java常用集合框架

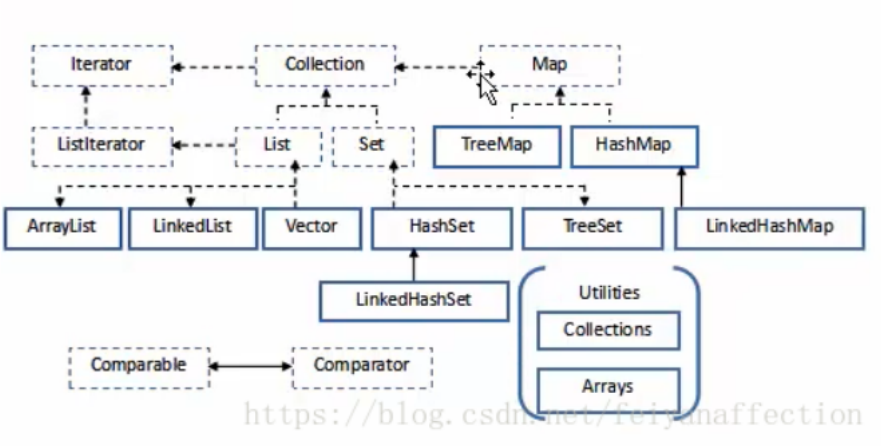
转载于（写的不错）：https://www.cnblogs.com/linliquan/p/11323172.html

