java很多基础类都实现了Comparable接口。

## 源码



## 方法

### int compareTo(T o)

比较此对象与指定对象的大小。

## 实现Comparable接口的常用类

（1）BigDecimal、BigInteger以及所有的数字型对应的包装类，按它们对应的数值大小进行比较。

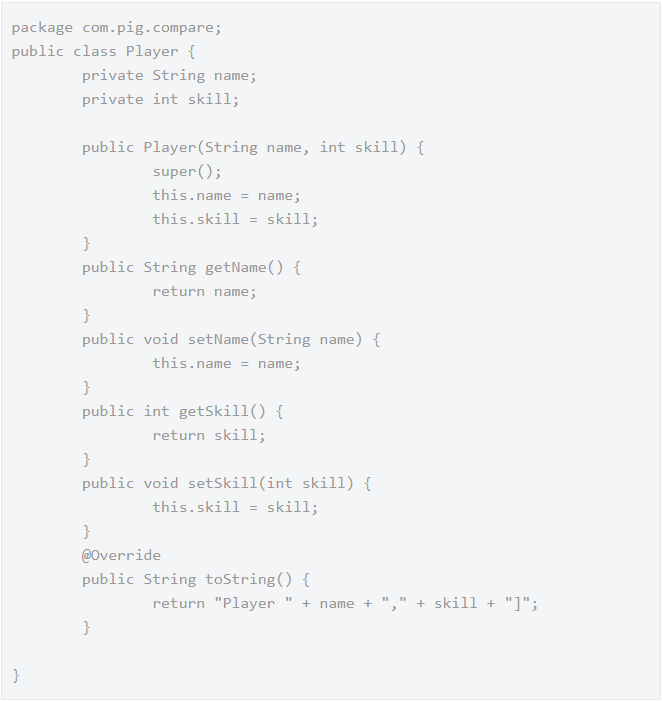
（2）Character：按字符的unicode值进行比较。

（3）Boolean：true对应的包装类实例大于false对应的包装类实例。

（4）**String**：按字符串中字符的unicode值进行比较。

（5）**Date、Time**：后面的时间、日期比前面的时间、日期大。

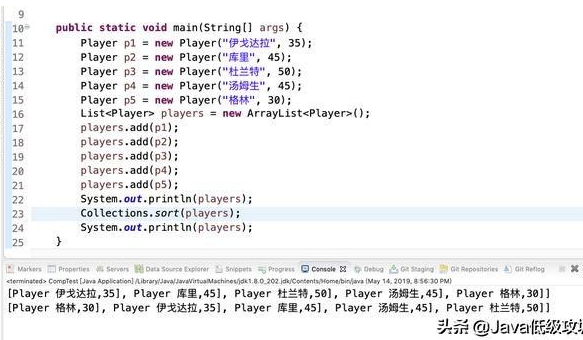
## 示例代码



首先定义一个Player，有名字和技能，假如是要实现内比较器，也就是实现Comparable的接口，在Player原有代码增加如下：

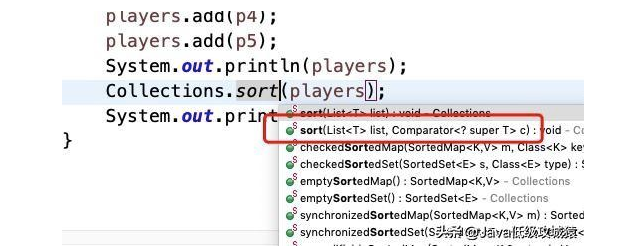


这样就实现了一个拥有比较器的对象，测试一下：

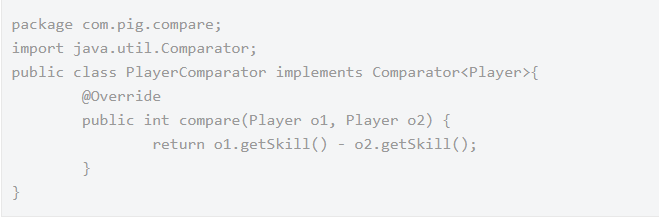


可以看出，经过Collections.sort()之后，成了一个升序的排列，这个升序和你的排序算法相关，也就是compareTo方法的实现。

Collections.sort()方法不但可以传进一个实现了比较器的对象集合，还可以指定比较器对象：



这个比较器对象，就是我们接下来要说的外部比较器：Comparator。对于外部比较器，我们不需要在我们写的实体类上加比较器，然后实现一个比较方法，使用外部比较器可以保留实体类的原汁原味，而只是在外部比较器中对指定实体类对象进行排序，代码如下：



而Player对象就是原来我们写的，不加Comparable接口的那个对象，运行一下看结果：



也是达到了同样的结果。

不知道大家注意到没有，Comparator加上了@FunctionalInterface注解，这是一个函数式接口，那我们就别客气了，就利用函数式接口的特性来实现比较器：



