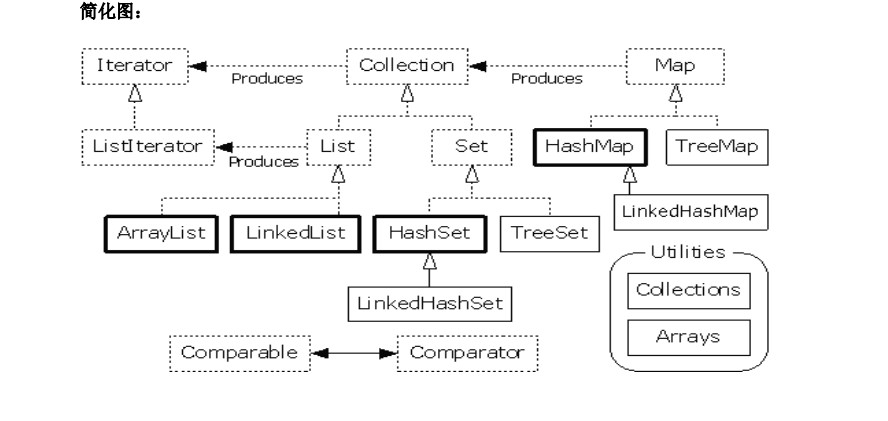
# 数据结构

# java集合框架（深入）

## 集合简化图



说明：对于以上的框架图有如下几点说明

1.所有集合类都位于java.util包下。Java的集合类主要由两个接口派生而出：Collection和Map，Collection和Map是Java集合框架的根接口，这两个接口又包含了一些子接口或实现类。

2. 集合接口：6个接口（短虚线表示），表示不同集合类型，是集合框架的基础。

3. 抽象类：5个抽象类（长虚线表示），对集合接口的部分实现。可扩展为自定义集合类。

4. 实现类：8个实现类（实线表示），对接口的具体实现。

5. Collection 接口是一组允许重复的对象。

6. Set 接口继承 Collection，集合元素不重复。

7. List 接口继承 Collection，允许重复，维护元素插入顺序。

8. Map接口是键－值对象，与Collection接口没有什么关系。

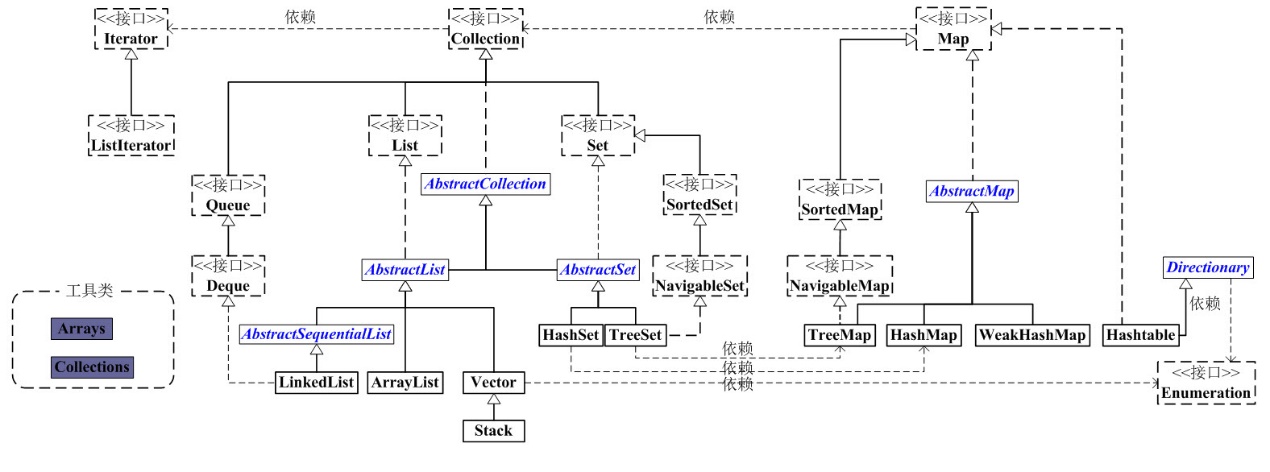
9.Set、List和Map可以看做集合的三大类：

List集合是有序集合，集合中的元素可以重复，访问集合中的元素可以根据元素的索引来访问。

Set集合是无序集合，集合中的元素不可以重复，访问集合中的元素只能根据元素本身来访问（也是集合里元素不允许重复的原因）。

Map集合中保存Key-value对形式的元素，访问时只能根据每项元素的key来访问其value。

## 总体分析



大致说明：

看上面的框架图，先抓住它的主干，即Collection和Map。

1、Collection是一个接口，是高度抽象出来的集合，它包含了集合的基本操作和属性。Collection包含了List和Set两大分支。

（1）List是一个有序的队列，每一个元素都有它的索引。第一个元素的索引值是0。List的实现类有LinkedList, ArrayList, Vector, Stack。

（2）Set是一个不允许有重复元素的集合。Set的实现类有HastSet和TreeSet。HashSet依赖于HashMap，它实际上是通过HashMap实现的；TreeSet依赖于TreeMap，它实际上是通过TreeMap实现的。

2、Map是一个映射接口，即key-value键值对。Map中的每一个元素包含“一个key”和“key对应的value”。AbstractMap是个抽象类，它实现了Map接口中的大部分API。而HashMap，TreeMap，WeakHashMap都是继承于AbstractMap。Hashtable虽然继承于Dictionary，但它实现了Map接口。

3、接下来，再看Iterator。它是遍历集合的工具，即我们通常通过Iterator迭代器来遍历集合。我们说Collection依赖于Iterator，是因为Collection的实现类都要实现iterator()函数，返回一个Iterator对象。ListIterator是专门为遍历List而存在的。

4、再看Enumeration，它是JDK 1.0引入的抽象类。作用和Iterator一样，也是遍历集合；但是Enumeration的功能要比Iterator少。在上面的框图中，Enumeration只能在Hashtable, Vector, Stack中使用。