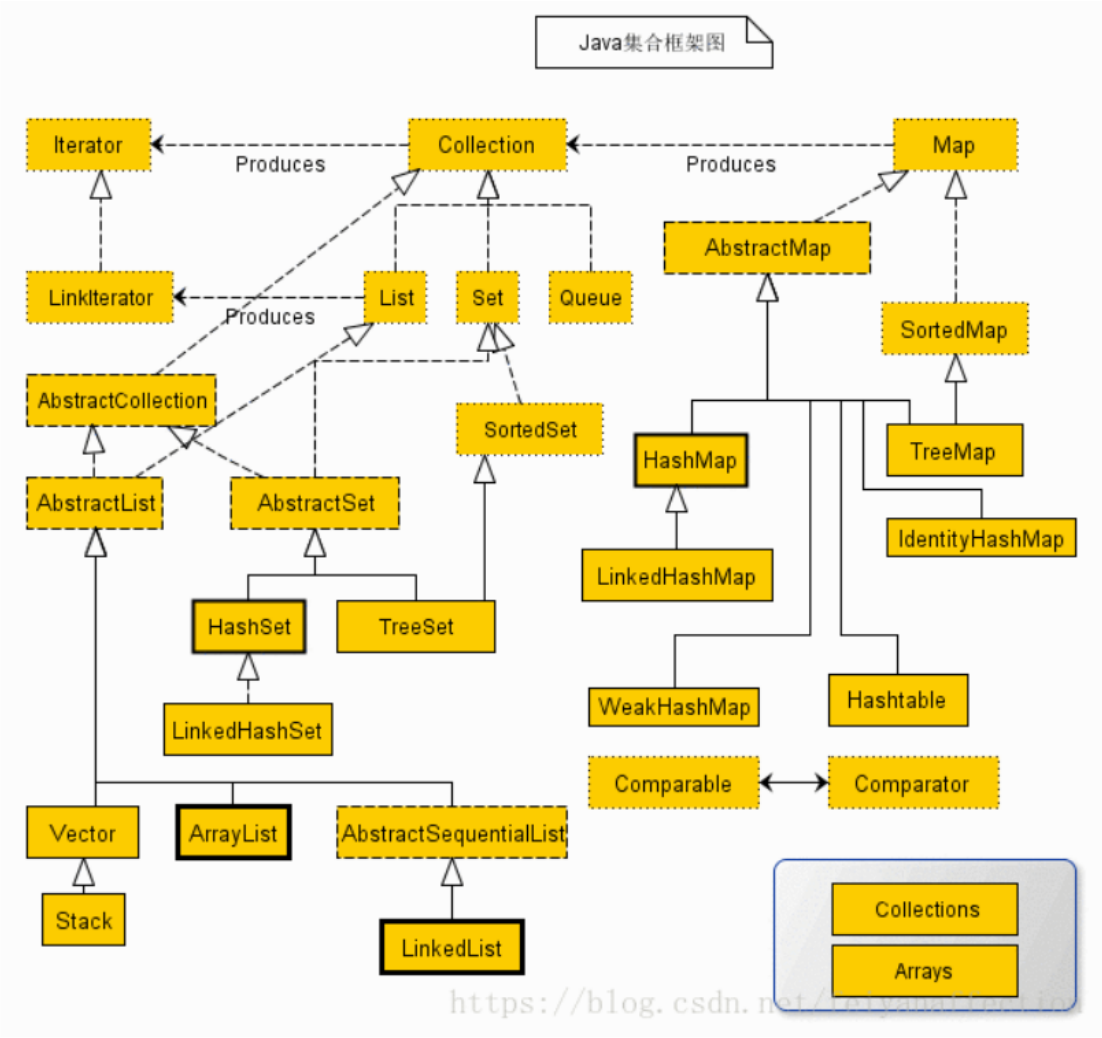
Java的集合主要有List , Set, Map

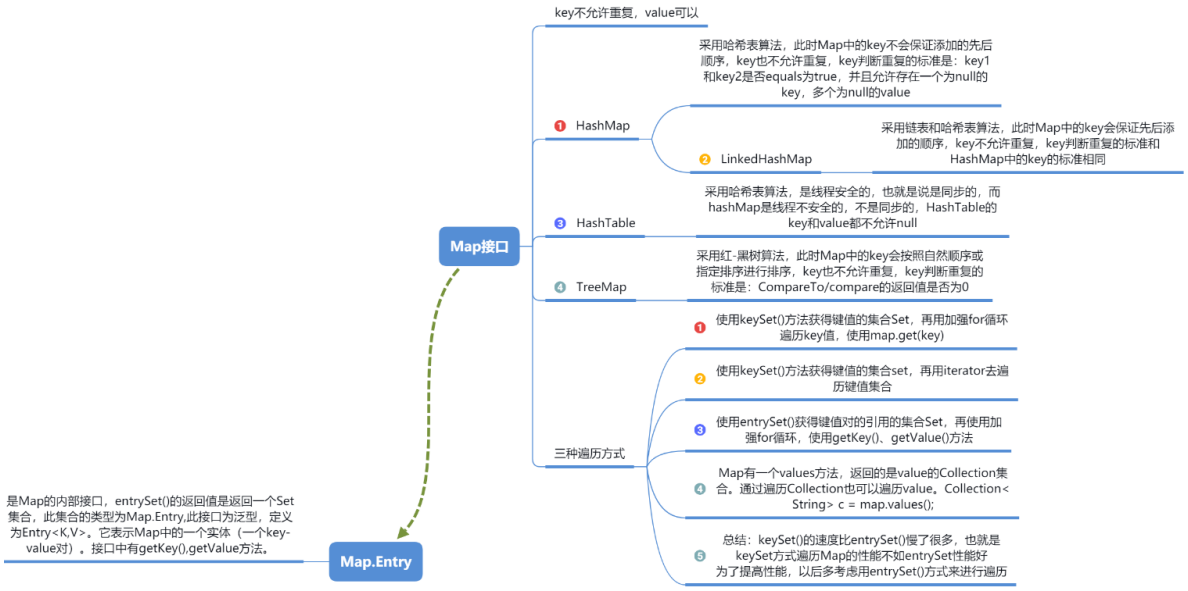
List , Set继承至Collection接口，Map为独立接口

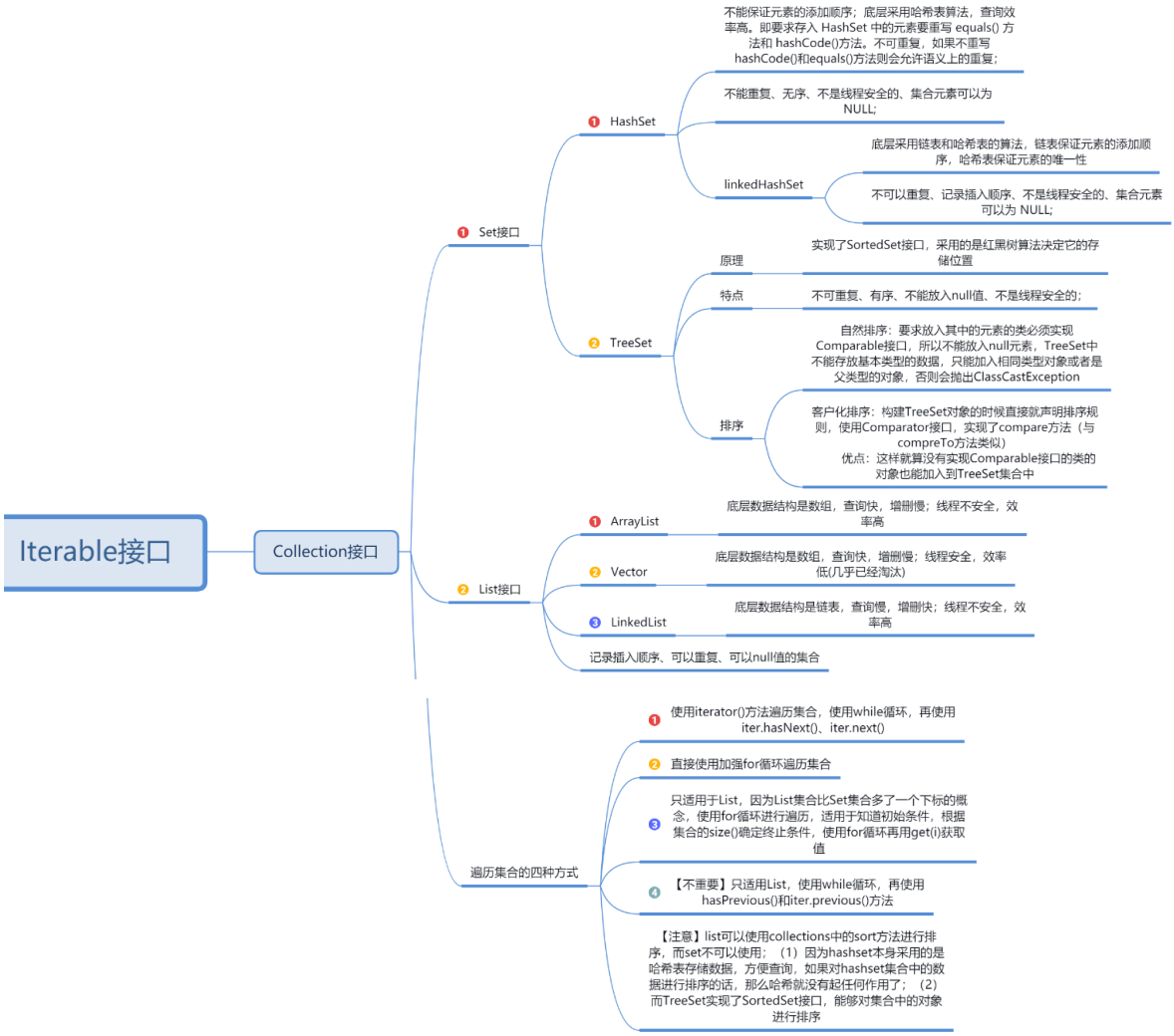
List下有ArrayList，LinkedList，Vector

Set下有HashSet，LinkedHashSet，TreeSet
Map下有HashMap，LinkedHashMap， TreeMap，Hashtable




Java集合框架图
Iterator
Produces
Collection
Produces
Map
LinkIterator
Produces
List
Set
Queue
AbstractMap
SortedMap
AbstractCollection
SortedSet
HashMap
TreeMap
AbstractList
AbstractSet
IdentityHashMap
LinkedHashMap
HashSet
TreeSet
WeakHashMap
Hashtable
LinkedHashSet
Comparable
Comparator
Vector
ArrayList
AbstractSequentialList
Stack
LinkedList
Collections
Arrays


key不允许重复，value可以
采用哈希表算法，此时Map中的key不会保证添加的先后顺序，key也不允许重复，key判断重复的标准是：key1和key2是否equals为true，并且允许存在一个为null的key，多个为null的value
HashMap
采用链表和哈希表算法，此时Map中的key会保证先后添加的顺序，key不允许重复，key判断重复的标准和HashMap中的key的标准相同
LinkedHashMap
采用哈希表算法，是线程安全的，也就是说是同步的，而hashMap是线程不安全的，不是同步的，HashTable的key和value都不允许null
HashTable
Map接口
采用红-黑树算法，此时Map中的key会按照自然顺序或指定排序进行排序，key也不允许重复，key判断重复的标准是：CompareTo/compare的返回值是否为0
TreeMap
使用keySet()方法获得键值的集合Set，再用加强for循环遍历key值，使用map.get(key)
使用keySet()方法获得键值的集合set，再用iterator去遍历键值集合
使用entrySet()获得键值对的引用的集合Set，再使用加强for循环，使用getKey()、getValue()方法
三种遍历方式
Map有一个values方法，返回的是value的Collection集合。通过遍历Collection也可以遍历value。Collection<String> c = map.values();
总结：keySet()的速度比entrySet()慢了很多，也就是keySet方式遍历Map的性能不如entrySet性能好为了提高性能，以后多考虑用entrySet()方式来进行遍历
是Map的内部接口，entrySet()的返回值是返回一个Set集合，此集合的类型为Map.Entry,此接口为泛型，定义为Entry<K,V>。它表示Map中的一个实体（一个key-value对）。接口中有getKey(),getValue方法。
Map.Entry



总结:
Connection接口:

1.List 有序,可重复

ArrayList:
优点: 底层数据结构是数组，查询快，增删慢。
缺点: 线程不安全，效率高

LinkedList:
优点: 底层数据结构是链表，查询慢，增删快。
缺点: 线程不安全，效率高

Vector:
优点: 底层数据结构是数组，查询快，增删慢。
缺点: 线程安全，效率低

2.Set 无序,唯一

（1）HashSet：
底层数据结构是哈希表。(无序,唯一)
如何来保证元素唯一性?
1.依赖两个方法：hashCode()和equals()

HashSet底层数据结构采用哈希表实现，元素无序且唯一，线程不安全，效率高，可以存储null元素，元素的唯一性是靠所存储元素类型是否重写hashCode()和equals()方法来保证的，如果没有重写这两个方法，则无法保证元素的唯一性。

具体实现唯一性的比较过程:

```
1.存储元素时首先会使用hash()算法函数生成一个int类型hashCode散列值，然后已经的所存储的元素的
hashCode值比较，如果hashCode不相等，肯定是不同的对象。
2.hashCode值相同，再比较equals方法。
3.equals相同，对象相同。（则无需储存）
```

（2）LinkedHashSet：
底层数据结构是链表和哈希表。(FIFO插入有序,唯一)
1.由链表保证元素有序
2.由哈希表保证元素唯一

LinkedHashSet底层数据结构采用链表和哈希表共同实现，链表保证了元素的顺序与存储顺序一致，哈希表保证了元素的唯一性。线程不安全，效率高。

（3）TreeSet：
底层数据结构是红黑树。(唯一，有序)
\1. 如何保证元素排序的呢?
自然排序
比较器排序
2.如何保证元素唯一性的呢?
根据比较的返回值是否是0来决定

