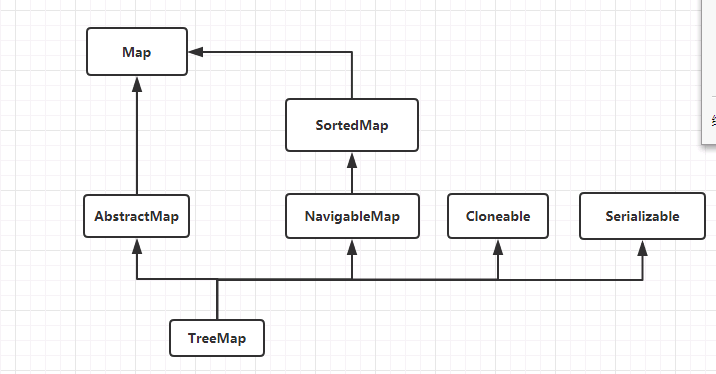
# TreeMap

TreeMap使用红黑树来作为存储结构，保存的数据具有顺序性，排序是根据实现Comparable接口的key或者自定义实现Comparetor接口的比较器来进行排序的，使用哪种排序方式是初始化TreeMap对象时是否传入自定义的比较器；如果没有传入比较器就使用key的compareTo方法来进行排序，如果key没有实现Comparable接口，并且没有传入比较器，那么添加过程就会抛出异常；如果key实现了Comparable接口并且也传入了比较器，默认使用比较器；TreeMap类比HashMap性能要差一些，TreeMap实现的Map集合中的映射关系是根据键值对象按一定顺序排列的，因此key不能为null；value可以为null

treeMap是有序的，排序规则是根据Comparetor或者Comparable来决定的



AbstractMap :实现类库了Map接口，作为Map的骨架实现，实现了Map的部分方法，也就是说为继承他的子类提供了Map的公共实现方法

当实现不可修改的Map结构时，只需要扩展此类的entrySet方法即可；当实现可修改的子类是，子类需要重写put等方法（在AbstractMap中put方法要抛出不支持操作异常）

SortedMap :所有已知的实现类TreeMap,ConcurrentSkipMap;为实现它的子类提供了排序方案，增加了获取集合定位的方法

SortedMap提供的方法

NavigableMap :继承SortdMap接口；具有给定搜索目标返回最接近匹配对象的导航方法，返回给定目标范围内的对象

方法 lowerEntry、floorEntry、ceilingEntry 和 higherEntry 分别返回与小于、小于等于、大于等于、大于给定键的键关联的 Map.Entry 对象，如果不存在这样的键，则返回 null，类似地，方法 lowerKey、floorKey、ceilingKey 和 higherKey 只返回关联的键。所有这些方法是为查找条目而不是遍历条目而设计的

## 成员变量

Comparator<? **super** K> comparator(); 接收比较器（实现Comparator接口的类）

SortedMap<K,V> subMap(K fromKey, K toKey);返回from和to之间的节点

1. SortedMap<K,V> headMap(K toKey);在k之前的节点

SortedMap<K,V> tailMap(K fromKey);返回key之后的节点

K firstKey();返回第一个key

K lastKey();返回最后一个key

Set<K> keySet();返回键的set集合

Collection<V> values();返回value的集合

Set<Map.Entry<K, V>> entrySet();获取Map中内部节点对象（Entry对象）的集合

## 构造方法

无参构造函数，没有指定比较器，就要去key必须实现Comparable接口

**public** TreeMap() {

comparator = **null**;

}

参数为实现comparetor接口的比较器，判断key的大小用这个比较器

**public** TreeMap(Comparator<? **super** K> comparator) {

**this**.comparator = comparator;

}

## 内部节点对象

内部节点对象Entry，实现Map.Entry接口

树节点的类结构

**static** **final** **class** Entry<K,V> **implements** Map.Entry<K,V> {

K key; 存放key

V value; 存放value

Entry<K,V> left; 指向左子树

Entry<K,V> right;指向右子树

Entry<K,V> parent;指向父节点

**boolean** color = *BLACK*;节点颜色（默认是黑色）

}

## put方法

添加新节点，从root节点开始；如果key已经存在，那么就替换原来的value，并返回原来的value；添加新节点，那么就是返回null

1. 当root为null时

校验key是否能够进行比较排序：判读是否初始化了Comparetor比较器或者key是否实现了Comparable接口；

Comare方法就是校验方法：内部通过 调用比较方法看是否抛出异常来判断

**final** **int** compare(Object k1, Object k2) {

**return** comparator==**null** ? ((Comparable<? **super** K>)k1).compareTo((K)k2)

: comparator.compare((K)k1, (K)k2);

}

校验通过后；创建新节点，指向root；root = **new** Entry<>(key, value, **null**);

1. Root不为null，判断是否初始化了比较器

Comparator<? **super** K> cpr = comparator;取出比较器判断是否为空；

**if** (cpr != **null**)，不为空有比较器，为空没有比较器

2.1有初始化TreeMap时有指定比较器

通过比较器来判断新增节点是左子树还是右子树：cmp = cpr.compare(key, t.key);