不用再专门定义一个外部比较器，然后传进排序集合工具类中进行排序了，方便快捷，减少代码量。

总结如下：

1、一个是内比较器（Comparable）对代码不友好，一个是外比较器（Comparator）没有代码侵入。

2、位置不同，Comparator位于包java.util下，而Comparable位于包 java.lang下

3、Comparator 是策略模式（strategy design pattern），就是不改变对象自身，而用一个策略对象来改变它的行为。

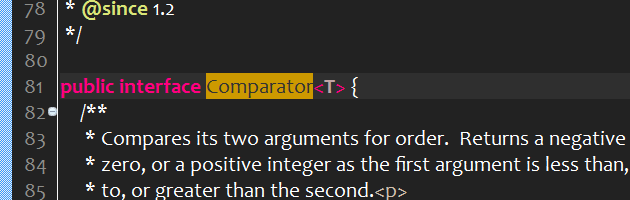
4、Comparator是一个函数式接口，可以使用 Lambda表达式来实现比较器。

# Comparator接口

## 基本概念

这个接口称为外比较器。

## 源码



## 方法

### int compare(T o1, T o2)

比较两个参数的顺序

### boolean equals(Object obj)

与其他对象进行比较

# Map接口

## 基本概念

用于保存具有映射关系的数据，因此Map集合里保存着两组值，一组值用于保存Map里的key，另一组值用于保存Map里的value，key和value都可以是任何引用类型的数据。Map的key不允许重复，即同一个Map对象的任何两个key通过equal方法比较总是返回false。key和value之间存在单向的一对一关系，即通过指定的key，总能找到唯一的、确定的value。从Map中取出数据时，只要给出指定的key，就可以取出对应的value。

## Map接口的常用方法

### void clear()

//移除Map中的所有元素

### boolean containsKey(Object key)

//包含指定的键，返回true

### boolean containsValue(Object value)

//包含指定的值，返回true

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

//返回此映射中包含的映射关系的set集合。 这个方法用于循环遍历Map集合中的数据

### Object get(Object key)

//返回指定键的值

### boolean isEmpty()

//Map集合为空，返回true

### Set<K> keySet()

//返回此映射中包含的键的Set集合

### Object put(Object key,Object value)

//添加数据

### void putAll(Map m)

//添加Map集合

### Object remove(Object key)

//删除数据

### int size()

//返回集合的大小

### Collection values()

//返回Map集合中包含的值的 [Collection](mk:@MSITStore:C:\Users\jinshihui\Desktop\我的编程知识总结\Java\JDK_API_1_6_zh_CN.CHM::/java/util/Collection.html) 视图

# Map.Entry类

Map中有一个内部类Entry，该类封装了一个key-value对，Entry包含如下三个方法：

（1）Object getKey() //返回此项对应的键

（2）Object getValue() //返回此项对应的值

（3）Object setValue(V value) //设置值

# Map接口的实现类

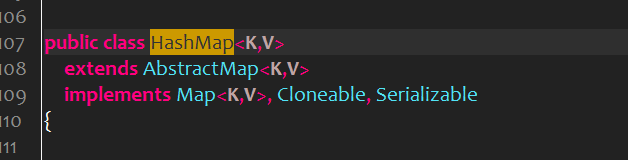
## HashMap

### 基本概念

HashMap 是一个散列表，它存储的内容是键值对(key-value)映射。HashMap 的实现不是同步的，这意味着它不是线程安全的。它的key、value都可以为null，此外HashMap中的映射不是有序的。在HashMap中，key-value总是会当做一个整体来处理，系统会根据hash算法来计算key-value的存储位置，我们总是可以通过key快速地存、取value。

HashMap 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。通常**默认加载因子是 0.75**, 这是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。

### 源码定义



### 构造函数

#### public HashMap(int initialCapacity, float loadFactor)

//构造一个带指定初始容量和加载因子的空 HashMap

#### public HashMap(int initialCapacity)

//构造一个带指定初始容量和默认加载因子 (0.75) 的空 HashMap

#### public HashMap()

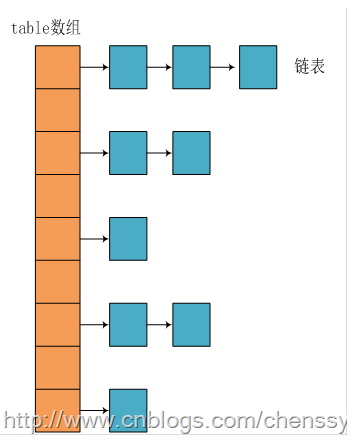
//构造一个具有默认初始容量 (16) 和默认加载因子 (0.75) 的空 HashMap