TreeSet底层数据结构采用红黑树来实现，元素唯一且已经排好序；唯一性同样需要重写hashCode和equals()方法，二叉树结构保证了元素的有序性。根据构造方法不同，分为自然排序（无参构造）和比较器排序（有参构造），自然排序要求元素必须实现Compareable接口，并重写里面的compareTo()方法，元素通过比较返回的int值来判断排序序列，返回0说明两个对象相同，不需要存储；比较器排需要在TreeSet初始化是时候传入一个实现Comparator接口的比较器对象，或者采用匿名内部类的方式new一个Comparator对象，重写里面的compare()方法；

红黑树：

在学习红黑树之前，咱们需要先来理解下二叉查找树（BST）。

二叉查找树

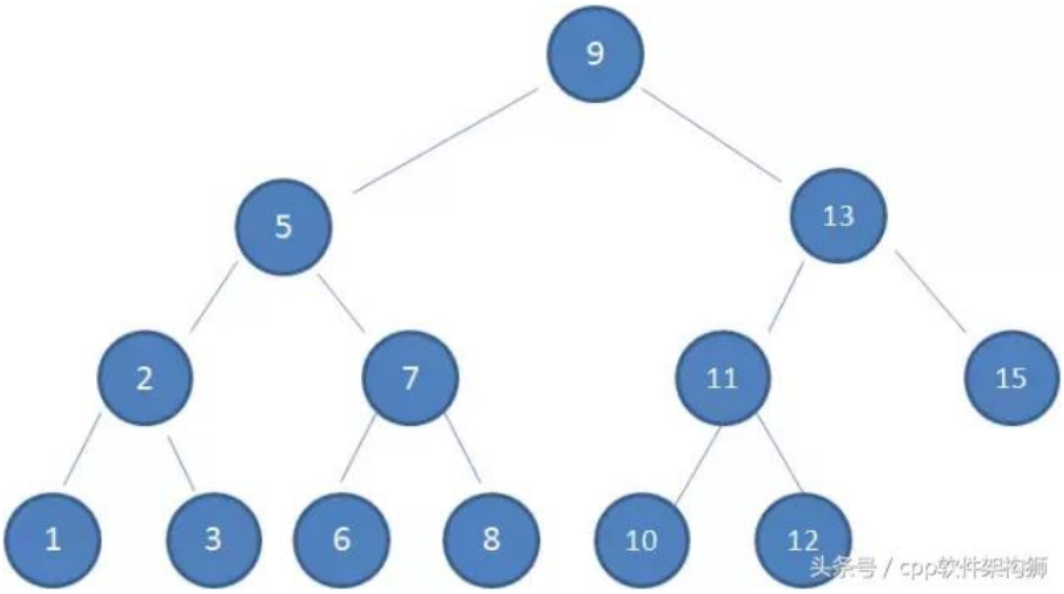
要想了解二叉查找树，我们首先看下二叉查找树有哪些特性呢?

1， 左子树上所有的节点的值均小于或等于他的根节点的值

2， 右子数上所有的节点的值均大于或等于他的根节点的值

3， 左右子树也一定分别为二叉排序树

我们来看下图的这棵树，他就是典型的二叉查找树



红黑树

红黑树就是一种平衡的二叉查找树，说他平衡的意思是他不会变成“瘸子”，左腿特别长或者右腿特别长。除了符合二叉查找树的特性之外，还具体下列的特性：

1. 节点是红色或者黑色

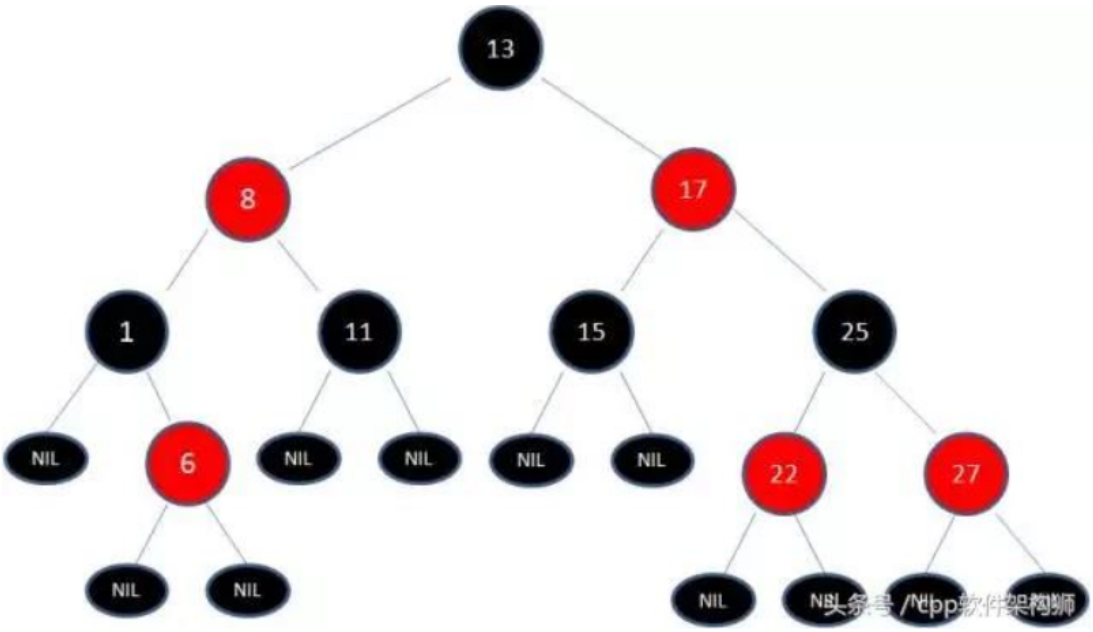
2. 根节点是黑色

3. 每个叶子的节点都是黑色的空节点（NULL）

4. 每个红色节点的两个子节点都是黑色的。

5. 从任意节点到其每个叶子的所有路径都包含相同的黑色节点。

看下图就是一个典型的红黑树：



红黑树详情：http://www.360doc.com/content/18/0904/19/25944647_783893127.shtml

TreeSet的两种排序方式比较

1.基本数据类型默认按升序排序

2.自定义排序

（1）自然排序：重写Comparable接口中的Compareto方法

（2）比较器排序：重写Comparator接口中的Compare方法

```
compare(T o1,T o2)        比较用来排序的两个参数。
o1：代表当前添加的数据
o2：代表集合中已经存在的数据
0：  表示 o1 == o2
-1(逆序输出)：  o1 < o2
1(正序输出): o1 > o2
```

1：o1 - o2（升序排列）
-1：o2 - o1 (降序排列)

例子1：

```
 1 import java.util.Comparator;
 2 import java.util.Set;
 3 import java.util.TreeSet;
 4
 5 public class Test {
 6     public static void main(String[] args) {
 7
 8         /**
 9          * 自定义规则的TreeSet
10          * 客户端排序：自己写一个比较器，转给TreeSet
11          *
12          * 比较规则
13          * 当TreeSet集合添加数据的时候就会触发比较器的compare()方法
14          */
15         Comparator<Integer> comp = new Comparator<Integer>() {
16             /**
17              * o1 当前添加的数据
18              * o2 集合中已经存在的数据
19              * 0：  表示 o1 == o2
20              * -1 ：  o1 < o2
21              * 1 : o1 > o2
22              */
23             @Override
24             public int compare(Integer o1, Integer o2) {
25                 System.out.println(o1+"--"+o2);
26                 return o2 -o1; //输出53 33 10，降序排序
27               //  return  0;  //只输出一个元素：33
28               //   return -1; //输出53 10 33，倒序输出
29               //  return 1;  //输出33 10 55
30             }
31         };
32
```

```
        Set<Integer> s2 = new TreeSet<>(comp);
        s2.add(33);
        s2.add(10);
        s2.add(55);

        System.out.println(s2); //输入53 33 10，降序排序

    }
}
```