当cmp小于0 ，表示应该在父节点的左边，大于0，表示应该在父节点的右边；key的父节点相等，替换节点value即可

从root节点开始依次遍历判断新增节点所在的位置，当遍历到最后没有节点和key相等时，根据判断的节点属性（就是比较key后节点应该是左子树还是右子树）新增对应的子节点，然后调整树结构，满足红黑树属性 ；

遍历过程

**do** {

parent = t;

cmp = cpr.compare(key, t.key);

**if** (cmp < 0)

t = t.left;

**else** **if** (cmp > 0)

t = t.right;

**else**

**return** t.setValue(value);

} **while** (t != **null**);

2.2初始化时没有指定比较器

执行过程基本和指定比较器的一致，只是在比较key大小用了不同的方法，遍历判断方式都是一样的流程

1. 新增节点

判断节点属性后创建节点，将节点放到指定的位置

Entry<K,V> e = **new** Entry<>(key, value, parent);

**if** (cmp < 0)

parent.left = e;

**else**

parent.right = e;

1. 调整树结构的属性

fixAfterInsertion(e);调整新增后的树结构，让其满足红黑树属性

**public** V put(K key, V value) {

Entry<K,V> t = root;

**if** (t == **null**) {

compare(key, key); // type (and possibly null) check

root = **new** Entry<>(key, value, **null**);

size = 1;

modCount++;

**return** **null**;

}

**int** cmp;

Entry<K,V> parent;

// split comparator and comparable paths

Comparator<? **super** K> cpr = comparator;

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

**do** {

parent = t;

cmp = cpr.compare(key, t.key);

**if** (cmp < 0)

t = t.left;

**else** **if** (cmp > 0)

t = t.right;

**else**

**return** t.setValue(value);

} **while** (t != **null**);

}

**else** {

**if** (key == **null**)

**throw** **new** NullPointerException();

@SuppressWarnings("unchecked")

Comparable<? **super** K> k = (Comparable<? **super** K>) key;

**do** {

parent = t;

cmp = k.compareTo(t.key);

**if** (cmp < 0)

t = t.left;

**else** **if** (cmp > 0)

t = t.right;

**else**

**return** t.setValue(value);

} **while** (t != **null**);

}

Entry<K,V> e = **new** Entry<>(key, value, parent);

**if** (cmp < 0)

parent.left = e;

**else**

parent.right = e;

fixAfterInsertion(e);

size++;

modCount++;

**return** **null**;

}

## Get()方法

Get()方法，获取key对应的value

首先找到key对应的节点，获取节点后返回节点的value

1. 查找节点的key

通过getEntry方法查找节点

从root节点开始遍历，判断节点的key和参数key是否相等，不相等根据比较的结果指定下个比较的是左子节点还是右子节点；重复比较，知道key和节点的key相等就是要找到的节点；

以下是没有比较器是的遍历过程，和有比较器的遍历过程是一样的只是在判断key大小时是用比较器对象来完成的

Entry<K,V> p = root;

**while** (p != **null**) {

**int** cmp = k.compareTo(p.key);

**if** (cmp < 0)

p = p.left;

**else** **if** (cmp > 0)

p = p.right;

**else**

**return** p;

}

1. 返回value

获取节点后返回value：如果节点为null，那么返回null

**final** Entry<K,V> getEntry(Object key) {

// Offload comparator-based version for sake of performance

**if** (comparator != **null**)

**return** getEntryUsingComparator(key);

**if** (key == **null**)

**throw** **new** NullPointerException();

@SuppressWarnings("unchecked")

Comparable<? **super** K> k = (Comparable<? **super** K>) key;

Entry<K,V> p = root;

**while** (p != **null**) {

**int** cmp = k.compareTo(p.key);

**if** (cmp < 0)

p = p.left;

**else** **if** (cmp > 0)

p = p.right;

**else**

**return** p;

}

**return** **null**;

}

**public** V get(Object key) {

Entry<K,V> p = getEntry(key);

**return** (p==**null** ? **null** : p.value);

}

Remove：删除key对应的节点，如果删除的节点存在，remove方法返回这个节点的value，节点不存在返回null

获取节点：调用getEntry来获取，节点存在执行删除，删除红黑树节点流程在deleteEntry中完成

**public** V remove(Object key) {

Entry<K,V> p = getEntry(key);

**if** (p == **null**)

**return** **null**;

V oldValue = p.value;

deleteEntry(p);

**return** oldValue;

}

## 获取内部信息的方法

keySet();返回treeMap的所有的key的集合

keyset方法内部调用navigableKeySet方法来返回集合对象

集合对象为内部类KeySet，KeySet类实现了接口Iterable,重写了iterator方法，返回指定的迭代对象

KeyIterator，KeyIterator类实现接口Iterator，重写了hashNext,next等方法在遍历时调用；

**public** Set<K> keySet() {

**return** navigableKeySet();

}

**public** NavigableSet<K> navigableKeySet() {

KeySet<K> nks = navigableKeySet;

**return** (nks != **null**) ? nks : (navigableKeySet = **new** KeySet<>(**this**));

}

values();返回treemap的所有的value的接口

entrySet();返回treemap的所有映射节点的集合

这2个方法的实现原理和keyset一样