

深入 Java 集合学习系列(二): HashSet 的实现原理

1. HashSet 概述

HashSet 实现 Set 接口, 由哈希表(实际上是一个 HashMap 实例)支持。它不保证 set 的迭代顺序; 特别是它不保证该顺序恒久不变。此类允许使用 null 元素。

2. HashSet 的实现

对于 HashSet 而言, 它是基于 HashMap 实现的, HashSet 底层使用 HashMap 来保存所有元素, 因此 HashSet 的实现比较简单, 相关 HashSet 的操作, 基本上都是直接调用底层 HashMap 的相关方法来完成, HashSet 的源代码如下:

Java 代码 ☆

```
1. public class HashSet<E>
2.     extends AbstractSet<E>
3.     implements Set<E>, Cloneable, java.io.Serializable
4. {
5.     static final long serialVersionUID = -5024744406713321676L;
6.
7.     // 底层使用 HashMap 来保存 HashSet 中所有元素。
8.     private transient HashMap<E, Object> map;
9.
10.    // 定义一个虚拟的 Object 对象作为 HashMap 的 value, 将此对象定义为 static final。
11.    private static final Object PRESENT = new Object();
12.
13.    /**
14.     * 默认的空参构造器, 构造一个空的 HashSet。
15.     *
16.     * 实际底层会初始化一个空的 HashMap, 并使用默认初始容量为 16 和加载因子 0.75。
17.     */
18.    public HashSet() {
19.        map = new HashMap<E, Object>();
20.    }
21.
```

```
22.    /**
23.     * 构造一个包含指定 collection 中的元素的新 set。
24.     *
25.     * 实际底层使用默认的加载因子 0.75 和足以包含指定
26.     * collection 中所有元素的初始容量来创建一个 HashMap。
27.     * @param c 其中的元素将存放在此 set 中的 collection。
28.     */
29.    public HashSet(Collection<? extends E> c) {
30.        map = new HashMap<E, Object>(Math.max((int) (c.size()/.75f) + 1, 16));
31.        addAll(c);
32.    }
33.
34.    /**
35.     * 以指定的 initialCapacity 和 loadFactor 构造一个空的 HashSet。
36.     *
37.     * 实际底层以相应的参数构造一个空的 HashMap。
38.     * @param initialCapacity 初始容量。
39.     * @param loadFactor 加载因子。
40.     */
41.    public HashSet(int initialCapacity, float loadFactor) {
42.        map = new HashMap<E, Object>(initialCapacity, loadFactor);
43.    }
44.
45.    /**
46.     * 以指定的 initialCapacity 构造一个空的 HashSet。
47.     *
48.     * 实际底层以相应的参数及加载因子 loadFactor 为 0.75 构造一个空的 HashMap。
49.     * @param initialCapacity 初始容量。
50.     */
51.    public HashSet(int initialCapacity) {
52.        map = new HashMap<E, Object>(initialCapacity);
53.    }
54.
55.    /**
56.     * 以指定的 initialCapacity 和 loadFactor 构造一个新的空链接哈希集合。
57.     * 此构造函数为包访问权限，不对外公开，实际只是是对 LinkedHashSet 的支持。
58.     *
59.     * 实际底层会以指定的参数构造一个空 LinkedHashMap 实例来实现。
60.     * @param initialCapacity 初始容量。
61.     * @param loadFactor 加载因子。
62.     * @param dummy 标记。
63.     */
```

```
64.    HashSet(int initialCapacity, float loadFactor, boolean dummy) {
65.        map = new LinkedHashMap<E, Object>(initialCapacity, loadFactor);
66.    }
67.
68.    /**
69.     * 返回对此 set 中元素进行迭代的迭代器。返回元素的顺序并不是特定的。
70.     *
71.     * 底层实际调用底层 HashMap 的 keySet 来返回所有的 key。
72.     * 可见 HashSet 中的元素，只是存放在了底层 HashMap 的 key 上，
73.     * value 使用一个 static final 的 Object 对象标识。
74.     * @return 对此 set 中元素进行迭代的 Iterator。
75.     */
76.    public Iterator<E> iterator() {
77.        return map.keySet().iterator();
78.    }
79.
80.    /**
81.     * 返回此 set 中的元素的数量（set 的容量）。
82.     *
83.     * 底层实际调用 HashMap 的 size()方法返回 Entry 的数量，就得到该 Set 中元素的个数。
84.     * @return 此 set 中的元素的数量（set 的容量）。