5、最后，看Arrays和Collections。它们是操作数组、集合的两个工具类。

有了上面的整体框架之后，我们接下来对每个类分别进行分析。

## Collection 和 Map 的子接口及其实现类

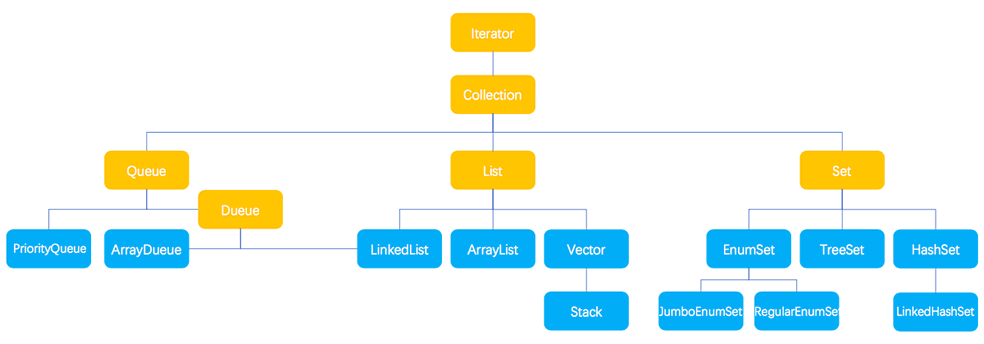


图 1  Collection接口基本结构

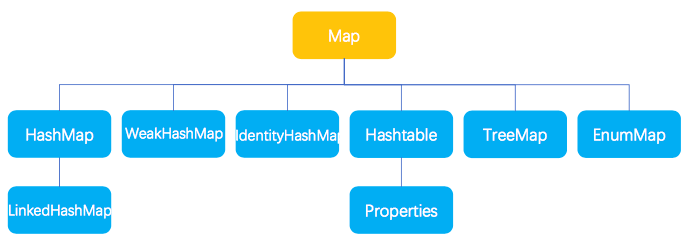


图 2 Map接口基本结构

 Java集合接口的作用

|  |  |
| --- | --- |
| **接口名称** | **作    用** |
| Iterator 接口 | 集合的输出接口，主要用于遍历输出（即迭代访问）Collection 集合中的元素，Iterator 对象被称之为迭代器。迭代器接口是集合接口的父接口，实现类实现 Collection 时就必须实现 Iterator 接口。 |
| Collection 接口 | 是 List、Set 和 Queue 的父接口，是存放一组单值的最大接口。所谓的单值是指集合中的每个元素都是一个对象。一般很少直接使用此接口直接操作。 |
| Queue 接口 | Queue 是 Java 提供的队列实现，有点类似于 List。 |
| Dueue 接口 | 是 Queue 的一个子接口，为双向队列。 |
| List 接口 | 是最常用的接口。是有序集合，允许有相同的元素。使用 List 能够精确地控制每个元素插入的位置，用户能够使用索引（元素在 List 中的位置，类似于数组下标）来访问 List 中的元素，与数组类似。 |
| Set 接口 | 不能包含重复的元素。 |
| Map 接口 | 是存放一对值的最大接口，即接口中的每个元素都是一对，以 key➡value 的形式保存。 |

表 2 Java集合实现类的作用

|  |  |
| --- | --- |
| **类名称** | **作用** |
| HashSet | 为优化査询速度而设计的 Set。它是基于 HashMap 实现的，HashSet 底层使用 HashMap 来保存所有元素，实现比较简单 |
| TreeSet | 实现了 Set 接口，是一个有序的 Set，这样就能从 Set 里面提取一个有序序列 |
| ArrayList | 一个用数组实现的 List，能进行快速的随机访问，效率高而且实现了可变大小的数组 |
| ArrayDueue | 是一个基于数组实现的双端队列，按“先进先出”的方式操作集合元素 |
| LinkedList | 对顺序访问进行了优化，但随机访问的速度相对较慢。此外它还有 addFirst()、addLast()、getFirst()、getLast()、removeFirst() 和 removeLast() 等方法，能把它当成栈（Stack）或队列（Queue）来用 |
| HsahMap | 按哈希算法来存取键对象 |
| TreeMap | 可以对键对象进行排序 |

## Collection接口

Collection接口是处理对象集合的根接口，其中定义了很多对元素进行操作的方法。Collection接口有两个主要的子接口List和Set，注意Map不是Collection的子接口，这个要牢记。

Collection接口中的方法如下:

|  |  |
| --- | --- |
| **方法名称** | **说明** |
| boolean add(E e) | 向集合中添加一个元素，如果集合对象被添加操作改变了，则返回 true。E 是元素的数据类型 |
| boolean addAll(Collection c) | 向集合中添加集合 c 中的所有元素，如果集合对象被添加操作改变了，则返回 true。 |
| void clear() | 清除集合中的所有元素，将集合长度变为 0。 |
| boolean contains(Object o) | 判断集合中是否存在指定元素 |
| boolean containsAll(Collection c) | 判断集合中是否包含集合 c 中的所有元素 |
| boolean isEmpty() | 判断集合是否为空 |
| Iterator<E>iterator() | 返回一个 Iterator 对象，用于遍历集合中的元素 |
| boolean remove(Object o) | 从集合中删除一个指定元素，当集合中包含了一个或多个元素 o 时，该方法只删除第一个符合条件的元素，该方法将返回 true。 |
| boolean removeAll(Collection c) | 从集合中删除所有在集合 c 中出现的元素（相当于把调用该方法的集合减去集合 c）。如果该操作改变了调用该方法的集合，则该方法返回 true。 |
| boolean retainAll(Collection c) | 从集合中删除集合 c 里不包含的元素（相当于把调用该方法的集合变成该集合和集合 c 的交集），如果该操作改变了调用该方法的集合，则该方法返回 true。 |
| int size() | 返回集合中元素的个数 |
| Object[] toArray() | 把集合转换为一个数组，所有的集合元素变成对应的数组元素。 |

其中，有几个比较常用的方法，比如方法add()添加一个元素到集合中，addAll()将指定集合中的所有元素添加到集合中，contains()方法检测集合中是否包含指定的元素，toArray()方法返回一个表示集合的数组。

另外，Collection中有一个iterator()函数，它的作用是返回一个Iterator接口。通常，我们通过Iterator迭代器来遍历集合。ListIterator是List接口所特有的，在List接口中，通过ListIterator()返回一个ListIterator对象。

Collection接口有两个常用的子接口，下面详细介绍。

### List接口

List集合代表一个有序集合，集合中每个元素都有其对应的顺序索引。List集合允许使用重复元素，可以通过索引来访问指定位置的集合元素。

List接口继承于Collection接口，它可以定义一个允许重复的有序集合。因为List中的元素是有序的，所以我们可以通过使用索引（元素在List中的位置，类似于数组下标）来访问List中的元素，这类似于Java的数组。

