# 集合Collection

1. 集合类继承图：

Hashmap类

Hashtable类

Hashset类

Vector类

Arraylist类

Linkedlist类

List接口

Set接口

collection接口

Map接口

其中set接口的子类还有treeset；map接口的子类还有treemap 。

1. 各集合框架的特点如下：

1：list接口是有序可重复的序列，是有序可重复的集合类的最终父类。

1. Arraylist，linkedList，vector 实现了list接口，所以它们是有序可重复的序列。它们主要区分有：ArrayList查询，遍历快，而增添，删除，修改元素慢，不安全。LinkedList与ArrayList正好相反，但也不安全。而vector是线性同步的，所以它是安全的。操作效率低。
2. Set是不可重复的散列。最常用的子类是hashSet，treeSet。hashSet底层是有hashMap实现的，与treeset相比较hashset查询，遍历效率高。增加，删除，修改元素效率低，treeset则与hashset相反。
3. Hashset与treeset区分的原理如下：

对于HashSet和TreeSet来说，里面的元素是不能重复的，若重复则会覆盖前面的那个元素。而对于HashMap和TreeMap来说，里面存储的是Key-Value对，在所有的键值对中，Key值是不能重复的，但Value值是可以重复的，若重复存储相同的Key值，则会把前面的那个Key-Value对中的Key和Value都覆盖掉。

OK，下面就HashSet、TreeSet和HashMap、TreeMap的存储方式作简单总结。

HashSet的底层是通过HashMap来实现存储的，而TreeSet的底层又是通过TreeMap来实现存储的，而且HashSet和TreeSet中的元素都是以Key值存储的，这也是HashSet和TreeSet的元素不能重复的原因。由此可见，我们只要理解HashMap和TreeMap的存储方式就可以了。

HashMap的存储相对于TreeMap来说要简单，在上一篇文章中已经讲到了，其底层实际上维护的是一个Entry类型的数组table，而具体怎么存储取决添加元素hashcode值和table的长度决定于，例如：String类中重写了hashCode方法，所以我们可以算出hash code值，但对于某些没有重写hashCode方法的类型，则直接调用Object类中的hashCode方法，然而这个方法对我们是不透明的，所以不能算出hashcode值，所以排序算法不得而知。这里我认为没必要深究，我们只需知道HashMap中的元素是乱序的即可，尽管这种乱序也是“有序”的。

而TreeMap的排序则透明了许多，与HashMap最大的不同点是：TreeMap底层维护的是一个Entry类型的双向链表，而不是数组，每个Entry类中都定义有key、value、parent、left、right等属性， key、value表示节点中的key-value对，left、right则指向链接到当前节点左右两边的那两个节点，parent表示最后加入链表的那个节点的父节点，只是起过渡的作用。

下面分两种情况来讨论添加元素的具体操作：

第一种是添加的Key值类型是Java已有的类型

我们知道TreeMap的底层代码是根据Key-Value对中的Key值来排序的，这种情况Coder可以自行定义比较器，当然也可以用TreeMap中默认的。若是自己定义的比较器，则在生成TreeSet或TreeMap对象的时候将其传入，如：

5：map接口：

Map接口是以键-值对来存储数组的。主要的子类有hashmap，treemap，hashtable。Hashmap底层是由一维对象数组来实现，所以查询，遍历快，而增添，删除，修改元素慢，不安全。而 treeMaap底层则是由双链表来实现的。所以增添，删除，修改元素效率高。 遍历效率低。也不安全。Hashtable操作效率低，但安全。

1. ArrayList主要方法有：add（），size（），remove（）；isEmpty（），clear（），indexof（）， lastindexof（）addAll（）；removeAll（） contain（），containsAll（），set（）。

Public <E> Iterator iterator();

Public boolean add(E e);

Public int size();

Public Object clone();

Public boolean remove(Object obj);

Public E remove(int index);

Public boolean isEmpty();

Public void clear();

Public int indexOf(Object elements);

Public int lastIndexOF(Object obj);

Public boolean contain(Object obj);

Public E set(E e,int index);

Public Boolean addAll([Collection](mk:@MSITStore:F:\\JDK1.6API_gr\\JDK1.6%20API帮助文档.CHM::/java/util/../../java/util/Collection.html" \o "java.util 中的接口)<? extends [E](mk:@MSITStore:F:\\JDK1.6API_gr\\JDK1.6%20API帮助文档.CHM::/java/util/../../java/util/ArrayList.html" \o "ArrayList 中的类型参数)> c)

Public Boolean addAll(int index,[Collection](mk:@MSITStore:F:\\JDK1.6API_gr\\JDK1.6%20API帮助文档.CHM::/java/util/../../java/util/Collection.html" \o "java.util 中的接口)<? extends [E](mk:@MSITStore:F:\\JDK1.6API_gr\\JDK1.6%20API帮助文档.CHM::/java/util/../../java/util/ArrayList.html" \o "ArrayList 中的类型参数)> c)

Public containAll([Collection](mk:@MSITStore:F:\\JDK1.6API_gr\\JDK1.6%20API帮助文档.CHM::/java/util/../../java/util/Collection.html" \o "java.util 中的接口)<? extends [E](mk:@MSITStore:F:\\JDK1.6API_gr\\JDK1.6%20API帮助文档.CHM::/java/util/../../java/util/ArrayList.html" \o "ArrayList 中的类型参数)> c))

Public Object[] toArray()

练1：

**import** java.util.Collections;

**import** java.util.Comparator;

**import** java.util.List;

**public** **class** ArrayListMethods {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

ArrayList<String> l = **new** ArrayList<String>();

**for** (**int** i = 0; i < 9; i++) {

l.add("a"+i);

}

ArrayList<String> li = **new** ArrayList<String>();

**for** (**int** i = 0; i < 9; i++) {

li.add(i+" ");

}

ArrayList<String> list = **new** ArrayList<String>();

//list.addAll(l);

li.addAll(1, l);

System.*out*.println(li);

System.*out*.println(li.containsAll(l));

System.*out*.println(list);

System.*out*.println(l.size());

l.set(4, "a900");

System.*out*.println(l);

l.add(4,"a8");

System.*out*.println(l);

System.*out*.println(l.get(5));

System.*out*.println(l.indexOf("a16"));

System.*out*.println(l.lastIndexOf("a16"));

l.clear();

System.out.println(l.size());

java.lang.Object[] obj = l.toArray();

**for**(java.lang.Object obj1 :obj){

System.*out*.println(obj1);

}

}

ArrayList list2 = (ArrayList) l.clone();

System.*out*.println(list2);

}

练2：用contain（Object obj），remove（Object，obj）时，注意重新equals（），hashcode（）方法。八大包装类，String类，date类重写了equals方法 toString方法。Hashcode方法。

**package** cn.com.ArrayListTest;

**public** **class** Teacher {

/\*\*

\* **@param** args

\*/

**private** String Tname;

**private** **int** Tage ;

**private** String TNo;

**public** Teacher(String tname, **int** tage, String tNo) {

**super**();

**this**.Tname = tname;

**this**.Tage = tage;

**this**.TNo = tNo;

}

**public** String getTname() {

**return** Tname;

}

**public** **void** setTname(String tname) {

Tname = tname;

}

**public** **int** getTage() {

**return** Tage;

}

**public** **void** setTage(**int** tage) {

Tage = tage;

}

**public** String getTNo() {

**return** TNo;

}

**public** **void** setTNo(String tNo) {

TNo = tNo;

}

**public** String toString(){

**return** Tname+"\t"+Tage+"\t"+TNo;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

Teacher other = (Teacher) obj;

**if** (TNo == **null**) {

**if** (other.TNo != **null**)

**return** **false**;

} **else** **if** (!TNo.equals(other.TNo))

**return** **false**;

**if** (Tage != other.Tage)

**return** **false**;

**if** (Tname == **null**) {

**if** (other.Tname != **null**)

**return** **false**;

} **else** **if** (!Tname.equals(other.Tname))

**return** **false**;

**return** **true**;

}

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + ((TNo == **null**) ? 0 : TNo.hashCode());

result = prime \* result + Tage;

result = prime \* result + ((Tname == **null**) ? 0 : Tname.hashCode());

**return** result;

}

}

**package** cn.com.ArrayListTest;

**import** java.util.ArrayList;

**import** java.util.Collection;

**public** **class** ArrayListTest {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Collection col = **new** ArrayList();//用父类的引用指向子类可以增加程序的灵活性

col.add("xiaoming");

col.add(**new** Integer(10));

col.add(**new** Teacher("李",23,"t45"));

System.*out*.println(col);

/\*

\*

\* 在集合中判断对象，是否相等是判断它们是否equals。在remove, contain 等方法中应注意重写equals(),hashCode()方法

\*/

col.remove("xiaoming");

col.remove("new Integer(10");

col.remove(**new** Teacher("李江 ",23,"t45"));

System.*out*.println(col.contains("xiaomi ng"));

System.*out*.println(col.contains(**new** Teacher("李江",23,"t45")));

}

}

4.LinkListed 和vector的用法与ArrayList 差不多就不一一举例。

5.自己写的Arraylist类的代码如下：

**package** cn.mycollection;

**import** java.util.Arrays;

**public** **class** MyArrayList <T>{

**private** T [] data;

**private** **int** capacity;

**private** **int** size;

**public** MyArrayList(**int** capacity){

**if**(capacity<0){

**try** {

**throw** **new** Exception();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**this**.capacity = capacity;

**this**.data = (T[])**new** Object[capacity];

}

**public** MyArrayList(){

**this**(10);

}

**public** **int** size(){

**return** **this**.size;

}

**public** **boolean** isEmpty(){

**if**(size==0){

**return** **false**;

}

**else**{

**return** **true**;

}

}

**public** **boolean** add( T t){

**boolean** bool = **false**;

**if**(size>=data.length-1){

T[] tempData = (T[]) **new** Object[size\*2];

System.*arraycopy*(data, 0, tempData, 0, data.length);

data = tempData;

}

data[size++] = t;

bool = **true**;

**return** bool;

}

**public** **void** add(**int** index,T elements ){

rangeCheck(index);

**if**(size>=data.length-1){

T[] tempData = (T[]) **new** Object[size\*2];

System.*arraycopy*(data, 0, tempData, 0, data.length);

data = tempData;

}

System.*arraycopy*(data, index, data, index+1, size-index);

data[index] = elements;

size++;

}

**public** T get(**int** index){

rangeCheck(index);

**return** data[index];

}

**public** T remove(**int** index){

rangeCheck(index);

**if**(size-index-1>=0){

System.*arraycopy*(data, index+1, data, index, size-index-1);

data[--size] = **null**;

}

**return** data[index];

}

**public** **boolean** remove(T obj){

**boolean** bool = **false**;

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

**if**(get(i).equals(obj)){

remove(i);

bool = **true**;

}

}

**return** bool;

}

**public** T set(**int** index,T elements){

rangeCheck(index);

**if**(size>=data.length-1){

T[] tempData = (T[]) **new** Object[size\*2];

System.*arraycopy*(data, 0, tempData, 0, data.length);

data = tempData;

}

remove(index);

data[index] = elements;

**return** data[index];

}

**public** **boolean** contains(T obj){

**boolean** bool = **false**;

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

**if**(get(i).equals(obj)){

bool = **true**;