85.     */
86.    public int size() {
87.        return map.size();
88.    }
89.
90.    /**
91.     * 如果此 set 不包含任何元素，则返回 true。
92.     *
93.     * 底层实际调用 HashMap 的 isEmpty()判断该 HashSet 是否为空。
94.     * @return 如果此 set 不包含任何元素，则返回 true。
95.     */
96.    public boolean isEmpty() {
97.        return map.isEmpty();
98.    }
99.
100.    /**
101.     * 如果此 set 包含指定元素，则返回 true。
102.     * 更确切地讲，当且仅当此 set 包含一个满足(o==null ? e==null : o.equals(e))
103.     * 的 e 元素时，返回 true。
104.     *
105.     * 底层实际调用 HashMap 的 containsKey 判断是否包含指定 key。
```

```
106.      * @param o 在此 set 中的存在已得到测试的元素。
107.      * @return 如果此 set 包含指定元素，则返回 true。
108.      */
109.      public boolean contains(Object o) {
110.          return map.containsKey(o);
111.      }
112.
113.      /**
114.       * 如果此 set 中尚未包含指定元素，则添加指定元素。
115.       * 更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2))
116.       * 的元素 e2，则向此 set 添加指定的元素 e。
117.       * 如果此 set 已包含该元素，则该调用不更改 set 并返回 false。
118.       *
119.       * 底层实际将将该元素作为 key 放入 HashMap。
120.       * 由于 HashMap 的 put()方法添加 key-value 对时,当新放入 HashMap 的 Entry 中 key
121.       * 与集合中原有 Entry 的 key 相同 (hashCode()返回值相等，通过 equals 比较也返回 true),
122.       * 新添加的 Entry 的 value 会将覆盖原来 Entry 的 value，但 key 不会有任何改变，
123.       * 因此如果向 HashSet 中添加一个已经存在的元素时，新添加的集合元素将不会被放入 H
124.       * 原来的元素也不会有任何改变，这也就满足了 Set 中元素不重复的特性。
125.       * @param e 将添加到此 set 中的元素。
126.       * @return 如果此 set 尚未包含指定元素，则返回 true。
127.       */
128.      public boolean add(E e) {
129.          return map.put(e, PRESENT)!=null;
130.      }
131.
132.      /**
133.       * 如果指定元素存在于此 set 中，则将其移除。
134.       * 更确切地讲，如果此 set 包含一个满足(o==null ? e==null : o.equals(e))的元素
135.       * 则将其移除。如果此 set 已包含该元素，则返回 true
136.       * （或者：如果此 set 因调用而发生更改，则返回 true）。（一旦调用返回，则此 set 不再
137.       * 包含该元素）。
138.       *
139.       * 底层实际调用 HashMap 的 remove 方法删除指定 Entry。
140.       * @param o 如果存在于此 set 中则需要将其移除的对象。
141.       * @return 如果 set 包含指定元素，则返回 true。
142.       */
143.      public boolean remove(Object o) {
144.          return map.remove(o)==PRESENT;
```

```
144.     }
145.
146.     /**
147.      * 从此 set 中移除所有元素。此调用返回后，该 set 将为空。
148.      *
149.      * 底层实际调用 HashMap 的 clear 方法清空 Entry 中所有元素。
150.      */
151.     public void clear() {
152.         map.clear();
153.     }
154.
155.     /**
156.      * 返回此 HashSet 实例的浅表副本：并没有复制这些元素本身。
157.      *
158.      * 底层实际调用 HashMap 的 clone() 方法，获取 HashMap 的浅表副本，并设置到 HashSe
159.      * t 中。
160.      */
161.     public Object clone() {
162.         try {
163.             HashSet<E> newSet = (HashSet<E>) super.clone();
164.             newSet.map = (HashMap<E, Object>) map.clone();
165.             return newSet;
166.         } catch (CloneNotSupportedException e) {
167.             throw new InternalError();
168.         }
169.     }
```

3. 相关说明

1) 相关 HashMap 的实现原理，请参考我的上一遍总结：深入 Java 集合学习系列：**HashMap 的实现原理**。

2) 对于 HashSet 中保存的对象，请注意正确重写其 equals 和 hashCode 方法，以保证放入的对象的唯一性。