例2：

```
import java.util.Comparator;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

/**
 * 使用TreeSet和Comparator（使用匿名类），写Test.java
 * 要求：对TreeSet中的元素
 *      1，2，3，4，5，6，7，8，9，10进行排列，
 * 排序逻辑为奇数在前偶数在后，
 * 奇数按照升序排列，偶数按照降序排列
 * 输出结果：1 3 5 7 9 10 8 6 4 2
 */
public class Test {
    public static void main(String[] args) {
        Set<Integer> s = new TreeSet<>(new Comparator<Integer>() {
            //重写compare方法
            @Override
            public int compare(Integer o1, Integer o2) {
                System.out.println("o1="+o1+" o2="+o2);
                if(o2%2==0){
                    if (o1%2==0){
                            return o2 -o1;
                    }else{
                        return -1;
                    }
                }else {
                    if (o1%2==0){
                        return 1;
                    }else{
                        return o1 -o2;
                    }
                }


            }
        });

        s.add(2);
        s.add(6);
```

```
41          s.add(4);
42          s.add(1);
43          s.add(3);
44          s.add(5);
45          s.add(8);
46          s.add(10);
47          s.add(9);
48          s.add(7);
49
50          Iterator iterator = s.iterator();
51
52          while(iterator.hasNext()){
53              System.out.print(iterator.next()+" ");
54          }
55
56      }
57  }
```

输出结果：

```
1 3 5 7 9 10 8 6 4 2
Process finished with exit code 0
```

3.Map接口:

Map用于保存具有映射关系的数据，Map里保存着两组数据：key和value，它们都可以使任何引用类型的数据，但key不能重复。所以通过指定的key就可以取出对应的value。

Map接口有四个比较重要的实现类，分别是HashMap、LinkedHashMap、TreeMap和HashTable。

TreeMap是有序的，HashMap和HashTable是无序的。

Hashtable的方法是同步的，HashMap的方法不是同步的。这是两者最主要的区别。

HashMap

Map 主要用于存储键(key)值(value)对，根据键得到值，因此键不允许重复,但允许值重复。
HashMap 是一个最常用的Map,它根据键的HashCode 值存储数据,根据键可以直接获取它的值，具有很快的访问速度。
HashMap最多只允许一条记录的键为Null;允许多条记录的值为 Null;
HashMap不支持线程的同步，即任一时刻可以有多个线程同时写HashMap;可能会导致数据的不一致。如果需要同步，可以用 Collections的synchronizedMap方法使HashMap具有同步的能力，或者使用ConcurrentHashMap。
HashMap基于哈希表结构实现的 ，当一个对象被当作键时，必须重写hasCode和equals方法。

LinkedHashMap

LinkedHashMap继承自HashMap，它主要是用链表实现来扩展HashMap类，HashMap中条目是没有顺序的，但是在LinkedHashMap中元素既可以按照它们插入图的顺序排序，也可以按它们最后一次被访问的顺序排序。

TreeMap

TreeMap基于红黑树数据结构的实现，键值可以使用Comparable或Comparator接口来排序。TreeMap继承自AbstractMap，同时实现了接口NavigableMap，而接口NavigableMap则继承自SortedMap。SortedMap是Map的子接口，使用它可以确保图中的条目是排好序的。

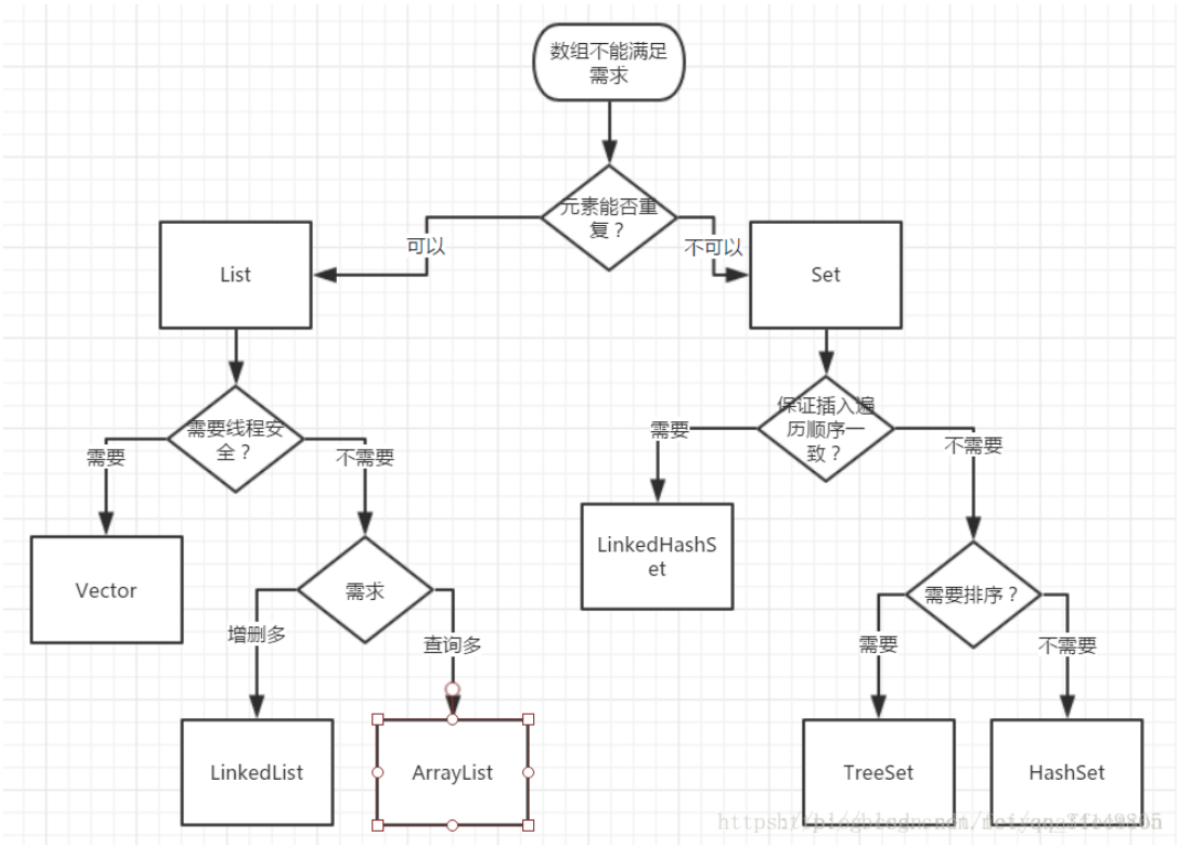
在实际使用中，如果更新图时不需要保持图中元素的顺序，就使用HashMap，如果需要保持图中元素的插入顺序或者访问顺序，就使用LinkedHashMap，如果需要使图按照键值排序，就使用TreeMap。

Hashtable

Hashtable和前面介绍的HashMap很类似，它也是一个散列表，存储的内容是键值对映射，不同之处在于，Hashtable是继承自Dictionary的，Hashtable中的函数都是同步的，这意味着它也是线程安全的，另外，Hashtable中key和value都不可以为null。

适用场景分析:HashSet是基于Hash算法实现的，其性能通常都优于TreeSet。为快速查找而设计的Set，我们通常都应该使用HashSet，在我们需要排序的功能时，我们才使用TreeSet。