}

}

**return** bool;

}

**public** Object [] toArray(){

**return** Arrays.*copyOf*(data, size);

}

**public** **void** clear(){

**if**(!isEmpty()){

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

remove(i);

}

}

}

**private** **void** rangeCheck(**int** index){

**if**(index<0||index>size){

**try** {

**throw** **new** Exception();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

MyArrayList<String> list = **new** MyArrayList<String> (3);

list.add("1df");

list.add("2sdfsdf");

list.add("3sdfsdf");

list.add("4sdfsdf");

list.add("5sdfsdf");

list.add("6sdfsdf");

list.add("7dsdf");

//list.get(8);

System.*out*.println(list.get(3));

System.*out*.println(list.size());

list.remove(6);

System.*out*.println(list.size());

System.*out*.println(list.get(2));

System.*out*.println(list.get(6));

list.add(2,"you");

System.*out*.println(list.get(2));

System.*out*.println(list.size());

System.*out*.println(list.remove("you"));

System.*out*.println(list.get(5));

System.*out*.println(list.set(2, "ni"));

System.*out*.println(list.get(2));

System.*out*.println(list.contains("ni"));

//System.out.println(list.remove("6sdfsdf"));

//list.add(2, "you");

//System.out.println(list.get(2));

System.*out*.println(Arrays.*toString*(list.toArray()));

list.clear();

System.*out*.println(list.isEmpty());

}

}

1. 自写LinkedList类如下（省写）：

**package** cn.linkedlist;

**public** **class** Node {

Node previous;

Object obj;

Node next;

**public** Node(){

}

**public** Node getPrevious() {

**return** previous;

}

**public** **void** setPrevious(Node previous) {

**this**.previous = previous;

}

**public** Object getObj() {

**return** obj;

}

**public** **void** setObj(Object obj) {

**this**.obj = obj;

}

**public** Node getNext() {

**return** next;

}

**public** **void** setNext(Node next) {

**this**.next = next;

}

}

**package** cn.linkedlist;

**public** **class** MyLinkedList {

**private** Node first;

**private** Node last;

**private** **int** size;

**public** **void** add(Object obj){

**if**(first==**null**){

Node n = **new** Node();

n.setPrevious(**null**);

n.setObj(obj);

n.setNext(**null**);

first = n;

last = n;

}

**else**{

Node n = **new** Node();

n.setPrevious(last);

n.setObj(obj);

n.setNext(**null**);

last.setNext(n);

last = n;

}

size++;

}

**public** **int** size(){

**return** size;

}

**public** Object get(**int** index){

**if**(first!=**null**){

Node temp = getNode(index);

**return** temp.obj;

}

**return** **null**;

}

**public** Object remove(**int** index){

Node temp = getNode(index);

**if**(temp!=**null**){

Node up = temp.previous;

Node down = temp.next;

up.next = down;

down.previous =up;

size--;

}

**return** temp.obj;

}

**public** Object add(**int** index,Object obj1){

Node newNode = **new** Node();

newNode.obj = obj1;

Node temp = getNode(index);

**if**(temp!=**null**){

Node up = temp.previous;

up.next =newNode;

newNode.previous = up;

newNode.next =temp;

temp.previous = newNode;

}

size++;

**return** newNode.obj;

}

**private** Node getNode(**int** index){

Node temp =**null**;

**if**(first!=**null**){

temp =first;

**for** (**int** i = 0; i < index; i++) {

temp = temp.next;

}

}

**return** temp;

}

**public** **static** **void** main(String[] args) {

MyLinkedList link = **new** MyLinkedList();

link.add("you");

link.add("my");

link.add(1,"me");

link.remove(1);

System.*out*.println(link.get(1));

System.*out*.println(link.size());

}

}

1. hashSet类（除与下标有关的方法比如：get（），indexOF（）lastIndexOF，其他常用方法与ArrayList类方法差不多）主要方法如下：set接口的子类同迭代器Iterator来取值。

Public <E> Iterator iterator();

Public boolean add(E e);

Public int size();

Public Object clone();

Public boolean remove(Object obj);

Public E remove(int index);

Public boolean isEmpty();

Public void clear();

HashSet练习一：//hashset是不能排序的，通过List子类的构造传一个hashset的引用，来产生一个list子类对象。再通过Collections.sort()来排序。或者通过treeset来自然排序（只有八大包装类和Strigng类）。Hashset去重的方法是重写hashcode和equals方法。hashcode和equals方法是检验集合元素是否相等的方法。

**package** cn.com.hashsettest;

**public** **class** Student {

//implements Comparable<Student>

**private** String sNO;

**private** String sName;

**private** **int** sAge;

**private** String sex;

**public** Student(String sNo,String sName,**int** sAge,String sex){

**this**.sNO = sNo;

**this**.sName = sName;

**this**.sAge = sAge;

**this**.sex = sex;

}

// public Student(String sNo){

// this.sNO = sNo;

// }

**public** String getsNO() {

**return** sNO;

}

**public** String getsName() {

**return** sName;

}

**public** **void** setsName(String sName) {

**this**.sName = sName;

}

**public** **int** getsAge() {

**return** sAge;

}

**public** **void** setsAge(**int** sAge) {

**this**.sAge = sAge;

}

**public** String getSex() {

**return** sex;

}

**public** **void** setSex(String sex) {

**this**.sex = sex;

}

**public** **void** setsNO(String sNO) {

**this**.sNO = sNO;

}

**public** String toString(){

**return** **this**.sNO+"\t"+**this**.sName+"\t"+**this**.sAge+"\t"+**this**.sex;

}

**public** **boolean** equals(Object obj){

**if**(obj==**null**){

**return** **false**;

}

**if**(obj **instanceof** Student){

Student s1 = (Student)obj;

**if**(s1.getsNO().equals(**this**.sNO)){

**return** **true**;

} **else** {

**return** **false**;

}

} **else**{

**return** **false**;

}

}

**public** **int** hashCode(){

**return** sName.hashCode();

}

//@Override

// public int compareTo(Student o) {

// // **TODO** Auto-generated method stub

// return this.sName.compareTo(o.sName);

// }

}

**package** cn.com.hashsettest;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.Collections;

**import** java.util.Comparator;

**import** java.util.HashSet;

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.Set;

**import** java.util.TreeSet;

**public** **class** HashSetTest {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Set<Student> set = **new** HashSet<Student>();

set.add(**new** Student("s234","jack",19,"男"));

set.add(**new** Student("s123","mary",20,"女"));

set.add(**new** Student("s234","jack",19,"男"));

set.add(**new** Student("s123456","kate",18,"男"));

set.add(**new** Student("s89","tom",17,"男"));

set.add(**new** Student("s234","jack",19,"男"));

List <Student> student = **new** ArrayList<Student>(set);

Collections.*sort*(student, **new** AgeComparator());

System.*out*.println(student);

// set.add(new Student("s123"));

// set.add(new Student("s234"));

// set.add(new Student("s123"));

// set.add(new Student("s890"));

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

**while**(it.hasNext()){

System.*out*.println(it.next());

}

}

}

**class** AgeComparator **implements** Comparator<Student>{

@Override

**public** **int** compare(Student o1, Student o2) {

// **TODO** Auto-generated method stub

**return** o1.getsAge()-o2.getsAge();

}

}

1. treeset用法与hashset差不多：

Hashset练习：

//hashset中（包装类和String类）的元素按按自然顺序存储。如果hashset中的元素要排序，可以通过treeset的构造方法传一个hashset对象。New一个treeset来排序。

**package** cn.Treeset;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.Collections;

**import** java.util.HashSet;

**import** java.util.TreeSet;

**public** **class** TreeSet1 {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

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

set.add("you");

set.add("in");

set.add("ta");

System.*out*.println(set);

TreeSet<String> ts = **new** TreeSet<String>(set);

ArrayList<String> list = **new** ArrayList<String>(set);

Collections.*sort*(list);

System.*out*.println(list);

// ts.add("tne");

// ts.add("beconde");

// ts.add("aaad");

**for**(String s1 : ts){

System.*out*.println(s1);

}

}

}

1. hashMap主要的方法如下：

//所有集合类可以通过iterator（）得到集合类的元素。也可以通过foreach即for（引用：集合对象）；

Public int size();

Public V remove(Object key);

Public boolean isEmpty();

Public boolean containKeys(Object key);

Public Object clone();

Public boolean containValue(Object value);

Public V put(K key,V values);

Public V get(Object key);

Public Set<K> keyset();

Public Collection<V> values();

hashMap练习：

**package** cn.hashmap;

**import** java.util.Collection;

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.Map;

**import** java.util.Set;

**public** **class** HashMap2 {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

HashMap<String, Student> map = **new** HashMap<String,Student>();

System.*out*.println(map.isEmpty());

map.put("mary", **new** Student("s031","mary",22));

map.put("kate", **new** Student("s045","kate",26));

System.*out*.println(map.size());

//map.clear();

System.*out*.println(map);

//map.remove("mary");

//System.out.println(map.containsValue(new Student("s031","mary",22)));

System.*out*.println();

Student s = map.get("mary");

System.*out*.println(s);

Map map1 = (Map) map.clone();

System.*out*.println(map.get("mary"));

//System.out.println(s.getStuAge()+"\n"+s.getStuName()+"\n"+s.getStuNo());

*printKeys*(map);

*printVaules*(map);

}

**public** **static** **void** printKeys(HashMap<String ,Student> map){

Set<String> keys = map.keySet();

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

**while**(it.hasNext()){

System.*out*.println(it.next());

}

}

**public** **static** **void** printVaules(HashMap<String ,Student> map){

Collection<Student> values = map.values();

**for**(Student s : values){

System.*out*.println(s.getStuAge()+"\n"+s.getStuName()+"\n"+s.getStuNo());

}

}

}

  public static void main(String[] args) {  
10.        Map<String,String> map=new HashMap<String,String>();   
11.        map.put("1", "张三");   
12.        map.put("2", "李四");   
13.        map.put("3", "王五");   
14.           
15.        System.out.println("方法一：");   
16.        Iterator iterator=map.entrySet().iterator();   
17.        while(iterator.hasNext()){         
18.           Map.Entry<String, String> entry= (Entry<String, String>) iterator.next();   
19.           System.out.println("key:"+entry.getKey()+" value"+entry.getValue());     
20.          }          
21.           
22.        System.out.println("方法二：");   
23.        for (Map.Entry<String, String> m : map.entrySet()) {   
24.              System.out.println("key:"+m.getKey()+" value"+m.getValue());   
25.          }    
26.      }

9：Collections类集合collection的工具类：

Collections练习1：

**package** cn.collection;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Iterator;

**public** **class** CollectionTest {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

ArrayList<String> list = **new** ArrayList<String>();

Collections.*addAll*(list, "xiayming","xiaoxuesheng","xiaoming");

**for**(String name:list){

System.*out*.println(name);

}

Collections.*reverse*(list);

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

**while**(it.hasNext()){

System.*out*.println(it.next());

}

Collections.*swap*(list, 0, 2);

**for**(String name:list){

System.*out*.println(name);

}

// for(String name :list){

// System.out.println(name);