List接口为Collection直接接口。List所代表的是有序的Collection，即它用某种特定的插入顺序来维护元素顺序。用户可以对列表中每个元素的插入位置进行精确地控制，同时可以根据元素的整数索引（在列表中的位置）访问元素，并搜索列表中的元素。实现List接口的集合主要有：ArrayList、LinkedList、Vector、Stack。

#### ArrayList

ArrayList的常用方法：

|  |  |
| --- | --- |
| **方法名称** | **说明** |
| E get(int index) | 获取此集合中指定索引位置的元素，E 为集合中元素的数据类型 |
| int indexOf(Object o) | 返回此集合中第一次出现指定元素的索引，如果此集合不包含该元素，则返回 -1 |
| int lastIndexOf(Object o) | 返回此集合中最后一次出现指定元素的索引，如果此集合不包含该元素，则返回 -1 |
| E set(int index, Eelement) | 将此集合中指定索引位置的元素修改为 element 参数指定的对象。此方法返回此集合中指定索引位置的原元素 |
| List<E> subList(int fromlndex, int tolndex) | 返回一个新的集合，新集合中包含 fromlndex 和 tolndex 索引之间的所有元素。包含 fromlndex 处的元素，不包含 tolndex 索引处的元素 |

ArrayList是一个动态数组，也是我们最常用的集合。它允许任何符合规则的元素插入甚至包括null。每一个ArrayList都有一个初始容量（10），该容量代表了数组的大小。随着容器中的元素不断增加，容器的大小也会随着增加。在每次向容器中增加元素的同时都会进行容量检查，当快溢出时，就会进行扩容操作。所以如果我们明确所插入元素的多少，最好指定一个初始容量值，避免过多的进行扩容操作而浪费时间、效率。

size、isEmpty、get、set、iterator 和 listIterator 操作都以固定时间运行。add 操作以分摊的固定时间运行，也就是说，添加 n 个元素需要 O(n) 时间（由于要考虑到扩容，所以这不只是添加元素会带来分摊固定时间开销那样简单）。

ArrayList擅长于随机访问。同时ArrayList是非同步的。

#### LinkedList

同样实现List接口的LinkedList与ArrayList不同，ArrayList是一个动态数组，而LinkedList是一个双向链表。所以它除了有ArrayList的基本操作方法外还额外提供了get，remove，insert方法在LinkedList的首部或尾部。

由于实现的方式不同，LinkedList不能随机访问，它所有的操作都是要按照双重链表的需要执行。在列表中索引的操作将从开头或结尾遍历列表（从靠近指定索引的一端）。这样做的好处就是可以通过较低的代价在List中进行插入和删除操作。

与ArrayList一样，LinkedList也是非同步的。如果多个线程同时访问一个List，则必须自己实现访问同步。一种解决方法是在创建List时构造一个同步的List：

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

#### Vector

与ArrayList相似，但是Vector是同步的。所以说Vector是线程安全的动态数组。它的操作与ArrayList几乎一样。

‌Vector和ArrayList在Java中主要有以下几个区别‌：

‌线程安全性‌：

‌Vector‌是线程安全的，它的所有方法都被synchronized关键字修饰，确保在多线程环境中不会出现并发问题。这种同步机制虽然保证了线程安全，但也带来了额外的性能开销。‌

‌ArrayList‌是非线程安全的，如果在多线程环境中使用，需要进行额外的同步处理，否则可能会导致数据不一致或抛出异常（如ConcurrentModificationException）。‌

‌性能‌：

‌Vector‌由于所有操作都是同步的，性能相对较低，特别是在没有多线程要求的环境中，这种同步机制实际上会降低程序的执行效率。

‌ArrayList‌由于没有同步开销，性能通常优于Vector，特别是在单线程环境或对同步没有要求的多线程环境中。

‌扩容机制‌：

‌Vector‌在扩容时，默认将其容量增加一倍，这种策略在某些情况下减少了扩容次数，但可能导致内存浪费，特别是在大数据量场景下。‌

‌ArrayList‌在扩容时，通常将容量增加约1.5倍，这种渐进扩容策略可以减少扩容操作的频率，提高整体性能。

‌使用场景‌：

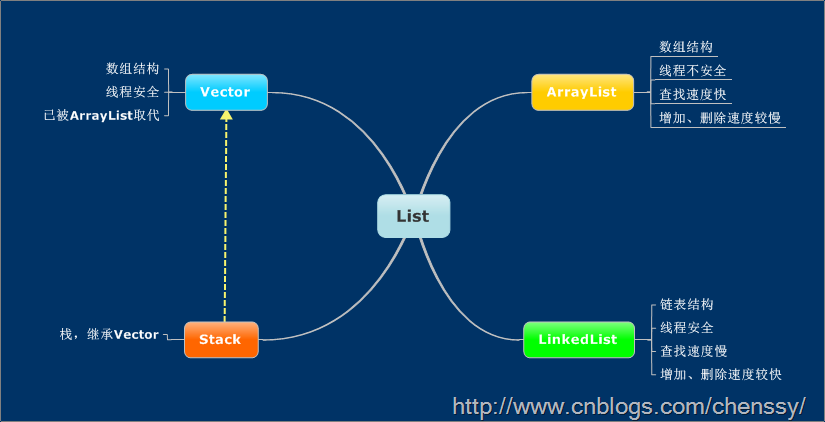
‌Vector‌适用于对线程安全性要求较高的场景，如多线程环境下的共享数据操作。‌

‌ArrayList‌则更适合在单线程或对性能要求较高的场景中使用。

#### Stack

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

|  |  |
| --- | --- |
| boolean empty() | 测试堆栈是否为空 |
| E push(E item) | 把项压入堆栈顶部 |
| E pop() | 移除堆栈顶部的对象，并作为此函数的值返回该对象 |
| E peek() | 查看堆栈顶部的对象，但不从堆栈中移除它 |
| int search(Object o) | 返回对象在堆栈中的位置，以1为基数 |



### Set接口

Set是一种不包括重复元素的Collection。它维持它自己的内部排序，所以随机访问没有任何意义。与List一样，它同样允许null的存在但是仅有一个。由于Set接口的特殊性，所有传入Set集合中的元素都必须不同，同时要注意任何可变对象，如果在对集合中元素进行操作时，导致e1.equals(e2)==true，则必定会产生某些问题。Set接口有三个具体实现类，分别是散列集HashSet、链式散列集LinkedHashSet和树形集TreeSet。