怎么选择：



遍历map实例

```
 1 import java.util.HashMap;
 2 import java.util.Iterator;
 3 import java.util.Map;
 4
 5 public class Test {
 6
 7     public static void main(String[] args) {
 8         Map<String, String> map = new HashMap<String, String>();
 9         map.put("first", "linlin");
10         map.put("second", "好好学java");
11         map.put("third", "sihai");
12         map.put("first", "sihai2");
13
14
15         // 第一种：通过Map.keySet遍历key和value
16         System.out.println("==================通过Map.keySet遍历key和
value:==================");
17         for (String key : map.keySet()) {
18             System.out.println("key= " + key + "  and  value= " +
map.get(key));
19         }
20
21         // 第二种：通过Map.entrySet使用iterator遍历key和value
22         System.out.println("==================通过Map.entrySet使用iterator遍历
key和value:==================");
23         Iterator<Map.Entry<String, String>> it = map.entrySet().iterator();

24         while (it.hasNext()) {
25             Map.Entry<String, String> entry = it.next();
26             System.out.println("key= " + entry.getKey() + "  and  value= "

27                     + entry.getValue());
28         }
29
30         // 第三种：通过Map.entrySet遍历key和value
31         System.out.println("==================通过Map.entrySet遍历key和
value:==================");
32         for (Map.Entry<String, String> entry : map.entrySet()) {
33             System.out.println("key= " + entry.getKey() + "  and  value= "

34                     + entry.getValue());
35         }
36
37         // 第四种：通过Map.values()遍历所有的value，但是不能遍历键key
38         System.out.println("==================通过Map.values()遍历所有的
value:==================");
39         for (String v : map.values()) {
40             System.out.println("value= " + v);
41         }
42     }
43
44 }
```

重点问题重点分析:

(一）说说List,Set,Map三者的区别?

- List(对付顺序的好帮手)： List接口存储一组不唯一（可以有多个元素引用相同的对象），有序的对象
- Set(注重独一无二的性质): 不允许重复的集合。不会有多个元素引用相同的对象。
- Map(用Key来搜索的专家): 使用键值对存储。Map会维护与Key有关联的值。两个Key可以引用相同的对象，但Key不能重复，典型的Key是String类型，但也可以是任何对象。

(二）Arraylist 与 LinkedList 区别?

- \1. 是否保证线程安全： `ArrayList` 和 `LinkedList` 都是不同步的，也就是不保证线程安全；
- \2. 底层数据结构： `Arraylist` 底层使用的是 `Object` 数组；`LinkedList` 底层使用的是 双向链表 数据结构（JDK1.6之前为循环链表，JDK1.7取消了循环。注意双向链表和双向循环链表的区别，下面有介绍到！）
- \3. 插入和删除是否受元素位置的影响： ① `ArrayList` 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。 比如：执行`add(E e)`方法的时候， `ArrayList` 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是O(1)。但是如果要在指定位置 i 插入和删除元素的话（`add(int index, E element)`）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。 ② `LinkedList` 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 O（1）而数组为近似 O（n）。
- \4. 是否支持快速随机访问： `LinkedList` 不支持高效的随机元素访问，而 `ArrayList` 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于`get(int index)`方法)。
- \5. 内存空间占用： ArrayList的空 间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）。

1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。
2.对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。
3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。
尽量避免同时遍历和删除集合。因为这会改变集合的大小；

(三）ArrayList 与 Vector 区别呢?为什么要用Arraylist取代Vector呢?

`Vector`类的所有方法都是同步的。可以由两个线程安全地访问一个Vector对象、但是一个线程访问Vector的话代码要在同步操作上耗费大量的时间。

`Arraylist`不是同步的，所以在不需要保证线程安全时建议使用Arraylist。

(四）说一说 ArrayList 的扩容机制吧

https://github.com/Snailclimb/JavaGuide/blob/master/docs/java/collection/ArrayList-Grow.md

（五）HashSet与TreeSet与LinkedHashSet对比

HashSet不能保证元素的排列顺序，顺序有可能发生变化，不是同步的，集合元素可以是null,但只能放入一个null
TreeSet是SortedSet接口的唯一实现类，TreeSet可以确保集合元素处于排序状态。TreeSet支持两种排序方式，自然排序 和定制排序，其中自然排序为默认的排序方式。向 TreeSet中加入的应该是同一个类的对象。
TreeSet判断两个对象不相等的方式是两个对象通过equals方法返回false，或者通过CompareTo方法比较没有返回0
自然排序
自然排序使用要排序元素的CompareTo（Object obj）方法来比较元素之间大小关系，然后将元素按照升序排列。
定制排序
自然排序是根据集合元素的大小，以升序排列，如果要定制排序，应该使用Comparator接口，实现 int compare(To1,To2)方法
LinkedHashSet集合同样是根据元素的hashCode值来决定元素的存储位置，但是它同时使用链表维护元素的次序。这样使得元素看起 来像是以插入顺 序保存的，也就是说，当遍历该集合时候，LinkedHashSet将会以元素的添加顺序访问集合的元素。
LinkedHashSet在迭代访问Set中的全部元素时，性能比HashSet好，但是插入时性能稍微逊色于HashSet。

（六）LinkedHashMap和HashMap，TreeMap对比

Hashtable与 HashMap类似,它继承自Dictionary类，不同的是:它不允许记录的键或者值为空;它支持线程的同步，即任一时刻只有一个线程能写Hashtable,因此也导致了 Hashtable在写入时会比较慢。
Hashmap 是一个最常用的Map,它根据键的HashCode 值存储数据,根据键可以直接获取它的值，具有很快的访问速度，遍历时，**取得数据的顺序是完全随机的。**
LinkedHashMap保存了记录的插入顺序，**在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的.也可以在构造时用带参数，按照应用次数排序。在遍历的时候会比HashMap慢，不过有种情况例外，当HashMap容量很大，实际数据较少时，遍历起来可能会比LinkedHashMap慢，因为LinkedHashMap的遍历速度只和实际数据有关，和容量无关，而HashMap的遍历速度和他的容量有关。**
TreeMap实现SortMap接口，能够把它保存的记录根据键排序,默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator 遍历TreeMap时，得到的记录是排过序的。
我们用的最多的是HashMap,HashMap里面存入的键值对在取出的时候是随机的,在Map 中插入、删除和定位元素，HashMap 是最好的选择。
TreeMap取出来的是排序后的键值对。但如果您要按**自然顺序或自定义顺序遍历键**，那么TreeMap会更好。
LinkedHashMap 是HashMap的一个子类，如果需要输出的顺序和输入的相同,那么用LinkedHashMap可以实现,它还可以按读取顺序来排列，像连接池中可以应用。

（七）HashMap 和 Hashtable 的区别

1. **线程是否安全**： HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过`synchronized`修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；
2. **效率**： 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；

3. **对Null key 和Null value的支持**： HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。
4. **初始容量大小和每次扩充容量大小的不同**： ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小（HashMap 中的`tableSizeFor()`方法保证，下面给出了源代码）。也就是说 HashMap 总是使用2的幂作为哈希表的大小,后面会介绍到为什么是2的幂次方。
5. **底层数据结构**： JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

（八）HashMap 和 HashSet区别

如果你看过`HashSet`源码的话就应该知道：HashSet 底层就是基于 HashMap 实现的。（HashSet 的源码非常非常少，因为除了`clone()`、`writeObject()`、`readObject()`是 HashSet 自己不得不实现之外，其他方法都是直接调用 HashMap 中的方法。

HashMap	HashSet
实现了Map接口	实现Set接口
存储键值对	仅存储对象
调用`put()`向map中添加元素	调用`add()`方法向Set中添加元素
HashMap使用键（Key）计算Hashcode	HashSet使用成员对象来计算hashcode值，对于两个对象来说hashcode可能相同，所以equals()方法用来判断对象的相等性，