// }

//collections是集合的帮助类。常用方法有reverse swap addAll min max sort等。

//sort()对应原始型和String而言，可以直接用。对于其他引用类型来说，要实现compareble接口。

}

}

Collections练习2：

**package** cn.collection;

//排序时可能要用到比较器。

**public** **class** Student **implements** Comparable<Student>{

/\*\*

\* **@param** args

\*/

**private** String stuNo;

**private** String name;

**private** **int** age;

**public** Student(){

}

**public** Student(String stuNo,String name,**int** age){

**this**.stuNo = stuNo;

**this**.name = name;

**this**.age = age;

}

**public** String getStuNo() {

**return** stuNo;

}

**public** **void** setStuNo(String stuNo) {

**this**.stuNo = stuNo;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

// public boolean equals(Object obj){

// if(obj != null){

// return false;

// }

// if(obj instanceof Student){

// Student s1 = (Student)obj;

// if(this.age==s1.age&&this.name.equals(s1.name)&&this.stuNo.equals(s1.stuNo)){

// return true;

// } else{

// return false;

// }

//

// } else{

// return false;

// }

// }

@Override

**public** **int** compareTo(Student o) { // **TODO** Auto-generated method stub

**return** **this**.name.compareTo(o.name);

}

**public** String toString(){

**return** **this**.stuNo+"\t" +**this**.name+"\t"+**this**.age;

}

**public** **boolean** equals(Object obj){

**if**(obj==**null**){

**return** **false**;

}

**if**(obj==**this**){

**return** **true**;

}

**if**(obj.getClass()==**this**.getClass()){

Student stu = (Student) obj;

**if**(stu.stuNo.equals(**this**.stuNo)&&stu.age==**this**.age&&stu.name.equals(**this**.name)){

**return** **true**;

} **else**{

**return** **false**;

}

} **else**{

**return** **false**;

}

}

}

**package** cn.collection;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Comparator;

**import** cn.collection.Student;

**public** **class** CollectionsSortTest {

/\*\*

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

ArrayList<Student> list = **new** ArrayList<Student>();

Student s1 =**new** Student("s34", "wck", 16);

Student s2 = **new** Student("s34", "wck", 16);

list.add(s1);

list.add(s2);

//System.out.println(s1.equals(s2));

list.add(**new** Student("s67", "mary", 20));

list.add(**new** Student("s34", "jack", 16));

Collections.*shuffle*(list);//随机交换元素的位置。

System.*out*.println(list);

System.*out*.println("排序前为：");

**for** (Student stu : list) {

System.*out*.println(stu);

}

System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

// Collections.sort(list, new Comparator<Student>(){

//

// @Override

// public int compare(Student o1, Student o2) {

// // **TODO** Auto-generated method stub

// return o1.getName().compareTo(o2.getName());

// }

//

// });

//Collections.sort(list,new ageComparator());

Collections.*sort*(list);

System.*out*.println("排序后的为：");

**for**(Student stu: list){

System.*out*.println(stu);

}

// Collections.reverse(list);

// System.out.println("倒置后为：");

// for(Student stu: list){

// System.out.println(stu);

// }

System.*out*.println();

System.*out*.println("最大元素为：");

System.*out*.println(Collections.*max*(list,**new** stuNameComparator()));

System.*out*.println("最小元素为：");

System.*out*.println(Collections.*min*(list,**new** stuNameComparator()));

System.*out*.println(Collections.*binarySearch*(list, **new** Student("s34", "jack", 16) , **new** ageComparator()));

ArrayList<Student> list1 = **new** ArrayList<Student>();

//l.add(new Student("f45","kav",20));

Collections.*copy*(list1, list);

System.*out*.println(list1);

// System.out.println("将的第一个元素和第三个元素交换后");

// Collections.swap(list, 0, 2);

// System.out.println(list);

//

// Collections.fill(list, new Student("r55","kaw",19));

// System.out.println(list);

}

}

// class AgeComparator implements Comparator<Student> {

//

// @Override

// public int compare(Student s1, Student s2) {

// // **TODO** Auto-generated method stub

// return s1.getAge() - s2.getAge();

// }

// }

// class stuNoComparator implements Comparator<Student> {

//

// @Override

// public int compare(Student s1, Student s2) {

// // **TODO** Auto-generated method stub

// return s1.getStuNo().compareTo(s2.getStuNo());

// }

//}

**class** stuNameComparator **implements** Comparator<Student>{

@Override

**public** **int** compare(Student o1, Student o2) {

// **TODO** Auto-generated method stub

**return** o1.getName().compareTo(o2.getName());

}

}

**class** ageComparator **implements** Comparator<Student>{

@Override

**public** **int** compare(Student o1, Student o2) {

// **TODO** Auto-generated method stub

**return** o1.getAge()-o2.getAge();

}

}

Hashmap原理

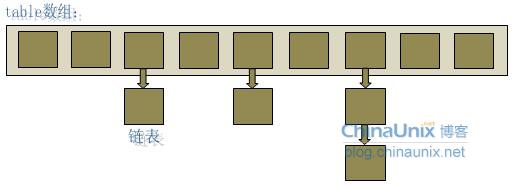
1.    HashMap概述

   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。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

2.    HashMap的数据结构

   在java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。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体。

　HashMap的底层主要是基于数组和链表来实现的，它之所以有相当快的查询速度主要是因为它是通过计算散列码来决定存储的位置。HashMap中主要是通过key的hashCode来计算hash值的，只要hashCode相同，计算出来的hash值就一样。如果存储的对象对多了，就有可能不同的对象所算出来的hash值是相同的，这就出现了所谓的hash冲突。学过数据结构的同学都知道，解决hash冲突的方法有很多，HashMap底层是通过链表来解决hash冲突的。

[](http://blog.chinaunix.net/attachment/201203/22/11775320_1332399870gIGw.jpg" \t "http://blog.chinaunix.net/_blank)

3.            HashMap的存取   
    HashMap的功能是通过“键(key)”能够快速的找到“值”。下面我们分析下HashMap存数据的基本流程：   
    1、 当调用put(key,value)时，首先获取key的hashcode，int hash = key.hashCode();   
    2、 再把hash通过一下运算得到一个int h.   
hash ^= (hash >>> 20) ^ (hash >>> 12);   
int h = hash ^ (hash >>> 7) ^ (hash >>> 4);   
为什么要经过这样的运算呢？这就是HashMap的高明之处。先看个例子，一个十进制数32768(二进制1000 0000 0000 0000)，经过上述公式运算之后的结果是35080(二进制1000 1001 0000 1000)。看出来了吗？或许这样还看不出什么，再举个数字61440(二进制1111 0000 0000 0000)，运算结果是65263(二进制1111 1110 1110 1111)，现在应该很明显了，它的目的是让“1”变的均匀一点，散列的本意就是要尽量均匀分布。那这样有什么意义呢？看第3步。   
    3、 得到h之后，把h与HashMap的承载量（HashMap的默认承载量length是16，可以自动变长。在构造HashMap的时候也可以指定一个长 度。这个承载量就是上图所描述的数组的长度。）进行逻辑与运算，即 h & (length-1)，这样得到的结果就是一个比length小的正数，我们把这个值叫做index。其实这个index就是索引将要插入的值在数组中的 位置。第2步那个算法的意义就是希望能够得出均匀的index，这是HashTable的改进，HashTable中的算法只是把key的 hashcode与length相除取余，即hash % length，这样有可能会造成index分布不均匀。还有一点需要说明，HashMap的键可以为null，它的值是放在数组的第一个位置。   
    4、 我们用table[index]表示已经找到的元素需要存储的位置。先判断该位置上有没有元素（这个元素是HashMap内部定义的一个类Entity， 基本结构它包含三个类，key，value和指向下一个Entity的next）,没有的话就创建一个Entity<k,v style="word-wrap: break-word;">对象，在 table[index]位置上插入，这样插入结束；如果有的话，通过链表的遍历方式去逐个遍历，看看有没有已经存在的key，有的话用新的value替 换老的value；如果没有，则在table[index]插入该Entity，把原来在table[index]位置上的Entity赋值给新的 Entity的next，这样插入结束。   
总结：keyàhashcodeàhàindexà遍历链表à插入

4.            扩展问题

要同时复写equals方法和hashCode方法。

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），

再回头看看前面提到的为什么覆盖了equals方法之后一定要覆盖hashCode方法，很简单，比如，String a = new String(“abc”);String b = new String(“abc”);如果不覆盖hashCode的话，那么a和b的hashCode就会不同，把这两个类当做key存到HashMap中的话就 会出现问题，就会和key的唯一性相矛盾。

一、HashMap概述

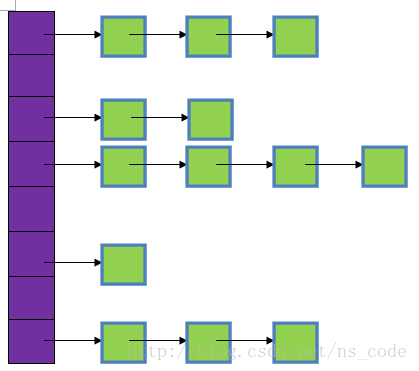
　　HashMap基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了不同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

　　值得注意的是HashMap不是线程安全的，如果想要线程安全的HashMap，可以通过Collections类的静态方法synchronizedMap获得线程安全的HashMap。

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

二、HashMap的数据结构

　　HashMap的底层主要是基于数组和链表来实现的，它之所以有相当快的查询速度主要是因为它是通过计算散列码来决定存储的位置。HashMap中主要是通过key的hashCode来计算hash值的，只要hashCode相同，计算出来的hash值就一样。如果存储的对象对多了，就有可能不同的对象所算出来的hash值是相同的，这就出现了所谓的hash冲突。学过数据结构的同学都知道，解决hash冲突的方法有很多，HashMap底层是通过链表来解决hash冲突的。



 图中，紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中。

我们看看HashMap中Entry类的代码：

[IMG_257](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

/\*\* Entry是单向链表。

\* 它是 “HashMap链式存储法”对应的链表。

\*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;

// 指向下一个节点

Entry<K,V> next;

final int hash;

// 构造函数。

// 输入参数包括"哈希值(h)", "键(k)", "值(v)", "下一节点(n)"

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;

}

// 判断两个Entry是否相等

// 若两个Entry的“key”和“value”都相等，则返回true。

// 否则，返回false public final boolean equals(Object o) {

if (!(o instanceof Map.Entry))

return false;

Map.Entry e = (Map.Entry)o;

Object k1 = getKey();

Object k2 = e.getKey();

if (k1 == k2 || (k1 != null && k1.equals(k2))) {

Object v1 = getValue();

Object v2 = e.getValue();

if (v1 == v2 || (v1 != null && v1.equals(v2)))

return true;

}

return false;

}

// 实现hashCode()

public final int hashCode() {

return (key==null ? 0 : key.hashCode()) ^

(value==null ? 0 : value.hashCode());

}

public final String toString() {

return getKey() + "=" + getValue();

}

// 当向HashMap中添加元素时，绘调用recordAccess()。

// 这里不做任何处理

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

}

// 当从HashMap中删除元素时，绘调用recordRemoval()。

// 这里不做任何处理

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

}

}

[IMG_258](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

HashMap其实就是一个Entry数组，Entry对象中包含了键和值，其中next也是一个Entry对象，它就是用来处理hash冲突的，形成一个链表。

三、HashMap源码分析

1、关键属性

　　先看看HashMap类中的一些关键属性：

[IMG_259](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

1 transient Entry[] table;//存储元素的实体数组2 3 transient int size;//存放元素的个数4 5 int threshold; //临界值 当实际大小超过临界值时，会进行扩容threshold = 加载因子\*容量6 7 final float loadFactor; //加载因子8 9 transient int modCount;//被修改的次数

[IMG_260](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

其中loadFactor加载因子是表示Hsah表中元素的填满的程度.