Set是一种不包含重复的元素的Collection，无序，即任意的两个元素e1和e2都有e1.equals(e2)=false，Set最多有一个null元素。需要注意的是:虽然Set中元素没有顺序，但是元素在set中的位置是由该元素的HashCode决定的，其具体位置其实是固定的。

此外需要说明一点，在set接口中的不重复是有特殊要求的。

举一个例子:对象A和对象B，本来是不同的两个对象，正常情况下它们是能够放入到Set里面的，但是如果对象A和B的都重写了hashcode和equals方法，并且重写后的hashcode和equals方法是相同的话。那么A和B是不能同时放入到Set集合中去的，也就是Set集合中的去重和hashcode与equals方法直接相关。

　　为了更好地理解，请看下面的例子：

public static void main(String[] args) {

Set<String> set = new HashSet<String>();  
 set.add("Hello");  
 set.add("world");  
 set.add("Hello");  
 System.*out*.println("集合的尺寸为:"+set.size());  
 System.*out*.println("集合中的元素为:"+set.toString());  
}

运行结果：

集合的尺寸为:2

集合中的元素为:[world, Hello]

分析：由于String类中重写了hashcode和equals方法，用来比较指向的字符串对象所存储的字符串是否相等。所以这里的第二个Hello是加不进去的。

再看一个例子：

public static void main(String[] args){

Set<String> books = new HashSet<String>();  
 //添加一个字符串对象  
 books.add(new String("Struts2权威指南"));  
 //再次添加一个字符串对象，  
 //因为两个字符串对象通过equals方法比较相等，所以添加失败，返回false  
 boolean result = books.add(new String("Struts2权威指南"));  
 System.*out*.println(result);  
 //下面输出看到集合只有一个元素  
 System.*out*.println(books);  
}

运行结果：

false

说明：程序中，book集合两次添加的字符串对象明显不是一个对象（程序通过new关键字来创建字符串对象），当使用==运算符判断返回false，使用equals方法比较返回true，所以不能添加到Set集合中，最后只能输出一个元素。

#### HashSet

HashSet 是一个没有重复元素的集合。它是由HashMap实现的，不保证元素的顺序(这里所说的没有顺序是指：元素插入的顺序与输出的顺序不一致)，而且HashSet允许使用null 元素。HashSet是非同步的，如果多个线程同时访问一个哈希set，而其中至少一个线程修改了该set，那么它必须保持外部同步。 HashSet按Hash算法来存储集合的元素，因此具有很好的存取和查找性能。

HashSet的实现方式大致如下，通过一个HashMap存储元素，元素是存放在HashMap的Key中，而Value统一使用一个Object对象。

HashSet使用和理解中容易出现的误区:

1. HashSet中存放null值

HashSet中是允许存入null值的，但是在HashSet中仅仅能够存入一个null值。

1. HashSet中存储元素的位置是固定的

HashSet中存储的元素的是无序的，这个没什么好说的，但是由于HashSet底层是基于Hash算法实现的，使用了hashcode，所以HashSet中相应的元素的位置是固定的。

1. 必须小心操作可变对象（Mutable Object）。

如果一个Set中的可变元素改变了自身状态导致Object.equals(Object)=true将导致一些问题。

#### LinkedHashSet

LinkedHashSet继承自HashSet，其底层是基于LinkedHashMap来实现的，有序，非同步。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。

#### TreeSet

TreeSet 类除了实现 Collection 接口的所有方法之外，还提供了如表 2 所示的方法

|  |  |
| --- | --- |
| 方法名称说明 | 说明 |
| E first() | 返回此集合中的第一个元素。其中，E 表示集合中元素的数据类型 |
| E last() | 返回此集合中的最后一个元素 |
| E pollFirst() | 获取并移除此集合中的第一个元素 |
| E pollLast() | 获取并移除此集合中的最后一个元素 |
| SortedSet<E> subSet(E fromElement,E toElement) | 返回一个新的集合，新集合包含原集合中 fromElement 对象与 toElement对象之间的所有对象。包含 fromElement 对象，不包含 toElement 对象 |
| SortedSet<E> headSet<E toElement〉 | 返回一个新的集合，新集合包含原集合中 toElement 对象之前的所有对象。不包含 toElement 对象 |
| SortedSet<E> tailSet(E fromElement) | 返回一个新的集合，新集合包含原集合中 fromElement 对象之后的所有对象。包含 fromElement 对象 |

TreeSet是一个有序集合，其底层是基于TreeMap实现的，非线程安全。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。当我们构造TreeSet时，若使用不带参数的构造函数，则TreeSet的使用自然比较器；若用户需要使用自定义的比较器，则需要使用带比较器的参数。

注意：TreeSet集合不是通过hashcode和equals函数来比较元素的.它是通过compare或者comparaeTo函数来判断元素是否相等.compare函数通过判断两个对象的id，相同的id判断为重复元素，不会被加入到集合中。

## Map接口

Map与List、Set接口不同，它是由一系列键值对组成的集合，提供了key到Value的映射。同时它也没有继承Collection。在Map中它保证了key与value之间的一一对应关系。也就是说一个key对应一个value，所以它不能存在相同的key值，当然value值可以相同。

Map接口常用方法

|  |  |
| --- | --- |
| **方法名称** | **说明** |
| void clear() | 删除该 Map 对象中的所有 key-value 对。 |
| boolean containsKey(Object key) | 查询 Map 中是否包含指定的 key，如果包含则返回 true。 |
| boolean containsValue(Object value) | 查询 Map 中是否包含一个或多个 value，如果包含则返回 true。 |
| V get(Object key) | 返回 Map 集合中指定键对象所对应的值。V 表示值的数据类型 |
| V put(K key, V value) | 向 Map 集合中添加键-值对，如果当前 Map 中已有一个与该 key 相等的 key-value 对，则新的 key-value 对会覆盖原来的 key-value 对。 |
| void putAll(Map m) | 将指定 Map 中的 key-value 对复制到本 Map 中。 |
| V remove(Object key) | 从 Map 集合中删除 key 对应的键-值对，返回 key 对应的 value，如果该 key 不存在，则返回 null |
| boolean remove(Object key, Object value) | 这是 [Java](http://m.biancheng.net/java/) 8 新增的方法，删除指定 key、value 所对应的 key-value 对。如果从该 Map 中成功地删除该 key-value 对，该方法返回 true，否则返回 false。 |
| Set entrySet() | 返回 Map 集合中所有键-值对的 Set 集合，此 Set 集合中元素的数据类型为 Map.Entry |
| Set keySet() | 返回 Map 集合中所有键对象的 Set 集合 |
| boolean isEmpty() | 查询该 Map 是否为空（即不包含任何 key-value 对），如果为空则返回 true。 |
| int size() | 返回该 Map 里 key-value 对的个数 |
| Collection values() | 返回该 Map 里所有 value 组成的 Collection |