（九）HashSet如何检查重复

当你把对象加入`HashSet`时，HashSet会先计算对象的`hashcode`值来判断对象加入的位置，同时也会与其他加入的对象的hashcode值作比较，如果没有相符的hashcode，HashSet会假设对象没有重复出现。但是如果发现有相同hashcode值的对象，这时会调用`equals()`方法来检查hashcode相等的对象是否真的相同。如果两者相同，HashSet就不会让加入操作成功。（摘自我的Java启蒙书《Head fist java》第二版）

hashCode（）与equals（）的相关规定：

1. 如果两个对象相等，则hashcode一定也是相同的
2. 两个对象相等,对两个equals方法返回true
3. 两个对象有相同的hashcode值，它们也不一定是相等的
4. 综上，equals方法被覆盖过，则hashCode方法也必须被覆盖
5. hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该class的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

（十）HashMap的底层实现

JDK1.8之前

JDK1.8 之前 `HashMap` 底层是 数组和链表 结合在一起使用也就是 链表散列。HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值，然后通过 (n - 1) & hash 判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。

所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。

HashMap实现原理（比较好的描述）：HashMap以键值对（key-value）的形式来储存元素，但调用put方法时，HashMap会通过hash函数来计算key的hash值，然后通过hash值&(HashMap.length-1)判断当前元素的存储位置，如果当前位置存在元素的话，就要判断当前元素与要存入的key是否相同，如果相同则覆盖，如果不同则通过拉链表来解决。JDk1.8时，当链表长度大于8时，将链表转为红黑树。

JDK 1.8 HashMap 的 hash 方法源码:

JDK 1.8 的 hash方法 相比于 JDK 1.7 hash 方法更加简化，但是原理不变。

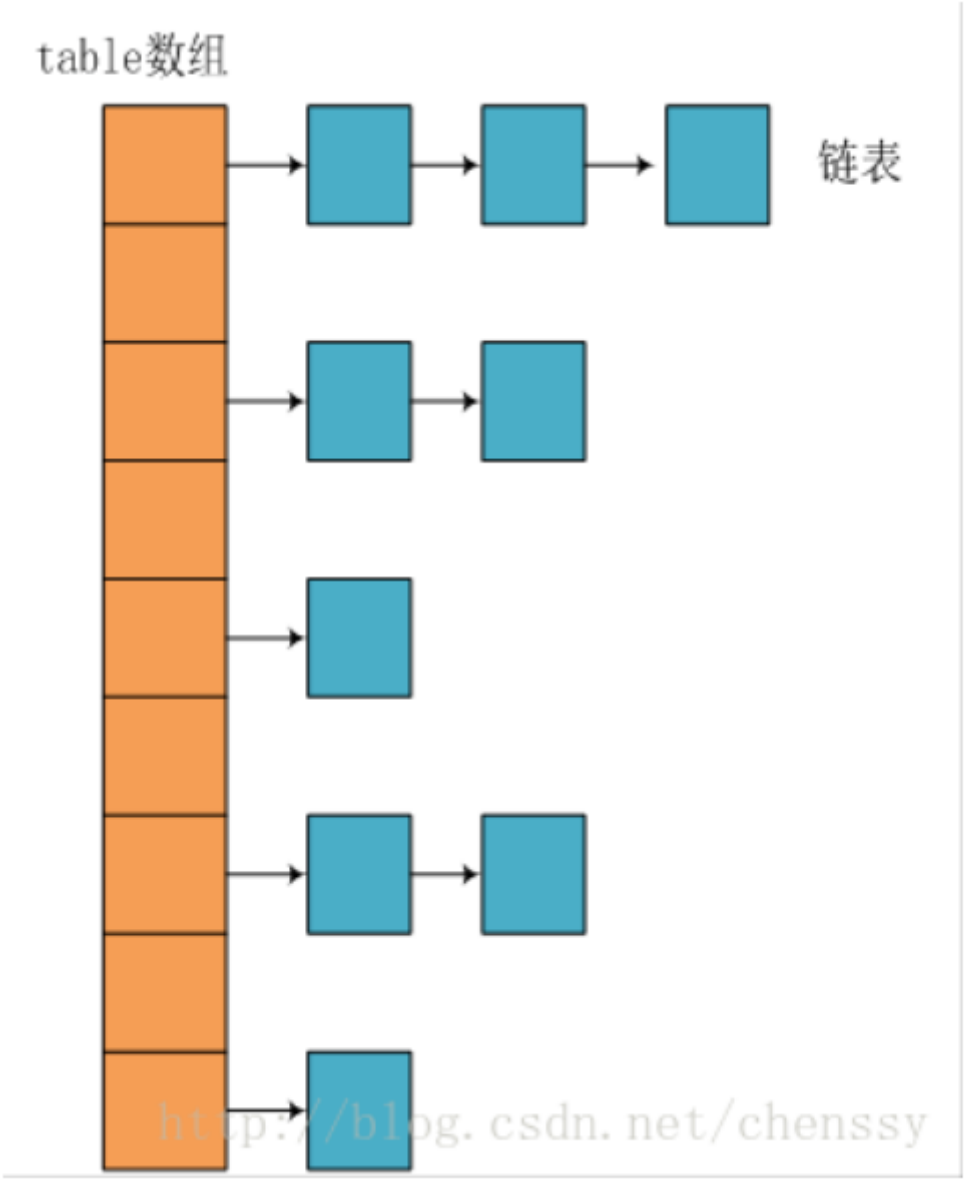
```
1     static final int hash(Object key) {
2       int h;
3       // key.hashCode()：返回散列值也就是hashcode
4       // ^ ：按位异或
5       // >>>:无符号右移，忽略符号位，空位都以0补齐
6       return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
7   }
```

对比一下 JDK1.7的 HashMap 的 hash 方法源码.

```
1 static int hash(int h) {
2     // This function ensures that hashCodes that differ only by
3     // constant multiples at each bit position have a bounded
4     // number of collisions (approximately 8 at default load factor).
5
6     h ^= (h >>> 20) ^ (h >>> 12);
7     return h ^ (h >>> 7) ^ (h >>> 4);
8 }
```

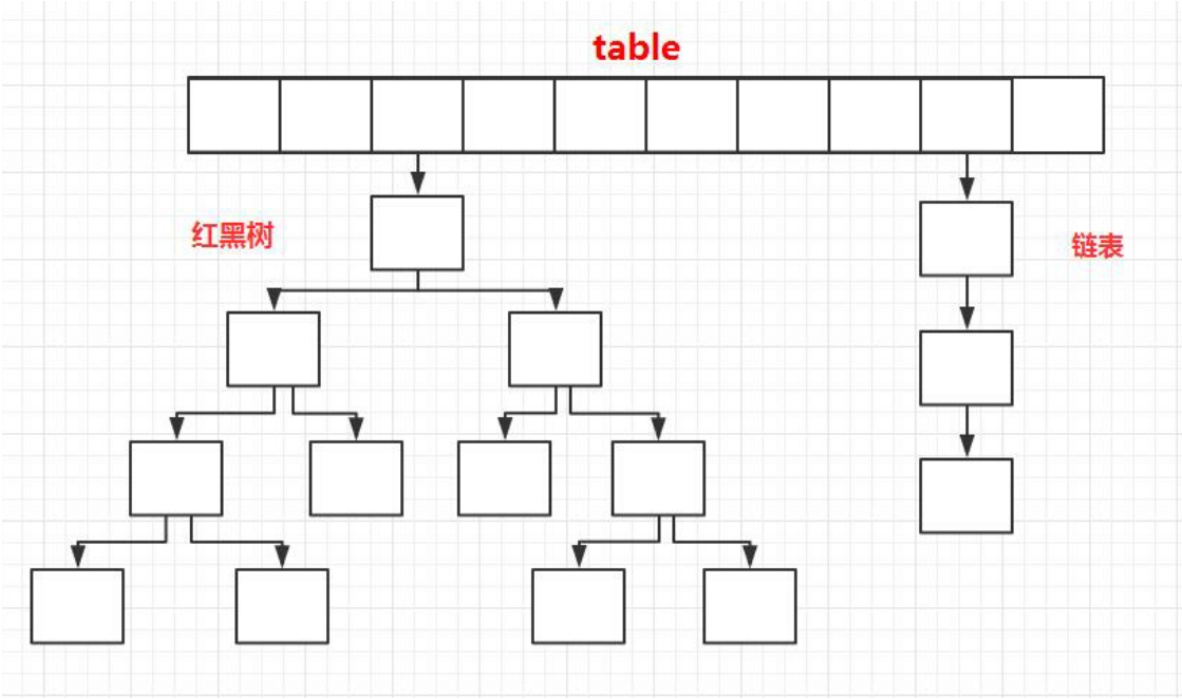
相比于 JDK1.8 的 hash 方法 ，JDK 1.7 的 hash 方法的性能会稍差一点点，因为毕竟扰动了 4 次。