若:加载因子越大,填满的元素越多,好处是,空间利用率高了,但:冲突的机会加大了.链表长度会越来越长,查找效率降低。

反之,加载因子越小,填满的元素越少,好处是:冲突的机会减小了,但:空间浪费多了.表中的数据将过于稀疏（很多空间还没用，就开始扩容了）

冲突的机会越大,则查找的成本越高.

因此,必须在 "冲突的机会"与"空间利用率"之间寻找一种平衡与折衷. 这种平衡与折衷本质上是数据结构中有名的"时-空"矛盾的平衡与折衷.

　　如果机器内存足够，并且想要提高查询速度的话可以将加载因子设置小一点；相反如果机器内存紧张，并且对查询速度没有什么要求的话可以将加载因子设置大一点。不过一般我们都不用去设置它，让它取默认值0.75就好了。

2、构造方法

下面看看HashMap的几个构造方法：

[IMG_261](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

public HashMap(int initialCapacity, float loadFactor) {

2 //确保数字合法

3 if (initialCapacity < 0)

4 throw new IllegalArgumentException("Illegal initial capacity: " +

5 initialCapacity);

6 if (initialCapacity > MAXIMUM\_CAPACITY)

7 initialCapacity = MAXIMUM\_CAPACITY;

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

9 throw new IllegalArgumentException("Illegal load factor: " +10 loadFactor);11 12 // Find a power of 2 >= initialCapacity13 int capacity = 1; //初始容量14 while (capacity < initialCapacity) //确保容量为2的n次幂，使capacity为大于initialCapacity的最小的2的n次幂15 capacity <<= 1;16 17 this.loadFactor = loadFactor;18 threshold = (int)(capacity \* loadFactor);19 table = new Entry[capacity];20 init();21 }22 23 public HashMap(int initialCapacity) {24 this(initialCapacity, DEFAULT\_LOAD\_FACTOR);25 }26 27 public HashMap() {28 this.loadFactor = DEFAULT\_LOAD\_FACTOR;29 threshold = (int)(DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR);30 table = new Entry[DEFAULT\_INITIAL\_CAPACITY];31 init();32 }

[IMG_262](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

我们可以看到在构造HashMap的时候如果我们指定了加载因子和初始容量的话就调用第一个构造方法，否则的话就是用默认的。默认初始容量为16，默认加载因子为0.75。我们可以看到上面代码中13-15行，这段代码的作用是确保容量为2的n次幂，使capacity为大于initialCapacity的最小的2的n次幂，至于为什么要把容量设置为2的n次幂，我们等下再看。

重点分析下HashMap中用的最多的两个方法put和get

 3、存储数据

　　下面看看HashMap存储数据的过程是怎样的，首先看看HashMap的put方法：

[IMG_263](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

public V put(K key, V value) {

// 若“key为null”，则将该键值对添加到table[0]中。

if (key == null)

return putForNullKey(value);

// 若“key不为null”，则计算该key的哈希值，然后将其添加到该哈希值对应的链表中。

int hash = hash(key.hashCode());

//搜索指定hash值在对应table中的索引

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

// 循环遍历Entry数组,若“该key”对应的键值对已经存在，则用新的value取代旧的value。然后退出！

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

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) { //如果key相同则覆盖并返回旧值

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

//修改次数+1

modCount++;

//将key-value添加到table[i]处 addEntry(hash, key, value, i);

return null;

}

[IMG_264](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

上面程序中用到了一个重要的内部接口：Map.Entry，每个 Map.Entry 其实就是一个 key-value 对。从上面程序中可以看出：当系统决定存储 HashMap 中的 key-value 对时，完全没有考虑 Entry 中的 value，仅仅只是根据 key 来计算并决定每个 Entry 的存储位置。这也说明了前面的结论：我们完全可以把 Map 集合中的 value 当成 key 的附属，当系统决定了 key 的存储位置之后，value 随之保存在那里即可。

我们慢慢的来分析这个函数，第2和3行的作用就是处理key值为null的情况，我们看看putForNullKey(value)方法：

[IMG_265](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

1 private V putForNullKey(V value) {

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

3 if (e.key == null) { //如果有key为null的对象存在，则覆盖掉

4 V oldValue = e.value;

5 e.value = value;

6 e.recordAccess(this);

7 return oldValue;

8 }

9 }10 modCount++;11 addEntry(0, null, value, 0); //如果键为null的话，则hash值为012 return null;13 }

[IMG_266](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

注意：如果key为null的话，hash值为0，对象存储在数组中索引为0的位置。即table[0]

我们再回去看看put方法中第4行，它是通过key的hashCode值计算hash码，下面是计算hash码的函数：

[IMG_267](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

1 //计算hash值的方法 通过键的hashCode来计算2 static int hash(int h) {3 // This function ensures that hashCodes that differ only by4 // constant multiples at each bit position have a bounded5 // number of collisions (approximately 8 at default load factor).6 h ^= (h >>> 20) ^ (h >>> 12);7 return h ^ (h >>> 7) ^ (h >>> 4);8 }

[IMG_268](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

得到hash码之后就会通过hash码去计算出应该存储在数组中的索引，计算索引的函数如下：

1 static int indexFor(int h, int length) { //根据hash值和数组长度算出索引值2 return h & (length-1); //这里不能随便算取，用hash&(length-1)是有原因的，这样可以确保算出来的索引是在数组大小范围内，不会超出3 }

这个我们要重点说下，我们一般对哈希表的散列很自然地会想到用hash值对length取模（即除法散列法），Hashtable中也是这样实现的，这种方法基本能保证元素在哈希表中散列的比较均匀，但取模会用到除法运算，效率很低，HashMap中则通过h&(length-1)的方法来代替取模，同样实现了均匀的散列，但效率要高很多，这也是HashMap对Hashtable的一个改进。

    接下来，我们分析下为什么哈希表的容量一定要是2的整数次幂。首先，length为2的整数次幂的话，h&(length-1)就相当于对length取模，这样便保证了散列的均匀，同时也提升了效率；其次，length为2的整数次幂的话，为偶数，这样length-1为奇数，奇数的最后一位是1，这样便保证了h&(length-1)的最后一位可能为0，也可能为1（这取决于h的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀性，而如果length为奇数的话，很明显length-1为偶数，它的最后一位是0，这样h&(length-1)的最后一位肯定为0，即只能为偶数，这样任何hash值都只会被散列到数组的偶数下标位置上，这便浪费了近一半的空间，因此，length取2的整数次幂，是为了使不同hash值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列。

　　这看上去很简单，其实比较有玄机的，我们举个例子来说明：

　　假设数组长度分别为15和16，优化后的hash码分别为8和9，那么&运算后的结果如下：

h & (table.length-1) hash table.length-1

8 & (15-1)： 0100 & 1110 = 0100

9 & (15-1)： 0101 & 1110 = 0100

-----------------------------------------------------------------------------------------------------------------------

8 & (16-1)： 0100 & 1111 = 0100

9 & (16-1)： 0101 & 1111 = 0101

从上面的例子中可以看出：当它们和15-1（1110）“与”的时候，产生了相同的结果，也就是说它们会定位到数组中的同一个位置上去，这就产生了碰撞，8和9会被放到数组中的同一个位置上形成链表，那么查询的时候就需要遍历这个链 表，得到8或者9，这样就降低了查询的效率。同时，我们也可以发现，当数组长度为15的时候，hash值会与15-1（1110）进行“与”，那么 最后一位永远是0，而0001，0011，0101，1001，1011，0111，1101这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！而当数组长度为16时，即为2的n次方时，2n-1得到的二进制数的每个位上的值都为1，这使得在低位上&时，得到的和原hash的低位相同，加之hash(int h)方法对key的hashCode的进一步优化，加入了高位计算，就使得只有相同的hash值的两个值才会被放到数组中的同一个位置上形成链表。

　　 所以说，当数组长度为2的n次幂的时候，不同的key算得得index相同的几率较小，那么数据在数组上分布就比较均匀，也就是说碰撞的几率小，相对的，查询的时候就不用遍历某个位置上的链表，这样查询效率也就较高了。

       根据上面 put 方法的源代码可以看出，当程序试图将一个key-value对放入HashMap中时，程序首先根据该 key 的 hashCode() 返回值决定该 Entry 的存储位置：如果两个 Entry 的 key 的 hashCode() 返回值相同，那它们的存储位置相同。如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖集合中原有 Entry 的 value，但key不会覆盖。如果这两个 Entry 的 key 通过 equals 比较返回 false，新添加的 Entry 将与集合中原有 Entry 形成 Entry 链，而且新添加的 Entry 位于 Entry 链的头部——具体说明继续看 addEntry() 方法的说明。

1 void addEntry(int hash, K key, V value, int bucketIndex) {2 Entry<K,V> e = table[bucketIndex]; //如果要加入的位置有值，将该位置原先的值设置为新entry的next,也就是新entry链表的下一个节点3 table[bucketIndex] = new Entry<>(hash, key, value, e);4 if (size++ >= threshold) //如果大于临界值就扩容5 resize(2 \* table.length); //以2的倍数扩容6 }

参数bucketIndex就是indexFor函数计算出来的索引值，第2行代码是取得数组中索引为bucketIndex的Entry对象，第3行就是用hash、key、value构建一个新的Entry对象放到索引为bucketIndex的位置，并且将该位置原先的对象设置为新对象的next构成链表。

　　第4行和第5行就是判断put后size是否达到了临界值threshold，如果达到了临界值就要进行扩容，HashMap扩容是扩为原来的两倍。

4、调整大小

resize()方法如下：

 重新调整HashMap的大小，newCapacity是调整后的单位

[IMG_269](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

1 void resize(int newCapacity) {

2 Entry[] oldTable = table;

3 int oldCapacity = oldTable.length;

4 if (oldCapacity == MAXIMUM\_CAPACITY) {

5 threshold = Integer.MAX\_VALUE;

6 return;

7 }

8

9 Entry[] newTable = new Entry[newCapacity];10 transfer(newTable);//用来将原先table的元素全部移到newTable里面11 table = newTable; //再将newTable赋值给table12 threshold = (int)(newCapacity \* loadFactor);//重新计算临界值13 }

[IMG_270](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

新建了一个HashMap的底层数组，上面代码中第10行为调用transfer方法，将HashMap的全部元素添加到新的HashMap中,并重新计算元素在新的数组中的索引位置

当HashMap中的元素越来越多的时候，hash冲突的几率也就越来越高，因为数组的长度是固定的。所以为了提高查询的效率，就要对HashMap的数组进行扩容，数组扩容这个操作也会出现在ArrayList中，这是一个常用的操作，而在HashMap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。