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

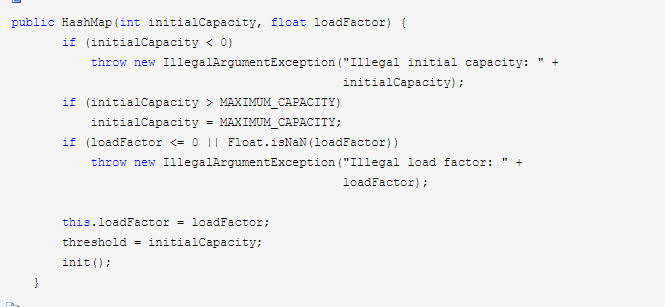
在这里提到了两个参数：初始容量，加载因子。这两个参数是影响HashMap性能的重要参数，其中容量表示哈希表中桶的数量，初始容量是创建哈希表时的容量，加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，它衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间是O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。系统默认负载因子为0.75，一般情况下我们是无需修改的。

### 数据结构

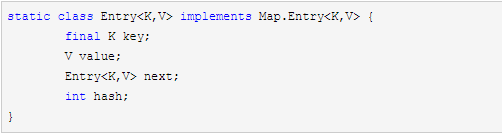
我们知道在Java中最常用的两种结构是数组和模拟指针(引用)，几乎所有的数据结构都可以利用这两种来组合实现，HashMap也是如此。实际上HashMap是一个“链表散列”，如下是它数据结构：



从上图我们可以看出HashMap底层实现还是数组，只是数组的每一项都是一条链。其中参数initialCapacity就代表了该数组的长度。下面为HashMap构造函数的源码：



从源码中可以看出，每次新建一个HashMap时，都会初始化一个table数组，table数组的元素为Entry节点。



其中Entry为HashMap的内部类，它包含了键key、值value、下一个节点next，以及hash值，这是非常重要的，正是由于Entry才构成了table数组的项为链表。

## Hashtable

和[HashMap](http://www.cnblogs.com/skywang12345/p/3310835.html)一样，Hashtable 也是一个散列表，它存储的内容是键值对(key-value)映射。Hashtable 继承于Dictionary，实现了Map、Cloneable、java.io.Serializable接口。Hashtable 的函数都是同步的，这意味着它是线程安全的。它的key、value都不可以为null。此外Hashtable中的映射不是有序的。

## LinkedHashMap

LinkedHashMap是HashMap的子类，LinkedHashMap不是线程安全的，LinkedHashMap使用双向链表来维护key-value对的次序，该链表负责维护Map的迭代顺序，迭代顺序与key-value对的插入顺序保持一致。HashMap存储元素的顺序是无序的，LinkedHashMap可以避免对HashMap、Hashtable里的key--value对进行排序（只要插入key-value对时保持顺序即可），同时又可避免使用TreeMap所增加的成本。LinkedHashMap需要维护元素的插入顺序，因此性能略低于HahMap的性能，但因为它以链表来维护内部顺序，所以迭代访问Map里的全部元素时将有较好的性能。

LinkedHashMap lhm = new LinkedHashMap();

lhm.put("语文", 80);

lhm.put("数学", 90);

lhm.put("英语", 80);

System.out.println(lhm);

for(Object str:lhm.keySet()){//使用增强for循环遍历Map集合

System.out.println(str+""+lhm.get(str));

}

## TreeMap

SortedMap接口有一个TreeMap实现类，TreeMap就是一个红黑树数据结构，每个key-value对作为红黑树的一个节点，TreeMap存储key-value对时，需要根据key-value对节点进行排序，TreeMap可以保证所有的key-value对处于有序状态。TreeMap默认按键的值升序排序，可以定制。TreeMap中判断两个key相等的标准是两个key通过compareTo()方法返回0，TreeMap即认为这两个key是相等的。再次强调：Set和Map的关系十分密切，Java源码就是先实现了HashMap、TreeMap等集合，然后通过包装一个所有的value都为null的Map集合实现了Set集合类。与TreeSet类似，TreeMap中提供了一系列根据key顺序访问key-value对的方法。

# HashMap和Hashtable的区别

HashMap和Hashtable是Map接口的典型实现类，它们之间的关系完全类似于ArrayList和Vector的关系，HashMap和Hashtable存在两点典型区别：

（1）Hashtable是一个线性安全的Map实现，但HashMap是线程不安全的实现，所以HashMap比Hashtable的性能高一点，但如果有多个线程访问同一个Map对象，使用Hashtable实现类会更好。

（2）Hashtable不允许使用null作为key和value，如果试图把null值放进Hashtable中，将会引发空指针异常，但是HashMap可以使用null作为key和value。由于HashMap里的key不能重复，所以HashMap里最多只有一个key为null，但是可以有多个value为null。