所谓 “**拉链法**” 就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。



JDK1.8之后

相比于之前的版本， JDK1.8之后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。



TreeMap、TreeSet以及JDK1.8之后的HashMap底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

（十一）HashMap 的长度为什么是2的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值-2147483648到2147483647，前后加起来大概40亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个40亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“`(n - 1) & hash`”。（n代表数组长度）。这也就解释了 HashMap 的长度为什么是2的幂次方。

这个算法应该如何设计呢?

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：“取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说 hash%length==hash&(length-1)的前提是 length 是2的 n 次方；）。”并且 采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是2的幂次方。

（十二）HashMap 多线程操作导致死循环问题

主要原因在于 并发下的Rehash 会造成元素之间会形成一个循环链表。不过，jdk 1.8 后解决了这个问题，但是还是不建议在多线程下使用 HashMap,因为多线程下使用 HashMap 还是会存在其他问题比如数据丢失。并发环境下推荐使用 ConcurrentHashMap 。

Rehash：一般来说，Hash表这个容器当有数据要插入时，都会检查容量有没有超过设定的thredhold，如果超过，需要增大Hash表的尺寸，但是这样一来，整个Hash表里的无素都需要被重算一遍。这叫rehash，这个成本相当的大。

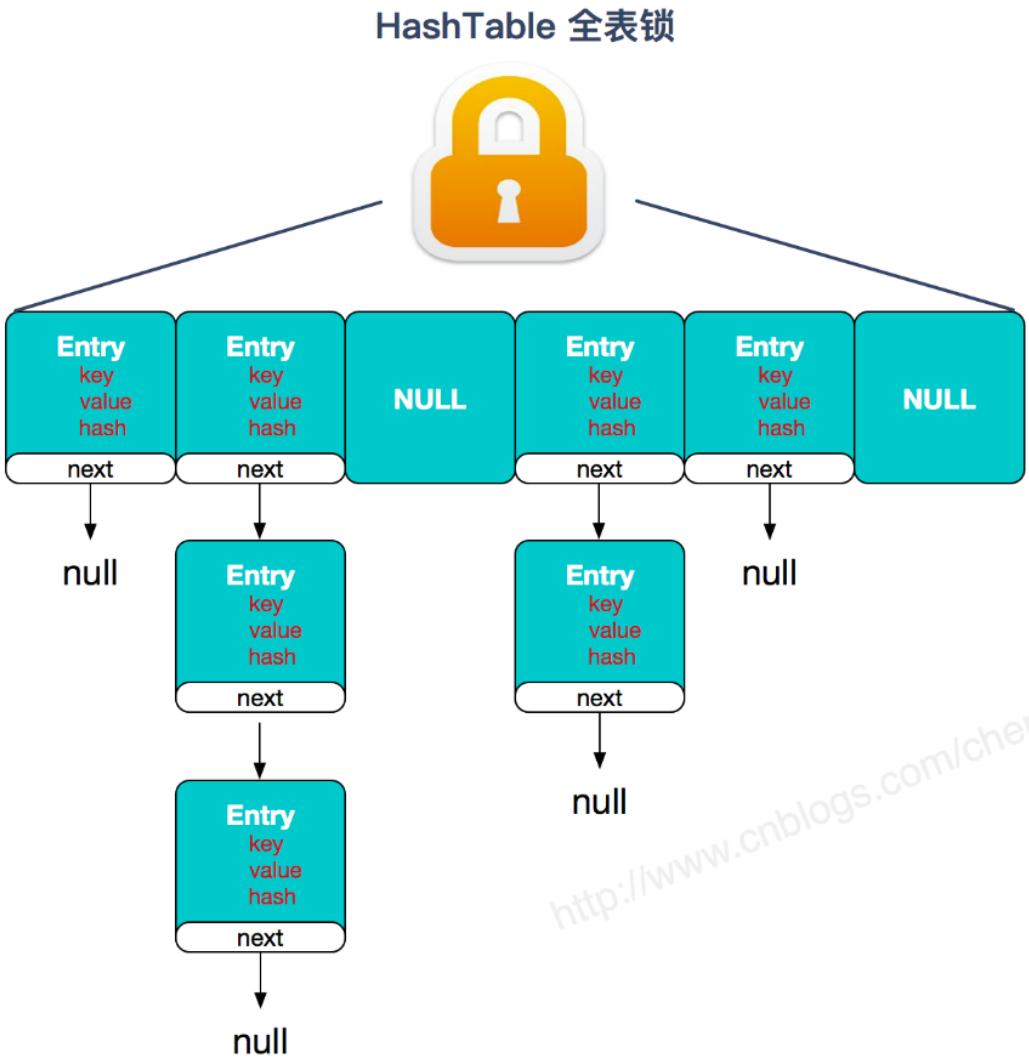
（十三）ConcurrentHashMap 和 Hashtable 的区别

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

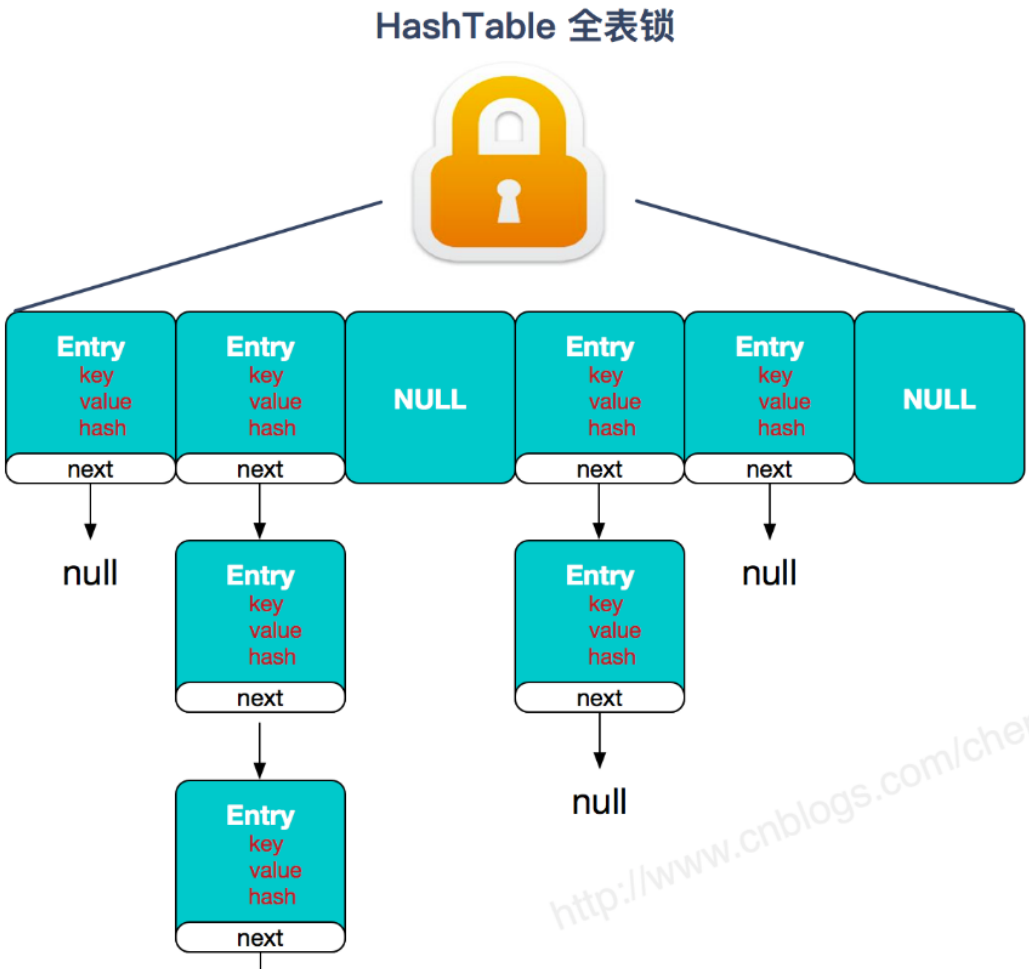
- **底层数据结构**： JDK1.7的 ConcurrentHashMap 底层采用 **分段的数组+链表** 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，**数组+链表/红黑二叉树。**Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 **数组+链表** 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；