   那么HashMap什么时候进行扩容呢？当HashMap中的元素个数超过数组大小\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为16，那么当HashMap中元素个数超过16\*0.75=12的时候，就把数组的大小扩展为 2\*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置，扩容是需要进行数组复制的，复制数组是非常消耗性能的操作，所以如果我们已经预知HashMap中元素的个数，那么预设元素的个数能够有效的提高HashMap的性能。

 5、数据读取

[IMG_271](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

1.public V get(Object key) { 2. if (key == null) 3. return getForNullKey(); 4. int hash = hash(key.hashCode()); 5. for (Entry<K,V> e = table[indexFor(hash, table.length)]; 6. e != null; 7. e = e.next) { 8. Object k; 9. if (e.hash == hash && ((k = e.key) == key || key.equals(k))) 10. return e.value; 11. } 12. return null; 13.}

[IMG_272](http://www.cnblogs.com/ITtangtang/p/javascript:void(0);" \o "复制代码)

有了上面存储时的hash算法作为基础，理解起来这段代码就很容易了。从上面的源代码中可以看出：从HashMap中get元素时，首先计算key的hashCode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。

6、HashMap的性能参数：

   HashMap 包含如下几个构造器：

   HashMap()：构建一个初始容量为 16，负载因子为 0.75 的 HashMap。

   HashMap(int initialCapacity)：构建一个初始容量为 initialCapacity，负载因子为 0.75 的 HashMap。

   HashMap(int initialCapacity, float loadFactor)：以指定初始容量、指定的负载因子创建一个 HashMap。

   HashMap的基础构造器HashMap(int initialCapacity, float loadFactor)带有两个参数，它们是初始容量initialCapacity和加载因子loadFactor。

   initialCapacity：HashMap的最大容量，即为底层数组的长度。

   loadFactor：负载因子loadFactor定义为：散列表的实际元素数目(n)/ 散列表的容量(m)。

   负载因子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。

   HashMap的实现中，通过threshold字段来判断HashMap的最大容量：

threshold = (int)(capacity \* loadFactor);

   结合负载因子的定义公式可知，threshold就是在此loadFactor和capacity对应下允许的最大元素数目，超过这个数目就重新resize，以降低实际的负载因子。默认的的负载因子0.75是对空间和时间效率的一个平衡选择。当容量超出此最大容量时， resize后的HashMap容量是容量的两倍：

Set集合

 1.HashSet  
  只去重复, 没有顺序  
  HashSet的add方法会调用hashCode和equals, 所以存储在HashSet中的对象需要重写这两个方法.

 2.TreeSet  
  去重复, 并且可以按照某种顺序排序  
  TreeSet的add方法会将对象转为Comparable, 然后调用compareTo方法, 所以存储在TreeSet中的对象必须实现Comparable, 重写compareTo方法

 3.LinkedHashSet  
  HashSet的子类, 去重复, 并且保留存储顺序

 4.迭代方式  
  使用迭代器Iterator  
  使用增强for循环

 HashSet原理:  
    
  /\*  
 \* 如果我们希望一个集合有去重复的功能, 可以在它的add方法中检查要添加的对象在集合中是否存在.   
 \* 迭代集合中每个元素, 和要添加的比较, 如果相同, 就不存.  
 \*   
 \* 如果使用上述方法, 当集合元素特别多的时候, 效率会很低.

 \*例如: 集合中有1万个元素, 当存储下一个的时候, 需要和前面1万个都比较, 效率较低.  
 \*   
 \* HashSet的工作原理:  
 \*   每次存储对象的时候, 调用对象的hashCode()方法, 计算一个哈希值. 在集合中查找是否包含哈希值相同的元素.  
 \*    如果没有哈希值相同元素, 直接存入.  
 \*    如果有哈希值相同的元素, 逐个使用equals()方法比较.  
 \*     比较结果全为false就存入.  
 \*     如果比较结果有true则不存.  
 \*   
 \* 如何将自定义类对象存入HashSet进行去重复  
 \*   类中必须重写hashCode()方法和equals()方法  
 \*   equals()方法中比较所有属性  
 \*   hashCode()方法要保证属性相同的对象返回值相同, 属性不同的对象尽量不同

   TreeSet原理:  
     
     
   /\*  
 \* TreeSet存储对象的时候, 可以排序, 但是需要指定排序的**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/zhangwenjun32/article/details/_blank)**  
 \*   
 \* Integer能排序(有默认顺序), String能排序(有默认顺序), 自定义的类存储的时候出现异常(没有顺序)  
 \*   
 \* 如果想把自定义类的对象存入TreeSet进行排序, 那么必须实现Comparable接口  
 \*   在类上implement Comparable  
 \*   重写compareTo()方法  
 \*   在方法内定义比较算法, 根据大小关系, 返回正数负数或零  
 \*   在使用TreeSet存储对象的时候, add()方法内部就会自动调用compareTo()方法进行比较, 根据比较结果使用二叉树形式进行存储

 之所以会定义栈和队列这样的数据结构  
    是因为他们有两大特性：  
    第一： 他们可以保存程序运行路径中各个点的信息，以便用于回溯操作或其他需要访问已经访问过的节点信息的操作。  
    比如： 栈用于解决迷宫问题，就是用到了若线路不通，需要回溯到已访问过的结点，从那个结点再做一次与这次路径不同的选择。  
      
    第二： 先进后出 和 先进先出的 次序  
  
           先进后出次序 其实就是一种将序列反序操作的次序  
           先进先出次序 其实就是一种将序列顺序操作的次序  
             
    比如： 利用栈的先进后出可以解决进制转化问题 ，即：先将个位余数进栈，再将十位余数进栈，然后百位，千位 等 ，这样出栈的时候顺序就成了反序出栈，即：先千位，百位，然后十位，最后个位。  
  
  
总结：  
  
           从保存信息角度来看 定义栈和队列数据结构是为了保存遍历所经过的路径，亦即保存遍历过程中的每一个结点的信息，而不仅仅只是终结点的信息。  而之所以要用栈或队列保存每个遍历节点信息，是因为在确定下一次的遍历节点时，可能需要利用已经遍历过的节点的信息，比如回溯操作  
  
           从次序角度来看，定义栈是为了对序列实行一种反序操作  
                          定义队列 是为了对序列实行一种顺序操作

Stack解决匹配对称问题

号匹配问题是指要匹配一个字符串的左，右括号：  
括号问题可以用来解决C语言中的“{”和“}”的匹配问题，可以观察到，如果从左至右扫描一个字符串，

那么每个右括号将于最近遇到的那个未匹配的左括号相匹配，在从左至右的扫描工程中把所遇到的左括号

存放到堆栈内，每当遇到一个右括号时，就将它与栈顶的左括号（如果存在）相匹配，同时从栈顶删除该

左括号

**public** **static** **boolean** match(String matchString) {

**boolean** bool = **false**;

Stack<Character> stack = **new** Stack<Character>();

**for** (**int** i = 0; i < matchString.length(); i++) {

**if** (matchString.charAt(i) == '{' || matchString.charAt(i) == '[') {

System.*out*.println(stack.push(matchString.charAt(i)));

}

**if** (matchString.charAt(i) == ']' && stack.peek() == '[') {

stack.pop();

}

**if** (matchString.charAt(i) == '}' && stack.peek() == '{') {

System.*out*.println(stack.pop());

}

}

**if** (stack.isEmpty()) {

bool = **true**;

}

System.*out*.println(stack.size());

**return** bool;

}

TreeMap 的实现就是红黑树数据结构，也就说是一棵自平衡的排序二叉树，这样就可以保证当需要快速检索指定节点。

TreeSet 和 TreeMap 的关系

为了让大家了解 TreeMap 和 TreeSet 之间的关系，下面先看 TreeSet 类的部分源代码：

|  |
| --- |
| public class TreeSet<E> extends AbstractSet<E>  implements NavigableSet<E>, Cloneable, java.io.Serializable  {  // 使用 NavigableMap 的 key 来保存 Set 集合的元素  private transient NavigableMap<E,Object> m;  // 使用一个 PRESENT 作为 Map 集合的所有 value。  private static final Object PRESENT = new Object();  // 包访问权限的构造器，以指定的 NavigableMap 对象创建 Set 集合  TreeSet(NavigableMap<E,Object> m)  {  this.m = m;  }  public TreeSet() // ①  {  // 以自然排序方式创建一个新的 TreeMap，  // 根据该 TreeSet 创建一个 TreeSet，  // 使用该 TreeMap 的 key 来保存 Set 集合的元素  this(new TreeMap<E,Object>());  }  public TreeSet(Comparator<? super E> comparator) // ②  {  // 以定制排序方式创建一个新的 TreeMap，  // 根据该 TreeSet 创建一个 TreeSet，  // 使用该 TreeMap 的 key 来保存 Set 集合的元素  this(new TreeMap<E,Object>(comparator));  }  public TreeSet(Collection<? extends E> c)  {  // 调用①号构造器创建一个 TreeSet，底层以 TreeMap 保存集合元素  this();  // 向 TreeSet 中添加 Collection 集合 c 里的所有元素  addAll(c);  }  public TreeSet(SortedSet<E> s)  {  // 调用②号构造器创建一个 TreeSet，底层以 TreeMap 保存集合元素  this(s.comparator());  // 向 TreeSet 中添加 SortedSet 集合 s 里的所有元素  addAll(s);  }  //TreeSet 的其他方法都只是直接调用 TreeMap 的方法来提供实现  ...  public boolean addAll(Collection<? extends E> c)  {  if (m.size() == 0 && c.size() > 0 &&  c instanceof SortedSet &&  m instanceof TreeMap)  {  // 把 c 集合强制转换为 SortedSet 集合  SortedSet<? extends E> set = (SortedSet<? extends E>) c;  // 把 m 集合强制转换为 TreeMap 集合  TreeMap<E,Object> map = (TreeMap<E, Object>) m;  Comparator<? super E> cc = (Comparator<? super E>) set.comparator();  Comparator<? super E> mc = map.comparator();  // 如果 cc 和 mc 两个 Comparator 相等  if (cc == mc || (cc != null && cc.equals(mc)))  {  // 把 Collection 中所有元素添加成 TreeMap 集合的 key  map.addAllForTreeSet(set, PRESENT);  return true;  }  }  // 直接调用父类的 addAll() 方法来实现  return super.addAll(c);  }  ...  } |

从上面代码可以看出，TreeSet 的 ① 号、② 号构造器的都是新建一个 TreeMap 作为实际存储 Set 元素的容器，而另外 2 个构造器则分别依赖于 ① 号和 ② 号构造器，由此可见，TreeSet 底层实际使用的存储容器就是 TreeMap。

与 HashSet 完全类似的是，TreeSet 里绝大部分方法都是直接调用 TreeMap 的方法来实现的，这一点读者可以自行参阅 TreeSet 的源代码，此处就不再给出了。

对于 TreeMap 而言，它采用一种被称为“红黑树”的排序二叉树来保存 Map 中每个 Entry —— 每个 Entry 都被当成“红黑树”的一个节点对待。例如对于如下程序而言：

|  |
| --- |
| public class TreeMapTest  {  public static void main(String[] args)  {  TreeMap<String , Double> map =  new TreeMap<String , Double>();  map.put("ccc" , 89.0);  map.put("aaa" , 80.0);  map.put("zzz" , 80.0);  map.put("bbb" , 89.0);  System.out.println(map);  }  } |