# Map集合中判断key和value相等的标准

为了成功的在HashMap、Hashtable中存储、获取对象，用作key的对象必须实现hashCode()方法和equal()方法。

HashMap、Hashtable判断两个key相等的标准是：两个key通过equals()方法比较返回true，两个key的hashCode值也相等。

HashMap、Hashtable判断两个value相等的标准更简单，只要两个对象通过equals()方法比较返回true即可。

# 使用Properties类读写属性文件

## 基本概念

Properties类是Hashtable的子类，该对象在处理属性文件时特别方便，Properties类可以把Map对象和属性文件关联起来，从而可以把Map对象中的key-value对写入属性文件中，也可以把属性文件中的“属性名=属性值”加载到Map对象中，由于属性文件里的属性名、属性值只能是字符串类型，所以Properties里的key、value都是字符串类型，该类提供了如下方法来修改Properties里的key、value值。

## 方法

### String getProperty(String key)

//用指定的键在此属性列表中搜索属性

### String getProperty(String key,String defaultValue)

//用指定的键在属性列表中搜索属性

### Object setProperty(String key,String value)

//设置键和值

### void load(InputStream inStream)

//从属性文件中读取key-value对，把加载到的key-value对追加到Properties类里。（Properties是hashtable的子类，它不保证key-value对之间的顺序）

### void store(OutputStream out,String comments)

//将Properties类中的key-value对写入到指定的属性文件（以输出流表示）中。

### void storeXML(OutputStream out,String comments);

//将Properties中的key-value对写入到指定的XML文件（以输出流表示）中。

### loadFromXML(InputStream inStream);

//从XML文件中读取key-value对，把加载到的key-value对追加到Properties里。

## 代码示例

### 把Properties中的键值对输出到属性文件中

Properties pro = new Properties();

pro.put("语文", "90");

pro.put("数学", "80");

System.out.println(pro);

pro.setProperty("英语", "90");

System.out.println(pro);

//把pro对象存放到属性文件中

pro.store(new FileOutputStream("src\\userinfo.properties"), "note");

### 从属性文件中读取键值对并加载到properties集合中

pro1.load(new FileInputStream("src\\userinfo.properties"));

System.out.println(pro1);

说明：Properties可以把key-value对以XML文件的形式保存起来，也可以从XML文件中加载key-value对。

# 各Map实现类的性能分析

对于Map的常用实现类而言，HashMap和Hashtable的效率大致相同，因为它们的实现机制几乎完全一样，但是HashMap通常比Hashtable要快一点，因为Hashtable需要额外的线程同步控制。

TreeMap通常比HashMap、hashtable要慢（尤其在插入、删除key-value对时更慢），因为TreeMap底层采用红黑树来管理key-value对（红黑树的每个节点就是一个key-value对）。

LinkedHashMap比HashMap慢一点，因为它需要维护链表来保持Map中key-value对的添加顺序。

对于一般的应用场景，程序应该多考虑使用HashMap，因为HashMap正是为快速查询设计的（HashMap底层其实也是采用数组来存储key-value对）。但如果程序需要一个总是排好序的Map时则可以考虑使用TreeMap。

# Collections类

## 基本概念

Java提供了一个操作Set、List和Map等集合的工具类Collections，这个类在java.util包中，该工具类里提供了大量方法对集合元素进行排序、查询和修改等操作，还提供了将集合对象设置为不可变、对集合对象实现同步控制等方法。

## List集合排序方法

Collections提供了如下几个方法用于对List集合元素进行排序

### （1）static void reverse(List list)

反转集合中的元素

### （2）static void shuffle(List list)

对集合中的元素进行随机排序

### （3）static void sort(List list)

根据元素的自然顺序对集合中的元素按升序进行排序

### （4）static void sort(List list,Comparator c)

根据指定Comparator产生的顺序对集合元素进行排序。

### （5）static void swap(List list,int i,int j)

将集合中的i处元素和j处元素进行交换。

### （6）static void rotate(List list,int distance)

当distance为正数时，将集合的后distance个元素整体移到前面，反之distance为负数时，将集合前的distance个元素整体移到后面。

### 代码示例

ArrayList al = new ArrayList();

al.add(1);

al.add(2);

al.add(-3);

al.add(4);

al.add(-10);

System.out.println(al);//输出集合中的元素

//将集合中的元素进行反转

Collections.reverse(al);

System.out.println(al);

//将集合中的元素按自然顺序进行升序排序

Collections.sort(al);

System.out.println(al);

//将集合中的元素进行随机排序

Collections.shuffle(al);