### HashMap

以哈希表数据结构实现，查找对象时通过哈希函数计算其位置，它是为快速查询而设计的，其内部定义了一个hash表数组（Entry[] table），元素会通过哈希转换函数将元素的哈希地址转换成数组中存放的索引，如果有冲突，则使用散列链表的形式将所有相同哈希地址的元素串起来，可能通过查看HashMap.Entry的源码它是一个单链表结构。

Map的遍历方式：

1. 使用for循环遍历entrySet

for (Map.Entry<Integer, String> entry : map.entrySet()) {  
 int mapKey = entry.getKey();  
 String mapValue = entry.getValue();  
 System.*out*.println(mapKey + "：" + mapValue);  
}

1. 使用 for-each 循环遍历 key 或者 values，一般适用于只需要 Map 中的 key 或者 value 时使用。性能上比 entrySet 较好

// 打印键集合  
for (Integer key : map.keySet()) {  
 System.*out*.println(key);  
}  
// 打印值集合  
for (String value : map.values()) {  
 System.*out*.println(value);  
}

3. 使用迭代器（Iterator）遍历

Iterator<Map.Entry<Integer, String>> entries = map.entrySet().iterator();  
while (entries.hasNext()) {  
 Map.Entry<Integer, String> entry = entries.next();  
 int key = entry.getKey();  
 String value = entry.getValue();  
 System.*out*.println(key + ":" + value);  
}

4. 通过键找值遍历，这种方式的效率比较低，因为本身从键取值是耗时的操作

for(int key : map.keySet()){  
 String value = map.get(key);  
 System.*out*.println(key+":"+value);  
}

### LinkedHashMap

LinkedHashMap是HashMap的一个子类，它保留插入的顺序，如果需要输出的顺序和输入时的相同，那么就选用LinkedHashMap。

LinkedHashMap是Map接口的哈希表和链接列表实现，具有可预知的迭代顺序。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。

LinkedHashMap实现与HashMap的不同之处在于，前者维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序。

根据链表中元素的顺序可以分为：按插入顺序的链表，和按访问顺序(调用get方法)的链表。默认是按插入顺序排序，如果指定按访问顺序排序，那么调用get方法后，会将这次访问的元素移至链表尾部，不断访问可以形成按访问顺序排序的链表。

注意，此实现不是同步的。如果多个线程同时访问链接的哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须保持外部同步。

由于LinkedHashMap需要维护元素的插入顺序，因此性能略低于HashMap的性能，但在迭代访问Map里的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。

### TreeMap

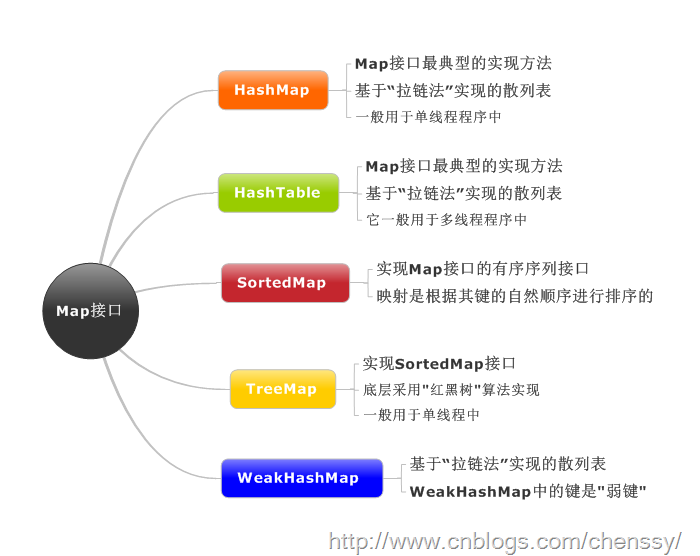
TreeMap 是一个有序的key-value集合，非同步，基于红黑树（Red-Black tree）实现，每一个key-value节点作为红黑树的一个节点。TreeMap存储时会进行排序的，会根据key来对key-value键值对进行排序，其中排序方式也是分为两种，一种是自然排序，一种是定制排序，具体取决于使用的构造方法。

自然排序：TreeMap中所有的key必须实现Comparable接口，并且所有的key都应该是同一个类的对象，否则会报ClassCastException异常。

定制排序：定义TreeMap时，创建一个comparator对象，该对象对所有的treeMap中所有的key值进行排序，采用定制排序的时候不需要TreeMap中所有的key必须实现Comparable接口。

TreeMap判断两个元素相等的标准：两个key通过compareTo()方法返回0，则认为这两个key相等。

如果使用自定义的类来作为TreeMap中的key值，且想让TreeMap能够良好的工作，则必须重写自定义类中的equals()方法，TreeMap中判断相等的标准是：两个key通过equals()方法返回为true，并且通过compareTo()方法比较应该返回为0。



## Iterator 与 ListIterator详解

### Iterator

Iterator的定义如下：

public interface Iterator<E> {}

Iterator是一个接口，它是集合的迭代器。集合可以通过Iterator去遍历集合中的元素。Iterator提供的API接口如下：

boolean hasNext()：判断集合里是否存在下一个元素。如果有，hasNext()方法返回 true。

Object next()：返回集合里下一个元素。

void remove()：删除集合里上一次next方法返回的元素。

使用示例：

ArrayList<String> a = new ArrayList<>();  
a.add("aaa");  
a.add("bbb");  
a.add("ccc");  
System.*out*.println("Before iterate : " + a);  
Iterator<String> it = a.iterator();  
while (it.hasNext()) {  
 String t = it.next();  
 if ("bbb".equals(t)) {  
 it.remove();  
 }  
}  
System.*out*.println("After iterate : " + a);

输出结果如下：

Before iterate : [aaa, bbb, ccc]  
After iterate : [aaa, ccc]

注意：

（1）Iterator只能单向移动。

（2）Iterator.remove()是唯一安全的方式来在迭代过程中修改集合；如果在迭代过程中以任何其它的方式修改了基本集合将会产生未知的行为。而且每调用一次next()方法，remove()方法只能被调用一次，如果违反这个规则将抛出一个异常。

### ListIterator

ListIterator是一个功能更加强大的迭代器, 它继承于Iterator接口,只能用于各种List类型的访问。可以通过调用listIterator()方法产生一个指向List开始处的ListIterator, 还可以调用listIterator(n)方法创建一个一开始就指向列表索引为n的元素处的ListIterator.