当程序执行 map.put("ccc" , 89.0); 时，系统将直接把 "ccc"-89.0 这个 Entry 放入 Map 中，这个 Entry 就是该“红黑树”的根节点。接着程序执行 map.put("aaa" , 80.0); 时，程序会将 "aaa"-80.0 作为新节点添加到已有的红黑树中。

以后每向 TreeMap 中放入一个 key-value 对，系统都需要将该 Entry 当成一个新节点，添加成已有红黑树中，通过这种方式就可保证 TreeMap 中所有 key 总是由小到大地排列。例如我们输出上面程序，将看到如下结果（所有 key 由小到大地排列）：

|  |
| --- |
| {aaa=80.0, bbb=89.0, ccc=89.0, zzz=80.0} |

TreeMap 的添加节点

## **红黑树**

红黑树是一种自平衡排序二叉树，树中每个节点的值，都大于或等于在它的左子树中的所有节点的值，并且小于或等于在它的右子树中的所有节点的值，这确保红黑树运行时可以快速地在树中查找和定位的所需节点。

对于 TreeMap 而言，由于它底层采用一棵“红黑树”来保存集合中的 Entry，这意味这 TreeMap 添加元素、取出元素的性能都比 HashMap 低：当 TreeMap 添加元素时，需要通过循环找到新增 Entry 的插入位置，因此比较耗性能；当从 TreeMap 中取出元素时，需要通过循环才能找到合适的 Entry，也比较耗性能。但 TreeMap、TreeSet 比 HashMap、HashSet 的优势在于：TreeMap 中的所有 Entry 总是按 key 根据指定排序规则保持有序状态，TreeSet 中所有元素总是根据指定排序规则保持有序状态。

为了理解 TreeMap 的底层实现，必须先介绍排序二叉树和红黑树这两种数据结构。其中红黑树又是一种特殊的排序二叉树。

排序二叉树是一种特殊结构的二叉树，可以非常方便地对树中所有节点进行排序和检索。

排序二叉树要么是一棵空二叉树，要么是具有下列性质的二叉树：

* 若它的左子树不空，则左子树上所有节点的值均小于它的根节点的值；
* 若它的右子树不空，则右子树上所有节点的值均大于它的根节点的值；
* 它的左、右子树也分别为排序二叉树。

图 1 显示了一棵排序二叉树：

**图 1. 排序二叉树**

对排序二叉树，若按中序遍历就可以得到由小到大的有序序列。如图 1 所示二叉树，中序遍历得：

|  |
| --- |
| {2，3，4，8，9，9，10，13，15，18} |

创建排序二叉树的步骤，也就是不断地向排序二叉树添加节点的过程，向排序二叉树添加节点的步骤如下：

1. 以根节点当前节点开始搜索。
2. 拿新节点的值和当前节点的值比较。
3. 如果新节点的值更大，则以当前节点的右子节点作为新的当前节点；如果新节点的值更小，则以当前节点的左子节点作为新的当前节点。
4. 重复 2、3 两个步骤，直到搜索到合适的叶子节点为止。
5. 将新节点添加为第 4 步找到的叶子节点的子节点；如果新节点更大，则添加为右子节点；否则添加为左子节点。

掌握上面理论之后，下面我们来分析 TreeMap 添加节点（TreeMap 中使用 Entry 内部类代表节点）的实现，TreeMap 集合的 put(K key, V value) 方法实现了将 Entry 放入排序二叉树中，下面是该方法的源代码：

|  |
| --- |
| public V put(K key, V value)  {  // 先以 t 保存链表的 root 节点  Entry<K,V> t = root;  // 如果 t==null，表明是一个空链表，即该 TreeMap 里没有任何 Entry  if (t == null)  {  // 将新的 key-value 创建一个 Entry，并将该 Entry 作为 root  root = new Entry<K,V>(key, value, null);  // 设置该 Map 集合的 size 为 1，代表包含一个 Entry  size = 1;  // 记录修改次数为 1  modCount++;  return null;  }  int cmp;  Entry<K,V> parent;  Comparator<? super K> cpr = comparator;  // 如果比较器 cpr 不为 null，即表明采用定制排序  if (cpr != null)  {  do {  // 使用 parent 上次循环后的 t 所引用的 Entry  parent = t;  // 拿新插入 key 和 t 的 key 进行比较  cmp = cpr.compare(key, t.key);  // 如果新插入的 key 小于 t 的 key，t 等于 t 的左边节点  if (cmp < 0)  t = t.left;  // 如果新插入的 key 大于 t 的 key，t 等于 t 的右边节点  else if (cmp > 0)  t = t.right;  // 如果两个 key 相等，新的 value 覆盖原有的 value，  // 并返回原有的 value  else  return t.setValue(value);  } while (t != null);  }  else  {  if (key == null)  throw new NullPointerException();  Comparable<? super K> k = (Comparable<? super K>) key;  do {  // 使用 parent 上次循环后的 t 所引用的 Entry  parent = t;  // 拿新插入 key 和 t 的 key 进行比较  cmp = k.compareTo(t.key);  // 如果新插入的 key 小于 t 的 key，t 等于 t 的左边节点  if (cmp < 0)  t = t.left;  // 如果新插入的 key 大于 t 的 key，t 等于 t 的右边节点  else if (cmp > 0)  t = t.right;  // 如果两个 key 相等，新的 value 覆盖原有的 value，  // 并返回原有的 value  else  return t.setValue(value);  } while (t != null);  }  // 将新插入的节点作为 parent 节点的子节点  Entry<K,V> e = new Entry<K,V>(key, value, parent);  // 如果新插入 key 小于 parent 的 key，则 e 作为 parent 的左子节点  if (cmp < 0)  parent.left = e;  // 如果新插入 key 小于 parent 的 key，则 e 作为 parent 的右子节点  else  parent.right = e;  // 修复红黑树  fixAfterInsertion(e); // ①  size++;  modCount++;  return null;  } |

上面程序中粗体字代码就是实现“排序二叉树”的关键算法，每当程序希望添加新节点时：系统总是从树的根节点开始比较 —— 即将根节点当成当前节点，如果新增节点大于当前节点、并且当前节点的右子节点存在，则以右子节点作为当前节点；如果新增节点小于当前节点、并且当前节点的左子节点存在，则以左子节点作为当前节点；如果新增节点等于当前节点，则用新增节点覆盖当前节点，并结束循环 —— 直到找到某个节点的左、右子节点不存在，将新节点添加该节点的子节点 —— 如果新节点比该节点大，则添加为右子节点；如果新节点比该节点小，则添加为左子节点。

TreeMap 的删除节点

当程序从排序二叉树中删除一个节点之后，为了让它依然保持为排序二叉树，程序必须对该排序二叉树进行维护。维护可分为如下几种情况：

（1）被删除的节点是叶子节点，则只需将它从其父节点中删除即可。

（2）被删除节点 p 只有左子树，将 p 的左子树 pL 添加成 p 的父节点的左子树即可；被删除节点 p 只有右子树，将 p 的右子树 pR 添加成 p 的父节点的右子树即可。

（3）若被删除节点 p 的左、右子树均非空，有两种做法：

* 将 pL 设为 p 的父节点 q 的左或右子节点（取决于 p 是其父节点 q 的左、右子节点），将 pR 设为 p 节点的中序前趋节点 s 的右子节点（s 是 pL 最右下的节点，也就是 pL 子树中最大的节点）。
* 以 p 节点的中序前趋或后继替代 p 所指节点，然后再从原排序二叉树中删去中序前趋或后继节点即可。（也就是用大于 p 的最小节点或小于 p 的最大节点代替 p 节点即可）。

图 2 显示了被删除节点只有左子树的示意图：

**图 2. 被删除节点只有左子树**

图 3 显示了被删除节点只有右子树的示意图：

**图 3. 被删除节点只有右子树**

图 4 显示了被删除节点既有左子节点，又有右子节点的情形，此时我们采用到是第一种方式进行维护：

**图 4. 被删除节点既有左子树，又有右子树**

图 5 显示了被删除节点既有左子树，又有右子树的情形，此时我们采用到是第二种方式进行维护：

**图 5. 被删除节点既有左子树，又有右子树**

TreeMap 删除节点采用图 5 所示右边的情形进行维护——也就是用被删除节点的右子树中最小节点与被删节点交换的方式进行维护。

TreeMap 删除节点的方法由如下方法实现：

|  |
| --- |
| private void deleteEntry(Entry<K,V> p)  {  modCount++;  size--;  // 如果被删除节点的左子树、右子树都不为空  if (p.left != null && p.right != null)  {  // 用 p 节点的中序后继节点代替 p 节点  Entry<K,V> s = successor (p);  p.key = s.key;  p.value = s.value;  p = s;  }  // 如果 p 节点的左节点存在，replacement 代表左节点；否则代表右节点。  Entry<K,V> replacement = (p.left != null ? p.left : p.right);  if (replacement != null)  {  replacement.parent = p.parent;  // 如果 p 没有父节点，则 replacemment 变成父节点  if (p.parent == null)  root = replacement;  // 如果 p 节点是其父节点的左子节点  else if (p == p.parent.left)  p.parent.left = replacement;  // 如果 p 节点是其父节点的右子节点  else  p.parent.right = replacement;  p.left = p.right = p.parent = null;  // 修复红黑树  if (p.color == BLACK)  fixAfterDeletion(replacement); // ①  }  // 如果 p 节点没有父节点  else if (p.parent == null)  {  root = null;  }  else  {  if (p.color == BLACK)  // 修复红黑树  fixAfterDeletion(p); // ②  if (p.parent != null)  {  // 如果 p 是其父节点的左子节点  if (p == p.parent.left)  p.parent.left = null;  // 如果 p 是其父节点的右子节点  else if (p == p.parent.right)  p.parent.right = null;  p.parent = null;  }  }  } |

红黑树

排序二叉树虽然可以快速检索，但在最坏的情况下：如果插入的节点集本身就是有序的，要么是由小到大排列，要么是由大到小排列，那么最后得到的排序二叉树将变成链表：所有节点只有左节点（如果插入节点集本身是大到小排列）；或所有节点只有右节点（如果插入节点集本身是小到大排列）。在这种情况下，排序二叉树就变成了普通链表，其检索效率就会很差。

为了改变排序二叉树存在的不足，Rudolf Bayer 与 1972 年发明了另一种改进后的排序二叉树：红黑树，他将这种排序二叉树称为“对称二叉 B 树”，而红黑树这个名字则由 Leo J. Guibas 和 Robert Sedgewick 于 1978 年首次提出。

红黑树是一个更高效的检索二叉树，因此常常用来实现关联数组。典型地，JDK 提供的集合类 TreeMap 本身就是一个红黑树的实现。

红黑树在原有的排序二叉树增加了如下几个要求：

## **Java 实现的红黑树**

上面的性质 3 中指定红黑树的每个叶子节点都是空节点，而且并叶子节点都是黑色。但 Java 实现的红黑树将使用 null 来代表空节点，因此遍历红黑树时将看不到黑色的叶子节点，反而看到每个叶子节点都是红色的。