## Collections提供了查找、替换集合元素的常用方法

### （1）static int binarySearch(List list,Object key)

使用二分搜索法搜索指定的List集合，以获得指定对象在List集合中的索引，如果要使该方法可以正常工作，则必须保证List中的元素已经处于有序状态。

### （2）static Object max(Collection coll)

根据元素的自然顺序，返回给定集合中的最大元素。

### （3）static Object min(Collection coll)

根据元素的自然顺序，返回给定集合中的最小元素。

### （4）static void fill(List list,Object obj)

使用指定元素obj替换指定List集合中的所有元素。

### （5）static int frequency(Collection c,Object o)

返回指定集合中指定元素的出现次数。

### （6）static int indexOfSubList(List source,List target)

返回子List对象在父List对象中第一次出现的位置索引，如果父List中没有出现这样的子List，则返回-1。

### （7）static int latIndexOfSubList(List source,List target)

返回子List对象在父List对象中最后一次出现的位置索引，如果父List中没有出现这样的子List，则返回-1。

### （8）static boolean replaceAll(List list,Object oldVal,Objct newVal)

使用一个新值newVal替换List对象的所有旧值oldVal。

### 代码示例

ArrayList al = new ArrayList();

al.add(2);

al.add(-5);

al.add(0);

al.add(3);

al.add(2);

System.out.println(al);

//输出集合中最大的元素

//输出集合中最小的元素

//将集合中的0用1去代替

//判断集合中元素2出现的次数

//对集合进行排序，从小到大

//搜搜集合中某个元素的索引位置

int num = Collections.max(al);

System.out.println("集合中的最大元素："+num);

int min = Collections.min(al);

System.out.println("集合中的最小元素："+min);

boolean a = Collections.replaceAll(al, 0, 1);

System.out.println(a);

System.out.println(al);

int num1 = Collections.frequency(al, 2);

System.out.println(num1);

Collections.sort(al);

System.out.println(al);

Collections.reverse(al);

System.out.println(al);

Collections.sort(al);

System.out.println(al);

int index = Collections.binarySearch(al,-5);

System.out.println(index);

# 同步控制

## 基本概念

Collections类中提供了多个synchronizedXxx()方法，该方法可以将指定集合包装成线程同步的集合，从而可以解决多线程并发访问集合时的线程安全问题。

Java中常用的集合框架中的实现类HashSet、TreeSet、ArrayList、LinkedList、HashMap和TreeMap都是线程不安全的，如果有多个线程访问它们，而且有超过一个线程试图修改它们，则可能出现错误。Collections提供了多个静态方法可以把它们包装成线程同步的集合。

## 代码示例

Collection coll = Collections.synchronizedCollection(new ArrayList());

List list = Collections.synchronizedList(new ArrayList());

Set set = Collections.synchronizedSet(new HashSet());

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

## 设置不可变对象

Collections提供了三个方法可以返回一个不可变的集合。

### 方法

#### （1）static List emptyXxx()

返回一个空的、不可变的集合对象，此处的集合可以是List、Set、Map。

#### （2）static List singletonXxx()

返回一个只包含指定对象的、不可变的集合对象，此处的集合可以是List、Set、Map。

#### （3）static List unmodifiableXxx()

返回指定集合对象的不可变视图，此处的集合可以是List、Set、Map。

上面三类方法的参数是原有的集合对象，返回值是该集合的“只读”版本，通过Collections提供的三类方法，可以产生“只读”的Collection或Map集合。

# 集合循环遍历的方法

## List集合遍历的方法

List list = new ArrayList();

list.add(3);

list.add(2);

list.add(10);

list.add(4);

list.add(70);

for(int i=0;i<list.size();i++){

int j = (Integer)list.get(i);

System.out.println(j);

}

说明：遍历List集合中的元素，使用for循环去遍历获取。

## Set集合循环遍历的方法

Set<String> set = new HashSet<String>();

Iterator<String> it = set.iterator();

while (it.hasNext()) {

String str = it.next();

System.out.println(str);

}

说明：遍历Set集合中的元素，使用iterator()方法去遍历。

## Map集合循环遍历的方法

（1）Iterator iterator = map.entrySet().iterator();

while(iterator.hasNext()){

Map.Entry mapEntry = (Map.Entry)iterator.next();

Object keyObject = mapEntry.getKey();

Object vaObject = mapEntry.getValue();

System.out.println(keyObject+""+vaObject);

}

（2）Map<String, Object> map = new HashMap<>();

map.put("aa", 11);

map.put("bb", 11);

map.put("cc", 11);

System.out.println(map);

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

while(it.hasNext()){

String key = it.next();

int value = (int) map.get(key);

System.out.println("key="+key+" value="+value);