ListIterator接口定义如下:

public interface ListIterator<E> extends Iterator<E> {

boolean hasNext();  
 E next();  
 boolean hasPrevious();  
 E previous();  
 int nextIndex();  
 int previousIndex();  
 void remove();  
 void set(E e);  
 void add(E e);  
  
}

由以上定义我们可以推出ListIterator可以:

1. 双向移动（向前/向后遍历）.
2. 产生相对于迭代器在列表中指向的当前位置的前一个和后一个元素的索引.
3. 可以使用set()方法替换它访问过的最后一个元素.
4. 可以使用add()方法在next()方法返回的元素之前或previous()方法返回的元素之后插入一个元素.

使用示例：

ArrayList<String> a = new ArrayList<>();  
 a.add("aaa");  
 a.add("bbb");  
 a.add("ccc");  
 System.*out*.println("Before iterate : " + a);  
 ListIterator<String> it = a.listIterator();  
 while (it.hasNext()) {  
 System.*out*.println(it.next() + ", " + it.previousIndex() + ", " + it.nextIndex());  
 }  
 while (it.hasPrevious()) {  
 System.*out*.print(it.previous() + " ");  
 }  
 System.*out*.println();  
 it = a.listIterator(1);  
 while (it.hasNext()) {  
 String t = it.next();  
 System.*out*.println(t);  
 if ("ccc".equals(t)) {  
 it.set("nnn");  
 System.*out*.println("After iterate1 : " + a);  
 } else {  
 it.add("kkk");  
 System.*out*.println("After iterate2 : " + a);  
 }  
 }  
 System.*out*.println("After iterate : " + a);  
}

输出结果如下：

Before iterate : [aaa, bbb, ccc]  
aaa, 0, 1  
bbb, 1, 2  
ccc, 2, 3  
ccc bbb aaa   
bbb

After iterate2 : [aaa, bbb, kkk, ccc]  
ccc

After iterate1 : [aaa, bbb, kkk, nnn]  
After iterate : [aaa, bbb, kkk, nnn]

## 异同点

### ArrayList和LinkedList

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绝对优于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。

这一点要看实际情况的。若只对单条数据插入或删除，ArrayList的速度反而优于LinkedList。但若是批量随机的插入删除数据，LinkedList的速度大大优于ArrayList. 因为ArrayList每插入一条数据，要移动插入点及之后的所有数据。

### HashTable与HashMap

相同点：

1. 都实现了Map、Cloneable、java.io.Serializable接口。
2. 都是存储"键值对(key-value)"的散列表，而且都是采用拉链法实现的。

不同点：

1. 历史原因:HashTable是基于陈旧的Dictionary类的，HashMap是Java 1.2引进的Map接口的一个实现 。
2. 同步性:HashTable是线程安全的，也就是说是同步的，而HashMap是线程序不安全的，不是同步的 。
3. 对null值的处理：HashMap的key、value都可为null，HashTable的key、value都不可为null 。
4. 基类不同：HashMap继承于AbstractMap，而Hashtable继承于Dictionary。

Dictionary是一个抽象类，它直接继承于Object类，没有实现任何接口。Dictionary类是JDK 1.0的引入的。虽然Dictionary也支持“添加key-value键值对”、“获取value”、“获取大小”等基本操作，但它的API函数比Map少；而且Dictionary一般是通过Enumeration(枚举类)去遍历，Map则是通过Iterator(迭代M器)去遍历。 然而由于Hashtable也实现了Map接口，所以，它即支持Enumeration遍历，也支持Iterator遍历。

AbstractMap是一个抽象类，它实现了Map接口的绝大部分API函数；为Map的具体实现类提供了极大的便利。它是JDK 1.2新增的类。

1. 支持的遍历种类不同：HashMap只支持Iterator(迭代器)遍历。而Hashtable支持Iterator(迭代器)和Enumeration(枚举器)两种方式遍历。

### HashMap、Hashtable、LinkedHashMap和TreeMap比较

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。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。

Hashtable 与 HashMap类似，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Hashtale在写入时会比较慢。

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。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。LinkedHashMap实现与HashMap的不同之处在于，后者维护着一个运行于所有条目的双重链表。此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序。对于LinkedHashMap而言，它继承与HashMap、底层使用哈希表与双向链表来保存所有元素。其基本操作与父类HashMap相似，它通过重写父类相关的方法，来实现自己的链接列表特性。

TreeMap实现SortMap接口，内部实现是红黑树。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。TreeMap不允许key的值为null。非同步的。

一般情况下，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，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。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。

### HashSet、LinkedHashSet、TreeSet比较

#### Set接口

Set不允许包含相同的元素，如果试图把两个相同元素加入同一个集合中，add方法返回false。

Set判断两个对象相同不是使用==运算符，而是根据equals方法。也就是说，只要两个对象用equals方法比较返回true，Set就不会接受这两个对象。

#### HashSet

HashSet有以下特点：

1. 不能保证元素的排列顺序，顺序有可能发生变化。
2. 不是同步的。
3. 集合元素可以是null，但只能放入一个null。

当向HashSet结合中存入一个元素时，HashSet会调用该对象的hashCode()方法来得到该对象的hashCode值，然后根据 hashCode值来决定该对象在HashSet中存储位置。简单的说，HashSet集合判断两个元素相等的标准是两个对象通过equals方法比较相等，并且两个对象的hashCode()方法返回值也相等。

注意，如果要把一个对象放入HashSet中，重写该对象对应类的equals方法，也应该重写其hashCode()方法。其规则是如果两个对象通过equals方法比较返回true时，其hashCode也应该相同。另外，对象中用作equals比较标准的属性，都应该用来计算 hashCode的值。

#### LinkedHashSet

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。

#### TreeSet类

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。

Java提供了一个Comparable接口，该接口里定义了一个compareTo(Object obj)方法，该方法返回一个整数值，实现了该接口的对象就可以比较大小。obj1.compareTo(obj2)方法如果返回0，则说明被比较的两个对象相等，如果返回一个正数，则表明obj1大于obj2，如果是负数，则表明obj1小于obj2。如果我们将两个对象的equals方法总是返回true，则这两个对象的compareTo方法返回应该返回0。