- **实现线程安全的方式（重要）**： ① **在JDK1.7的时候，ConcurrentHashMap（分段锁）** 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。 **到了 JDK1.8 的时候已经摒弃了Segment的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。** （JDK1.6以后 对 synchronized锁做了很多优化） 整个看起来就像是优化过且线程安全的 HashMap，虽然在JDK1.8中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；② **Hashtable**(同一把锁) :**使用 synchronized 来保证线程安全，get/put所有相关操作都是synchronized的，这相当于给整个哈希表加了一把大锁,****效率非常低下**。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

两者的对比图：

HashTable:


HashTable 全表锁
Entry
key
value
hash
next
NULL
null

**

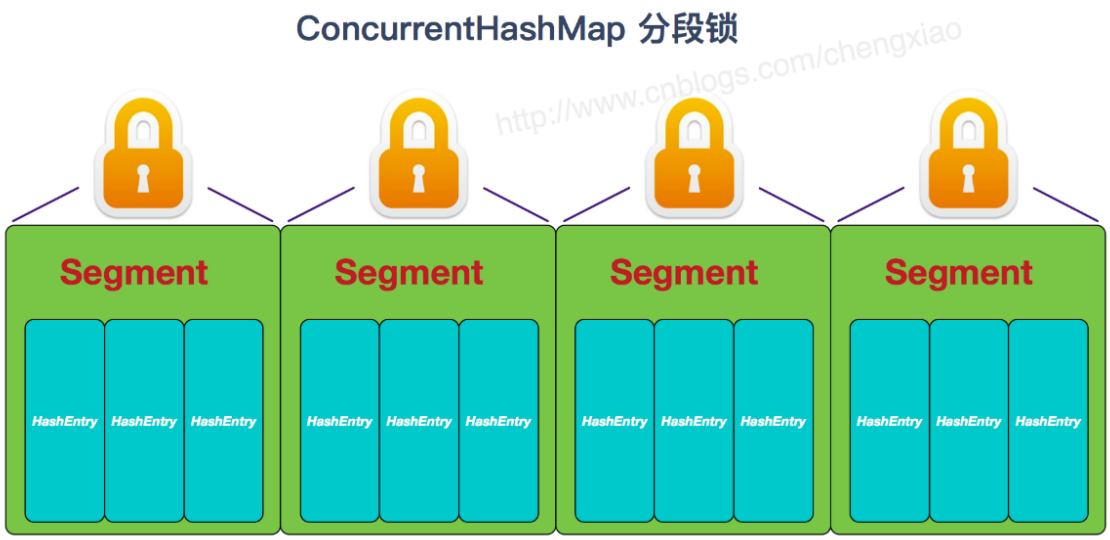

HashTable 全表锁
Entry
key
value
hash
next
NULL
null

↓

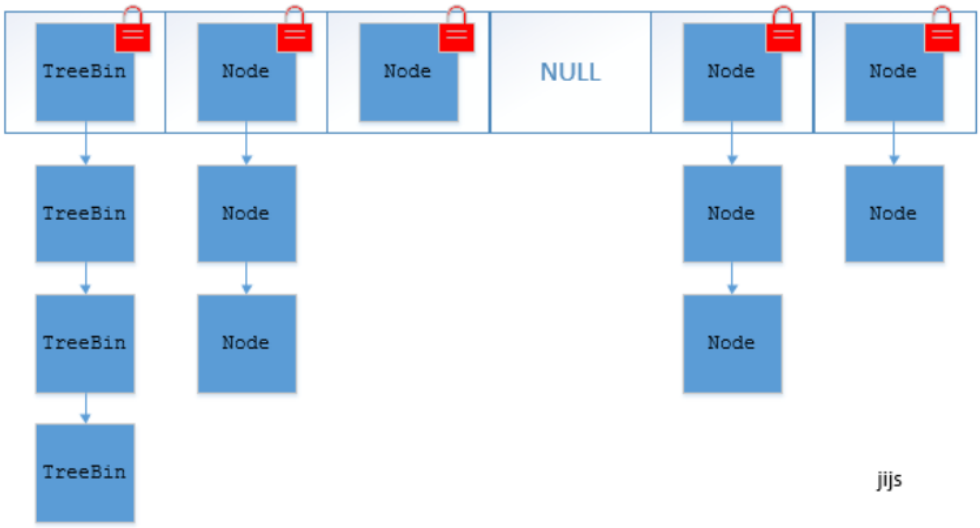
null

**

JDK1.7的ConcurrentHashMap：



JDK1.8的ConcurrentHashMap（TreeBin: 红黑二叉树节点 Node: 链表节点）：



（十四）ConcurrentHashMap线程安全的具体实现方式/底层具体实现

JDK1.7（上面有示意图）

首先将数据分为一段一段的存储，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时，其他段的数据也能被其他线程访问。

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。

Segment 实现了 ReentrantLock,所以 Segment 是一种可重入锁，扮演锁的角色。HashEntry 用于存储键值对数据。

```
static class Segment<K,V> extends ReentrantLock implements Serializable {
}
```

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 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，必须首先获得对应的 Segment的锁。

JDK1.8 （上面有示意图）

ConcurrentHashMap取消了Segment分段锁，采用CAS和synchronized来保证并发安全。数据结构跟HashMap1.8的结构类似，数组+链表/红黑二叉树。Java 8在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为O(N)）转换为红黑树（寻址时间复杂度为O(log(N))）

synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。

（十五）comparable 和 Comparator的区别

- comparable接口实际上是出自java.lang包 它有一个 `compareTo(Object obj)` 方法用来排序
- comparator接口实际上是出自 java.util 包它有一个`compare(Object obj1, Object obj2)`方法用来排序

一般我们需要对一个集合使用自定义排序时，我们就要重写`compareTo()`方法或`compare()`方法，当我们需要对某一个集合实现两种排序方式，比如一个song对象中的歌名和歌手名分别采用一种排序方法的话，我们可以重写`compareTo()`方法和使用自制的Comparator方法或者以两个Comparator来实现歌名排序和歌星名排序，第二种代表我们只能使用两个参数版的`Collections.sort()`.