}

说明：遍历Map集合的数据使用iterator()方法去遍历。

# ConcurrentMap接口

## 方法

### V putIfAbsent(K key, V value);

//如果指定的键尚未与值关联，请将其与给定值相关联。

### boolean remove(Object key, Object value);

//仅当前映射到给定值时才删除键的条目。

### boolean replace(K key, V oldValue, V newValue);

//仅当前映射到给定值时才替换键的条目。

参数：

key：与指定值关联的键

oldValue：预期与指定键关联的值

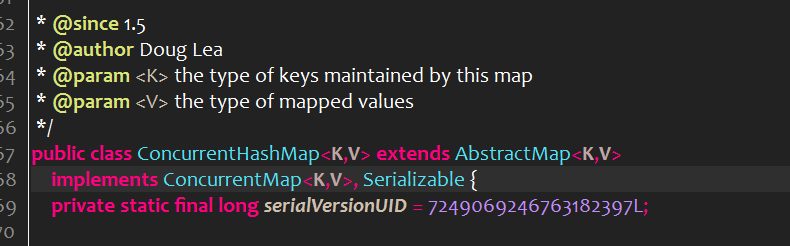
newValue：与指定键关联的值

### V replace(K key, V value);

//仅当前映射到某个值时才替换键的条目

# ConcurrentHashMap类

## 源码定义



## 背景

因为多线程环境下，使用Hashmap进行put操作会引起死循环，导致CPU利用率接近100%，所以在并发情况下不能使用HashMap。

HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时可能会进入阻塞或轮询状态。如线程1使用put方法进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，所以竞争越激烈效率越低。

## 基本概念

JDK5中添加了新的concurrent包，相对同步容器而言，并发容器通过一些机制改进了并发性能。因为同步容器将所有对容器状态的访问都串行化了，这样保证了线程的安全性，所以这种方法的代价就是严重降低了并发性，当多个线程竞争容器时，吞吐量严重降低。因此Java5.0开始针对多线程并发访问设计提供了并发性能较好的并发容器，引入了java.util.concurrent包。

## 并发容器解决的问题

与Vector和Hashtable、Collections.synchronizedXxx()同步容器等相比，util.concurrent中引入的并发容器主要解决了两个问题：

（1）根据具体场景进行设计，尽量避免synchronized，提高并发性。

（2）定义了一些并发安全的复合操作，并且保证并发环境下的迭代操作不会出错。

util.concurrent中容器在迭代时，可以不封装在synchronized中，可以保证不抛异常，但是未必每次看到的都是"最新的、当前的"数据。

ConcurrentHashMap代替同步的Map（Collections.synchronized（new HashMap()）），众所周知HashMap是根据散列值分段存储的，同步Map在同步的时候锁住了所有的段，而ConcurrentHashMap加锁的时候根据散列值锁住了散列值锁对应的那段，因此提高了并发性能。ConcurrentHashMap也增加了对常用复合操作的支持，比如"若没有则添加"：putIfAbsent()，替换：replace()，这2个操作都是原子操作。

说明：原子操作是指不会被[线程调度](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B%E8%B0%83%E5%BA%A6/10226112)机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch （切 [1]  换到另一个线程）

CopyOnWriteArrayList和CopyOnWriteArraySet分别代替List和Set，主要是在遍历操作为主的情况下来代替同步的List和同步的Set，这也就是上面所述的思路：迭代过程要保证不出错，除了加锁，另外一种方法就是"克隆"容器对象。

大家都知道HashMap是非线程安全的，Hashtable是线程安全的，但是由于Hashtable是采用synchronized进行同步，相当于所有线程进行读写时都去竞争一把锁，导致效率非常低下。ConcurrentHashMap可以做到读取数据不加锁，并且其内部的结构可以让其在进行写操作的时候能够将锁的粒度保持地尽量地小，不用对整个ConcurrentHashMap加锁。

## 构造函数

### public ConcurrentHashMap()

### public ConcurrentHashMap(int initialCapacity)

### public ConcurrentHashMap(int initialCapacity,float loadFactor, int concurrencyLevel)

## 方法

### public boolean isEmpty()

### public V remove(Object key)

### public V get(Object key)

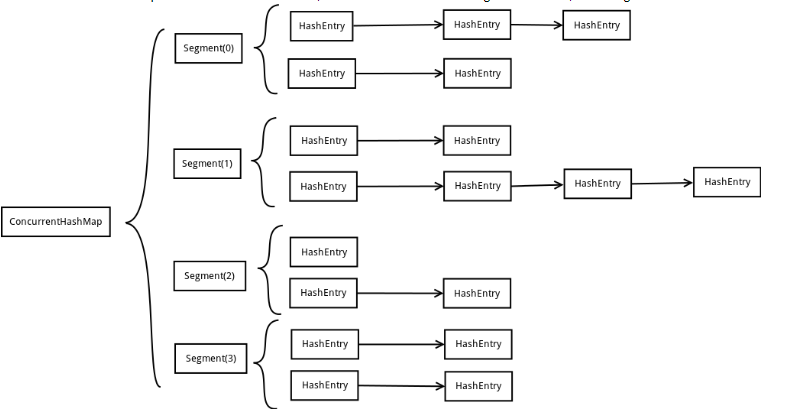
### public V put(K key, V value)