* 性质 1：每个节点要么是红色，要么是黑色。
* 性质 2：根节点永远是黑色的。
* 性质 3：所有的叶节点都是空节点（即 null），并且是黑色的。
* 性质 4：每个红色节点的两个子节点都是黑色。（从每个叶子到根的路径上不会有两个连续的红色节点）
* 性质 5：从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点。

Java 中实现的红黑树可能有如图 6 所示结构：

**图 6. Java 红黑树的示意**

备注：本文中所有关于红黑树中的示意图采用白色代表红色。黑色节点还是采用了黑色表示。

根据性质 5：红黑树从根节点到每个叶子节点的路径都包含相同数量的黑色节点，因此从根节点到叶子节点的路径中包含的黑色节点数被称为树的“黑色高度（black-height）”。

性质 4 则保证了从根节点到叶子节点的最长路径的长度不会超过任何其他路径的两倍。假如有一棵黑色高度为 3 的红黑树：从根节点到叶节点的最短路径长度是 2，该路径上全是黑色节点（黑节点 - 黑节点 - 黑节点）。最长路径也只可能为 4，在每个黑色节点之间插入一个红色节点（黑节点 - 红节点 - 黑节点 - 红节点 - 黑节点），性质 4 保证绝不可能插入更多的红色节点。由此可见，红黑树中最长路径就是一条红黑交替的路径。

## **红黑树和平衡二叉树**

红黑树并不是真正的平衡二叉树，但在实际应用中，红黑树的统计性能要高于平衡二叉树，但极端性能略差。

由此我们可以得出结论：对于给定的黑色高度为 N 的红黑树，从根到叶子节点的最短路径长度为 N-1，最长路径长度为 2 \* (N-1)。

提示：排序二叉树的深度直接影响了检索的性能，正如前面指出，当插入节点本身就是由小到大排列时，排序二叉树将变成一个链表，这种排序二叉树的检索性能最低：N 个节点的二叉树深度就是 N-1。

红黑树通过上面这种限制来保证它大致是平衡的——因为红黑树的高度不会无限增高，这样保证红黑树在最坏情况下都是高效的，不会出现普通排序二叉树的情况。

由于红黑树只是一个特殊的排序二叉树，因此对红黑树上的只读操作与普通排序二叉树上的只读操作完全相同，只是红黑树保持了大致平衡，因此检索性能比排序二叉树要好很多。

但在红黑树上进行插入操作和删除操作会导致树不再符合红黑树的特征，因此插入操作和删除操作都需要进行一定的维护，以保证插入节点、删除节点后的树依然是红黑树。

添加节点后的修复

上面 put(K key, V value) 方法中①号代码处使用fixAfterInsertion(e) 方法来修复红黑树——因此每次插入节点后必须进行简单修复，使该排序二叉树满足红黑树的要求。

插入操作按如下步骤进行：

（1）以排序二叉树的方法插入新节点，并将它设为红色。

（2）进行颜色调换和树旋转。

## **插入后的修复**

在插入操作中，红黑树的性质 1 和性质 3 两个永远不会发生改变，因此无需考虑红黑树的这两个特性。

这种颜色调用和树旋转就比较复杂了，下面将分情况进行介绍。在介绍中，我们把新插入的节点定义为 N 节点，N 节点的父节点定义为 P 节点，P 节点的兄弟节点定义为 U 节点，P 节点父节点定义为 G 节点。

下面分成不同情形来分析插入操作

**情形 1：新节点 N 是树的根节点，没有父节点**

在这种情形下，直接将它设置为黑色以满足性质 2。

**情形 2：新节点的父节点 P 是黑色**

在这种情况下，新插入的节点是红色的，因此依然满足性质 4。而且因为新节点 N 有两个黑色叶子节点；但是由于新节点 N 是红色，通过它的每个子节点的路径依然保持相同的黑色节点数，因此依然满足性质 5。

**情形 3：如果父节点 P 和父节点的兄弟节点 U 都是红色**

在这种情况下，程序应该将 P 节点、U 节点都设置为黑色，并将 P 节点的父节点设为红色（用来保持性质 5）。现在新节点 N 有了一个黑色的父节点 P。由于从 P 节点、U 节点到根节点的任何路径都必须通过 G 节点，在这些路径上的黑节点数目没有改变（原来有叶子和 G 节点两个黑色节点，现在有叶子和 P 两个黑色节点）。

经过上面处理后，红色的 G 节点的父节点也有可能是红色的，这就违反了性质 4，因此还需要对 G 节点递归地进行整个过程（把 G 当成是新插入的节点进行处理即可）。

图 7 显示了这种处理过程：

**图 7. 插入节点后进行颜色调换**

备注：虽然图 11.28 绘制的是新节点 N 作为父节点 P 左子节点的情形，其实新节点 N 作为父节点 P 右子节点的情况与图 11.28 完全相同。

**情形 4：父节点 P 是红色、而其兄弟节点 U 是黑色或缺少；且新节点 N 是父节点 P 的右子节点，而父节点 P 又是其父节点 G 的左子节点。**

在这种情形下，我们进行一次左旋转对新节点和其父节点进行，接着按情形 5 处理以前的父节点 P（也就是把 P 当成新插入的节点即可）。这导致某些路径通过它们以前不通过的新节点 N 或父节点 P 的其中之一，但是这两个节点都是红色的，因此不会影响性质 5。

图 8 显示了对情形 4 的处理：

**图 8. 插入节点后的树旋转**

备注：图 11.29 中 P 节点是 G 节点的左子节点，如果 P 节点是其父节点 G 节点的右子节点，那么上 **面的处理情况应该左、右对调一下。**

**情形 5：父节点 P 是红色、而其兄弟节点 U 是黑色或缺少；且新节点 N 是其父节点的左子节点，而父节点 P 又是其父节点 G 的左子节点。**

在这种情形下，需要对节点 G 的一次右旋转，在旋转产生的树中，以前的父节点 P 现在是新节点 N 和节点 G 的父节点。由于以前的节点 G 是黑色，否则父节点 P 就不可能是红色，我们切换以前的父节点 P 和节点 G 的颜色，使之满足性质 4，性质 5 也仍然保持满足，因为通过这三个节点中任何一个的所有路径以前都通过节点 G，现在它们都通过以前的父节点 P。在各自的情形下，这都是三个节点中唯一的黑色节点。

图 9 显示了情形 5 的处理过程：

**图 9. 插入节点后的颜色调整、树旋转**

备注：图 11.30 中 P 节点是 G 节点的左子节点，如果 P 节点是其父节点 G 节点的右子节点，那么上面的处理情况应该左、右对调一下。

TreeMap 为插入节点后的修复操作由 fixAfterInsertion(Entry<K,V> x) 方法提供，该方法的源代码如下：

|  |
| --- |
| // 插入节点后修复红黑树  private void fixAfterInsertion(Entry<K,V> x)  {  x.color = RED;  // 直到 x 节点的父节点不是根，且 x 的父节点不是红色  while (x != null && x != root  && x.parent.color == RED)  {  // 如果 x 的父节点是其父节点的左子节点  if (parentOf(x) == leftOf(parentOf(parentOf(x))))  {  // 获取 x 的父节点的兄弟节点  Entry<K,V> y = rightOf(parentOf(parentOf(x)));  // 如果 x 的父节点的兄弟节点是红色  if (colorOf(y) == RED)  {  // 将 x 的父节点设为黑色  setColor(parentOf(x), BLACK);  // 将 x 的父节点的兄弟节点设为黑色  setColor(y, BLACK);  // 将 x 的父节点的父节点设为红色  setColor(parentOf(parentOf(x)), RED);  x = parentOf(parentOf(x));  }  // 如果 x 的父节点的兄弟节点是黑色  else  {  // 如果 x 是其父节点的右子节点  if (x == rightOf(parentOf(x)))  {  // 将 x 的父节点设为 x  x = parentOf(x);  rotateLeft(x);  }  // 把 x 的父节点设为黑色  setColor(parentOf(x), BLACK);  // 把 x 的父节点的父节点设为红色  setColor(parentOf(parentOf(x)), RED);  rotateRight(parentOf(parentOf(x)));  }  }  // 如果 x 的父节点是其父节点的右子节点  else  {  // 获取 x 的父节点的兄弟节点  Entry<K,V> y = leftOf(parentOf(parentOf(x)));  // 如果 x 的父节点的兄弟节点是红色  if (colorOf(y) == RED)  {  // 将 x 的父节点设为黑色。  setColor(parentOf(x), BLACK);  // 将 x 的父节点的兄弟节点设为黑色  setColor(y, BLACK);  // 将 x 的父节点的父节点设为红色  setColor(parentOf(parentOf(x)), RED);  // 将 x 设为 x 的父节点的节点  x = parentOf(parentOf(x));  }  // 如果 x 的父节点的兄弟节点是黑色  else  {  // 如果 x 是其父节点的左子节点  if (x == leftOf(parentOf(x)))  {  // 将 x 的父节点设为 x  x = parentOf(x);  rotateRight(x);  }  // 把 x 的父节点设为黑色  setColor(parentOf(x), BLACK);  // 把 x 的父节点的父节点设为红色  setColor(parentOf(parentOf(x)), RED);  rotateLeft(parentOf(parentOf(x)));  }  }  }  // 将根节点设为黑色  root.color = BLACK;  } |

删除节点后的修复

与添加节点之后的修复类似的是，TreeMap 删除节点之后也需要进行类似的修复操作，通过这种修复来保证该排序二叉树依然满足红黑树特征。大家可以参考插入节点之后的修复来分析删除之后的修复。TreeMap 在删除之后的修复操作由 fixAfterDeletion(Entry<K,V> x) 方法提供，该方法源代码如下：

|  |
| --- |
| // 删除节点后修复红黑树  private void fixAfterDeletion(Entry<K,V> x)  {  // 直到 x 不是根节点，且 x 的颜色是黑色  while (x != root && colorOf(x) == BLACK)  {  // 如果 x 是其父节点的左子节点  if (x == leftOf(parentOf(x)))  {  // 获取 x 节点的兄弟节点  Entry<K,V> sib = rightOf(parentOf(x));  // 如果 sib 节点是红色  if (colorOf(sib) == RED)  {  // 将 sib 节点设为黑色  setColor(sib, BLACK);  // 将 x 的父节点设为红色  setColor(parentOf(x), RED);  rotateLeft(parentOf(x));  // 再次将 sib 设为 x 的父节点的右子节点  sib = rightOf(parentOf(x));  }  // 如果 sib 的两个子节点都是黑色  if (colorOf(leftOf(sib)) == BLACK  && colorOf(rightOf(sib)) == BLACK)  {  // 将 sib 设为红色  setColor(sib, RED);  // 让 x 等于 x 的父节点  x = parentOf(x);  }  else  {  // 如果 sib 的只有右子节点是黑色  if (colorOf(rightOf(sib)) == BLACK)  {  // 将 sib 的左子节点也设为黑色  setColor(leftOf(sib), BLACK);  // 将 sib 设为红色  setColor(sib, RED);  rotateRight(sib);  sib = rightOf(parentOf(x));  }  // 设置 sib 的颜色与 x 的父节点的颜色相同  setColor(sib, colorOf(parentOf(x)));  // 将 x 的父节点设为黑色  setColor(parentOf(x), BLACK);  // 将 sib 的右子节点设为黑色  setColor(rightOf(sib), BLACK);  rotateLeft(parentOf(x));  x = root;  }  }  // 如果 x 是其父节点的右子节点  else  {  // 获取 x 节点的兄弟节点  Entry<K,V> sib = leftOf(parentOf(x));  // 如果 sib 的颜色是红色  if (colorOf(sib) == RED)  {  // 将 sib 的颜色设为黑色  setColor(sib, BLACK);  // 将 sib 的父节点设为红色  setColor(parentOf(x), RED);  rotateRight(parentOf(x));  sib = leftOf(parentOf(x));  }  // 如果 sib 的两个子节点都是黑色  if (colorOf(rightOf(sib)) == BLACK  && colorOf(leftOf(sib)) == BLACK)  {  // 将 sib 设为红色  setColor(sib, RED);  // 让 x 等于 x 的父节点  x = parentOf(x);  }  else  {  // 如果 sib 只有左子节点是黑色  if (colorOf(leftOf(sib)) == BLACK)  {  // 将 sib 的右子节点也设为黑色  setColor(rightOf(sib), BLACK);  // 将 sib 设为红色  setColor(sib, RED);  rotateLeft(sib);  sib = leftOf(parentOf(x));  }  // 将 sib 的颜色设为与 x 的父节点颜色相同  setColor(sib, colorOf(parentOf(x)));  // 将 x 的父节点设为黑色  setColor(parentOf(x), BLACK);  // 将 sib 的左子节点设为黑色  setColor(leftOf(sib), BLACK);  rotateRight(parentOf(x));  x = root;  }  }  }  setColor(x, BLACK);  } |