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(T o1,T o2)方法。

### Iterator和ListIterator区别

我们在使用List，Set的时候，为了实现对其数据的遍历，我们经常使用到了Iterator(迭代器)。使用迭代器，你不需要干涉其遍历的过程，只需要每次取出一个你想要的数据进行处理就可以了。但是在使用的时候也是有不同的。List和Set都有iterator()来取得其迭代器。对List来说，你也可以通过listIterator()取得其迭代器，两种迭代器在有些时候是不能通用的，Iterator和ListIterator主要区别在以下方面：

（1）ListIterator有add()方法，可以向List中添加对象，而Iterator不能

（2）ListIterator和Iterator都有hasNext()和next()方法，可以实现顺序向后遍历，但是ListIterator有hasPrevious()和previous()方法，可以实现逆向（顺序向前）遍历。Iterator就不可以。

（3）ListIterator可以定位当前的索引位置，nextIndex()和previousIndex()可以实现。Iterator没有此功能。

（4）都可实现删除对象，但是ListIterator可以实现对象的修改，set()方法可以实现。Iierator仅能遍历，不能修改。

因为ListIterator的这些功能，可以实现对LinkedList等List数据结构的操作。其实，数组对象也可以用迭代器来实现。

### Collection 和 Collections区别

（1）java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List与Set。

（2）java.util.Collections 是一个包装类（工具类/帮助类）。它包含有各种有关集合操作的静态多态方法。此类不能实例化，就像一个工具类，用于对集合中元素进行排序、搜索以及线程安全等各种操作，服务于Java的Collection框架。

## Comparator 比较器

Java提供了一个Compatator比较器，这个类是一个接口，该接口里面有一个compare方法：

int compare(T o1, T o2);

实现Comparator接口，并实现compare方法

class MyComparator implements Comparator<A> {  
  
 @Override  
 public int compare(A o1, A o2) {  
 //升序  
 //return o1.a - o2.a;  
 //降序：后面会具体分析为什么降序  
 return o2.a - o1.a;  
 }  
  
}

compare返回值：

0: 表示o1和o2 相等

-1：表示o1 小于 o2，表示不需要交换o1和o2的位置

1：表示o1 大于 o2，表示需要交换o1和o2的位置

一般来说使用o1 – o2表示升序，使用o2 – o1表示降序。

## 解决线程安全问题

1. 使用Collections.synchronizedList（）定义一个集合类

## 其他

### ConcurrentLinkedQueue

一个基于链接节点的无界线程安全队列。此队列按照 FIFO（先进先出）原则对元素进行排序。队列的头部 是队列中时间最长的元素。队列的尾部 是队列中时间最短的元素。

新的元素插入到队列的尾部，队列获取操作从队列头部获得元素。当多个线程共享访问一个公共 collection 时，ConcurrentLinkedQueue 是一个恰当的选择。此队列不允许使用 null 元素。

|  |  |
| --- | --- |
| **方法摘要** | |
| boolean | [add](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html#add%28E%29)([E](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html) e)            将指定元素插入此队列的尾部。 |
| boolean | [contains](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html#contains%28java.lang.Object%29)([Object](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/Object.html) o)            如果此队列包含指定元素，则返回 true。 |
| boolean | [isEmpty](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html#isEmpty%28%29)()            如果此队列不包含任何元素，则返回 true。 |
| [Iterator](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/Iterator.html)<[E](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html)> | [iterator](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html#iterator%28%29)()            返回在此队列元素上以恰当顺序进行迭代的迭代器。 |
| boolean | [offer](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html#offer%28E%29)([E](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html) e)            将指定元素插入此队列的尾部。 |
| [E](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html) | [peek](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html#peek%28%29)()            获取但不移除此队列的头；如果此队列为空，则返回 null。 |
| [E](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html) | [poll](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html#poll%28%29)()            获取并移除此队列的头，如果此队列为空，则返回 null。 |
| boolean | [remove](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html#remove%28java.lang.Object%29)([Object](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/Object.html) o)            从队列中移除指定元素的单个实例（如果存在）。 |
| int | [size](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html#size%28%29)()            返回此队列中的元素数量。 |
| [Object](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/lang/Object.html)[] | [toArray](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html#toArray%28%29)()            返回以恰当顺序包含此队列所有元素的数组。 |
| |  | | --- | | <T> T[] | | [toArray](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/ConcurrentLinkedQueue.html#toArray%28T%5B%5D%29)(T[] a)            返回以恰当顺序包含此队列所有元素的数组；返回数组的运行时类型是指定数组的运行时类型。 |