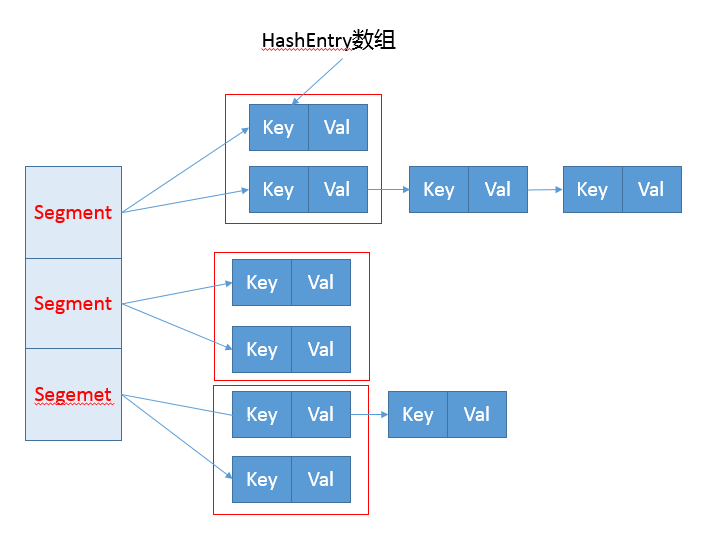
//将指定的键映射到此表中的指定值。键和值都不能为空。可以通过使用等于原始键的键调用get方法来检索该值。

返回：与key关联的以前的值，或者这个key没有映射则返回null。

## ConcurrentHashMap的内部结构

ConcurrentHashMap为了提高本身的并发能力，在内部采用了一个叫做Segment的结构，一个Segment其实就是一个类Hash Table的结构，Segment内部维护了一个链表数组，我们用下面这一幅图来看下ConcurrentHashMap的内部结构：

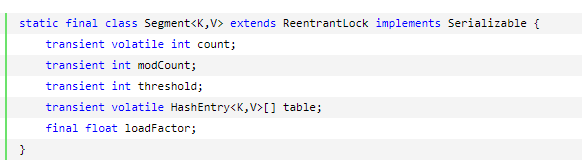




从上面的结构我们可以了解到，ConcurrentHashMap定位一个元素的过程需要进行两次Hash操作，第一次Hash定位到Segment，第二次Hash定位到元素所在的链表的头部，因此，这一种结构的带来的副作用是Hash的过程要比普通的HashMap要长，但是带来的好处是写操作的时候可以只对元素所在的Segment进行加锁即可，不会影响到其他的Segment，这样，在最理想的情况下，ConcurrentHashMap可以最高同时支持Segment数量大小的写操作（刚好这些写操作都非常平均地分布在所有的Segment上），所以，通过这一种结构，ConcurrentHashMap的并发能力可以大大的提高。

### Segment

#### 源码定义



详细解释一下Segment里面的成员变量的意义：

count：Segment中元素的数量

modCount：对table的大小造成影响的操作的数量（比如put或者remove操作）

threshold：阈值，Segment里面元素的数量超过这个值依旧就会对Segment进行扩容

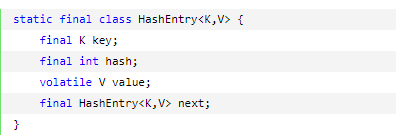
table：链表数组，数组中的每一个元素代表了一个链表的头部

loadFactor：负载因子，用于确定threshold

### HashEntry

#### 源码定义

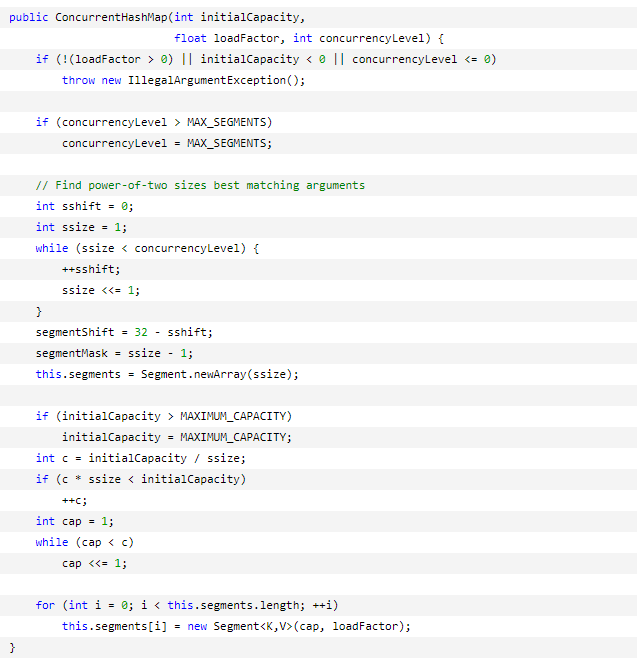
Segment中的元素是以HashEntry的形式存放在链表数组中的，看一下HashEntry的结构：