检索节点

当 TreeMap 根据 key 来取出 value 时，TreeMap 对应的方法如下：

|  |
| --- |
| public V get(Object key)  {  // 根据指定 key 取出对应的 Entry  Entry>K,V< p = getEntry(key);  // 返回该 Entry 所包含的 value  return (p==null ? null : p.value);  } |

从上面程序的粗体字代码可以看出，get(Object key) 方法实质是由于 getEntry() 方法实现的，这个 getEntry() 方法的代码如下：

|  |
| --- |
| final Entry<K,V> getEntry(Object key)  {  // 如果 comparator 不为 null，表明程序采用定制排序  if (comparator != null)  // 调用 getEntryUsingComparator 方法来取出对应的 key  return getEntryUsingComparator(key);  // 如果 key 形参的值为 null，抛出 NullPointerException 异常  if (key == null)  throw new NullPointerException();  // 将 key 强制类型转换为 Comparable 实例  Comparable<? super K> k = (Comparable<? super K>) key;  // 从树的根节点开始  Entry<K,V> p = root;  while (p != null)  {  // 拿 key 与当前节点的 key 进行比较  int cmp = k.compareTo(p.key);  // 如果 key 小于当前节点的 key，向“左子树”搜索  if (cmp < 0)  p = p.left;  // 如果 key 大于当前节点的 key，向“右子树”搜索  else if (cmp > 0)  p = p.right;  // 不大于、不小于，就是找到了目标 Entry  else  return p;  }  return null;  } |

上面的 getEntry(Object obj) 方法也是充分利用排序二叉树的特征来搜索目标 Entry，程序依然从二叉树的根节点开始，如果被搜索节点大于当前节点，程序向“右子树”搜索；如果被搜索节点小于当前节点，程序向“左子树”搜索；如果相等，那就是找到了指定节点。

当 TreeMap 里的 comparator != null 即表明该 TreeMap 采用了定制排序，在采用定制排序的方式下，TreeMap 采用 getEntryUsingComparator(key) 方法来根据 key 获取 Entry。下面是该方法的代码：

|  |
| --- |
| final Entry<K,V> getEntryUsingComparator(Object key)  {  K k = (K) key;  // 获取该 TreeMap 的 comparator  Comparator<? super K> cpr = comparator;  if (cpr != null)  {  // 从根节点开始  Entry<K,V> p = root;  while (p != null)  {  // 拿 key 与当前节点的 key 进行比较  int cmp = cpr.compare(k, p.key);  // 如果 key 小于当前节点的 key，向“左子树”搜索  if (cmp < 0)  p = p.left;  // 如果 key 大于当前节点的 key，向“右子树”搜索  else if (cmp > 0)  p = p.right;  // 不大于、不小于，就是找到了目标 Entry  else  return p;  }  }  return null;  } |

其实 getEntry、getEntryUsingComparator 两个方法的实现思路完全类似，只是前者对自然排序的 TreeMap 获取有效，后者对定制排序的 TreeMap 有效。

通过上面源代码的分析不难看出，TreeMap 这个工具类的实现其实很简单。或者说：从内部结构来看，TreeMap 本质上就是一棵“红黑树”，而 TreeMap 的每个 Entry 就是该红黑树的一个节点。

二叉树遍历算法

void PreOrder(BiTree root)

{

if(root==NULL)

return ;

printf("%c ", root->data); //输出数据

PreOrder(root->lchild); //递归调用，先序遍历左子树

PreOrder(root->rchild); //递归调用，先序遍历右子树

}

二叉树的度：

首先说说什么是度：通俗的讲二叉树中连接节点和节点的线就是度，有n个节点，就有n-1个度，节点数总是比度要多一个，那么度为0的节点一定是叶子节点，因为该节点的下面不再有线；度为1的节点即：该节点只有一个分支；同理度为2的节点就是有两个分支。在二叉树中不可能存在度为3或大于3的节点！  
  
关于度和节点之间的关系还有很多公式：度为0的节点数为度为2的节点数加1，即n0=n2+1  
这个公式的推理方法如下：  
设：  
k:总度数  
k+1:总节点数  
n0:度为0的节点  
n1:度为1的节点  
n2:度为二的节点  
根据二叉树中度和节点的守衡原理，可列出以下一组方程：  
k=n2\*2+n1;  
k+1=n2+n1+n0;  
将上面两式相减得到：n0=n2+1；  
  
例如：已知767个节点的完全二叉树，求其叶子节点个数：

n0=n2+1;

n=n0+n1+n2;

由上面，消掉n2得到：n=2n0+n1-1;

由于完全二叉树度为1的只有0个或1个两种情况，所以，将0或1带入上面公式，整理后得：

n0=（n+1）/2或者n0=n/2;

看看n是否能被2整除，能则用n0=n/2。否则用n0=（n+1）/2

既叶子节点为n0=（n+1）/2=384

-----------------------------------------------------------------

再比如一棵二叉树有10个度为1的节点，7个度为2的节点，则二叉树有多少个节点（25）

根据刚才说的，节点数比度数多1，可以列出计算式子：

10 \* 1 + 7 \* 2 + 1 = 25

**性质1** 二叉树第i层上的结点数目最多为2i-1(i≥1)。  
*证明*：用数学归纳法证明：  
　    归纳基础：i=1时，有2i-1=20=1。因为第1层上只有一个根结点，所以命题成立。  
    　归纳假设：假设对所有的j(1≤j<i)命题成立，即第j层上至多有2j-1个结点，证明j=i时命题亦成立。  
    　归纳步骤：根据归纳假设，第i-1层上至多有2i-2个结点。由于二叉树的每个结点至多有两个孩子，故第i层上的结点数至多是第i-1层上的最大结点数的2倍。即j=i时，该层上至多有2×2i-2=2i-1个结点，故命题成立。  
  
**性质2** 深度为k的二叉树至多有2k-1个结点(k≥1)。  
*证明*：在具有相同深度的二叉树中，仅当每一层都含有最大结点数时，其树中结点数最多。因此利用性质1可得，深度为k的二叉树的结点数至多为：  
                20+21+…+2k-1=2k-1  
    故命题正确。  
  
**性质3** 在任意-棵二叉树中，若终端结点的个数为n0，度为2的结点数为n2，则no=n2+1。  
*证明*：因为二叉树中所有结点的度数均不大于2，所以结点总数(记为n)应等于0度结点数、1度结点(记为n1)和2度结点数之和：  
                     n=no+n1+n2 (式子1)  
　    另一方面，1度结点有一个孩子，2度结点有两个孩子，故二叉树中孩子结点总数是：  
                      nl+2n2  
　　树中只有根结点不是任何结点的孩子，故二叉树中的结点总数又可表示为：  
                      n=n1+2n2+1 (式子2)  
　　由式子1和式子2得到：  
                      no=n2+1  
  
**满二叉树和完全二叉树是二叉树的两种特殊情形。**  
1、满二叉树(FullBinaryTree)   
    　一棵深度为k且有2k-1个结点的二又树称为满二叉树。  
    　满二叉树的特点：  
　　（1） 每一层上的结点数都达到最大值。即对给定的高度，它是具有最多结点数的二叉树。  
　　（2） 满二叉树中不存在度数为1的结点，每个分支结点均有两棵高度相同的子树，且树叶都在最下一层上。  
　　【例】图(a)是一个深度为4的满二叉树。  
           
  
  
2、完全二叉树(Complete BinaryTree)   
    若一棵二叉树至多只有最下面的两层上结点的度数可以小于2，并且最下一层上的结点都集中在该层最左边的若干位置上，则此二叉树称为完全二叉树。  
  **特点：**  
  （1） 满二叉树是完全二叉树，完全二叉树不一定是满二叉树。  
  （2） 在满二叉树的最下一层上，从最右边开始连续删去若干结点后得到的二叉树仍然是一棵完全二叉树。  
  （3） 在完全二叉树中，若某个结点没有左孩子，则它一定没有右孩子，即该结点必是叶结点。  
【例】如图(c)中，结点F没有左孩子而有右孩子L，故它不是一棵完全二叉树。  
【例】图(b)是一棵完全二叉树。  
    
**性质4** 具有n个结点的完全二叉树的深度为  
                          IMG_256   
*证明*：设所求完全二叉树的深度为k。由完全二叉树定义可得：  
　　深度为k得完全二叉树的前k-1层是深度为k-1的满二叉树，一共有2k-1-1个结点。  
由于完全二叉树深度为k，故第k层上还有若干个结点，因此该完全二叉树的结点个数：  
                  n>2k-1-1。  
    　另一方面，由性质2可得：  
                  n≤2k-1，  
       即：2k-1-l<n≤2k-1  
    　由此可推出：2k-1≤n<2k，取对数后有：  
                  k-1≤lgn<k  
    　又因k-1和k是相邻的两个整数，故有  
                                    IMG_257 ,  
    　由此即得：  
                 IMG_258   
注意：  
         IMG_259 的证明【参见参考书目】

Queen deque的用法，队列的用法

package cn.com.List;

import java.io.Serializable;

import java.util.ArrayDeque;

import java.util.Queue;

import com.sun.jmx.remote.internal.ArrayQueue;

/\*\*

\* 模拟银行存款

\* @author Administrator

\*

\*/

public class QueueDemo {

public static void main(String[] args) {

Queue<Request> queue= new ArrayDeque<Request>();

for ( int i = 0; i < 10; i++) {

final int num=i;

queue.offer(new Request() {

public void desposit() {

System.out.println("第"+num+"個人存了"+(num+num\*20)+"元");

}

});

}

Request req=null;

if(queue!=null){

while((req=queue.poll())!=null){

req.desposit();

}

}

}

}

interface Request{

void desposit();

}