数据结构

# 树

## 二叉查找树

二叉查找树（BST， Binary Search Tree ），也称二叉排序树（ Binary Sort Tree ），或二叉搜索树。

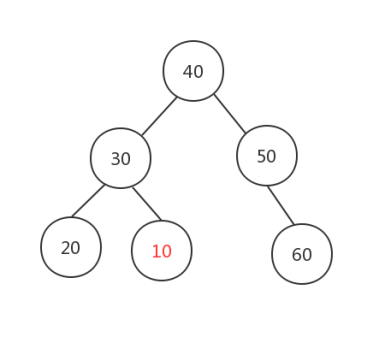
定义：一颗二叉树，满足以下属性

左子树的所有的值小于根节点的值

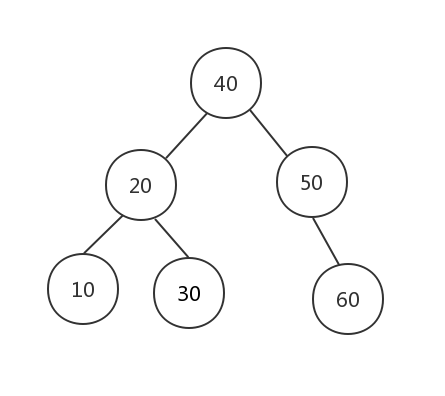
右子树的所有的值大于根节点的值

左、右子树满足以上两点

下面不是二叉查找树：



下面是二叉查找树



## 红黑树

红黑树（R-B Tree，Red-Black Tree）它一种特殊的二叉查找树，同时具备以下特征：

节点非红即黑

根节点是黑色

所有NUll节点称为叶子节点，且认为颜色为黑

所有红色节点的子节点都为黑色

从任一节点到其叶子节点的所有路径上都包含相同数目的黑节点

从根到叶子的最长的路径不多于最短的可能路径的两倍长

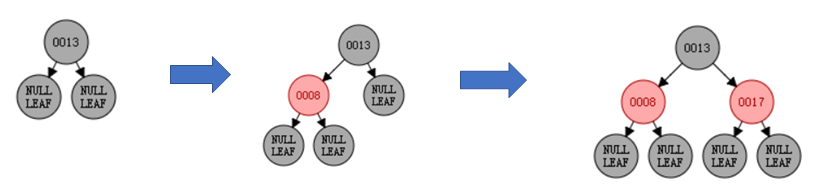
### 红黑树的插入操作

红黑树性质1：根节点是黑色

红黑树性质5：从任一节点到其叶子节点的所有路径上都包含相同数目的黑节点

插入原则：因为插入节点的颜色如果为黑肯定破坏红黑树性质5，所以每次插入的点首先都是红结点

情况1：插入的新节点N位于树的根上、插入的新节点的父节点是黑色



### 红黑树的插入后的操作——变色

红黑树性质1：根节点是黑色

红黑树性质4：所有红色节点的子节点都为黑色

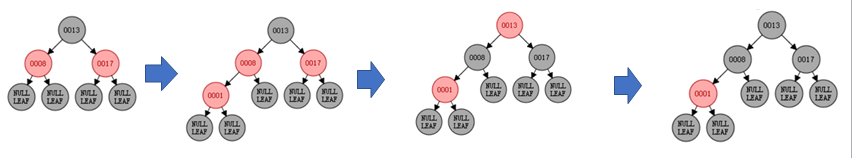
红黑树性质5：从任一节点到其叶子节点的所有路径上都包含相同数目的黑节点

叔父节点来指新节点的父节点的兄弟节点

祖父节点指新节点的父节点的父节点

情况2：如果新节点的父节点(0008)和叔父节点(0017)都是红色节点，

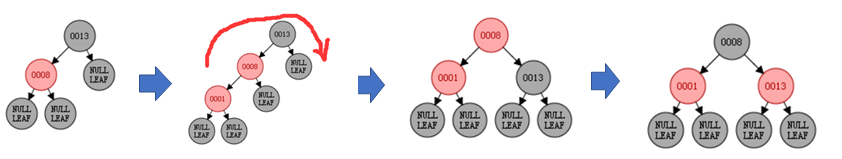
先插入新节点(红色)，（变色）新节点的父节点、叔父节点、祖父节点都需要变色



### 红黑树的插入后的操作——单次旋转

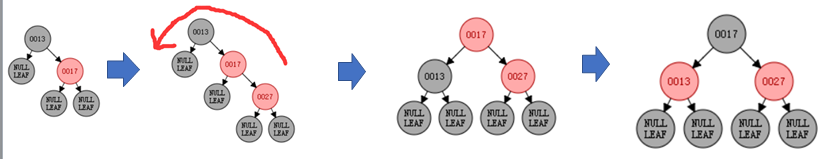
情况3：如果新节点的父节点是红色同时叔父节点都是黑色，同时新节点是其父节点的左子节点而父节点又是其父节点的左子节点。

我们进行一次右旋转调换新节点和其父节点的角色（以父节点为轴）



情况4：如果新节点的父节点是红色同时叔父节点都是黑色，同时新节点是其父节点的右子节点而父节点又是其父节点的右子节点。

我们对祖父节点进行一次左旋转调换新节点和其父节点的角色（以父节点为轴）

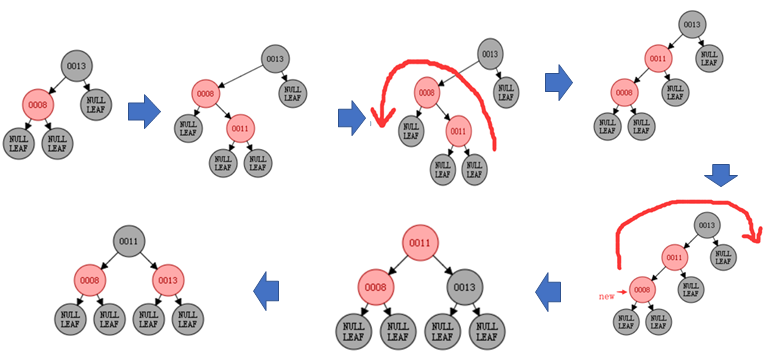


### 红黑树的插入后的操作——两次旋转

情况5：如果新节点的父节点是红色同时叔父节点都是黑色，同时新节点是其父节点的右子节点而父节点又是其父节点的左子节点。

我们进行一次左旋转调换新节点和其父节点的角色（第一次旋转），

同时我发现节点（0008）符合情况3，再进行一次右旋转（第二次旋转）

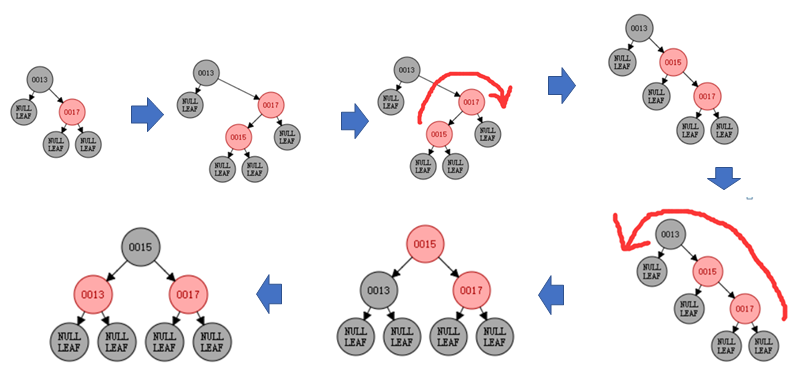


### 红黑树的插入后的操作——两次旋转2

情况6：如果新节点的父节点是红色同时叔父节点都是黑色，同时新节点是其父节点的左子节点而父节点又是其父节点的右子节点。

我们进行一次右旋转调换新节点和其父节点的角色（第一次旋转），

同时我发现节点（0017）符合情况4，再进行一次左旋转（第二次旋转）



### 操作总结图

|  |  |  |  |
| --- | --- | --- | --- |
| 父节点 | 叔节点 | 类型 | 操作 |
| 黑 |  |  | 无需操作 |
| 红 | 红 |  | 父叔都变黑，祖父变红，祖父变成当前节点，递归这个规则 |
| 红 | 黑 | 左左 | 右旋+变色 |
| 红 | 黑 | 右右 | 左旋+变色 |
| 红 | 黑 | 左右 | 先左旋，再右旋+变色 |
| 红 | 黑 | 右左 | 先右旋，再左旋+变色 |

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5