可以看到HashEntry的一个特点，除了value以外，其他的几个变量都是final的，这样做是为了防止链表结构被破坏，出现ConcurrentModification的情况。

### ConcurrentHashMap的初始化

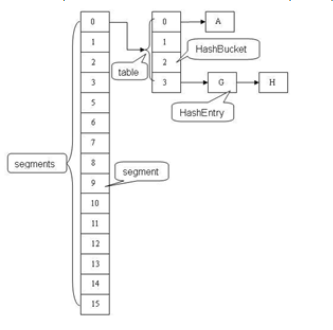
下面我们来结合源代码来具体分析一下ConcurrentHashMap的实现，先看下初始化方法：



## 锁分段技术

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术。

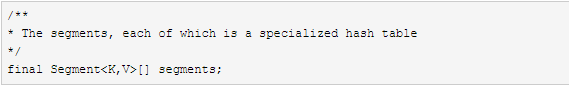
首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在ConcurrentHashMap内部，段数组是final的，并且其成员变量实际上也是final的，但是仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。



ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护着一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。

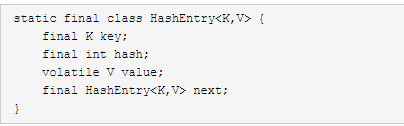
## 源码解读（1.7及之前）

ConcurrentHashMap(1.7及之前)中主要实体类就是三个：ConcurrentHashMap（整个Hash表）,Segment（桶），HashEntry（节点）



## 不变(Immutable)和易变(Volatile)

ConcurrentHashMap完全允许多个读操作并发进行，读操作并不需要加锁。如果使用传统的技术，如HashMap中的实现，如果允许可以在hash链的中间添加或删除元素，读操作不加锁将得到不一致的数据。ConcurrentHashMap实现技术是保证HashEntry几乎是不可变的。HashEntry代表每个hash链中的一个节点，其结构如下所示：



可以看到除了value不是final的，其它值都是final的，这意味着不能从hash链的中间或尾部添加或删除节点，因为这需要修改next 引用值，所有的节点的修改只能从头部开始。对于put操作，可以一律添加到Hash链的头部。但是对于remove操作，可能需要从中间删除一个节点，这就需要将要删除节点的前面所有节点整个复制一遍，最后一个节点指向要删除结点的下一个结点。为了确保读操作能够看到最新的值，将value设置成volatile，这避免了加锁。

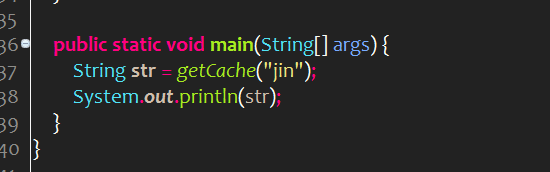
## 应用场景

ConcurrentHashMap适用于读的数量超过写时，当写的数量大于等于读时，ConcurrentHashMap的性能是低于Hashtable和synchronized Map的。这是因为当锁住了整个Map时，读操作要等待对同一部分执行写操作的线程结束。ConcurrentHashMap适用于做cache,在程序启动时初始化，之后可以被多个请求线程访问。正如Javadoc说明的那样，ConcurrentHashMap是HashTable一个很好的替代，但要记住，ConcurrentHashMap比HashTable的同步性稍弱。

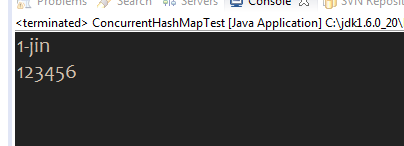
## 作为缓存的代码示例

代码如下：





输出结果：



## 总结

ConcurrentHashMap允许并发的读和线程安全的更新操作，在执行写操作时，ConcurrentHashMap只锁住部分的Map。并发的更新是通过内部根据并发级别将Map分割成小部分实现的，高的并发级别会造成时间和空间的浪费，低的并发级别在写线程多时会引起线程间的竞争。ConcurrentHashMap的所有操作都是线程安全，ConcurrentHashMap返回的迭代器是弱一致性，fail-safe并且不会抛出ConcurrentModificationException异常，ConcurrentHashMap不允许null的键值，可以使用ConcurrentHashMap代替HashTable，但要记住ConcurrentHashMap不会锁住整个Map。