java.util.Map

`Map` 接口不是 `Collection` 的子接口，使用键、值映射表来存储

```
1 public interface Map<K,V>
```

- `Map` 不能有重复的键(覆盖)，每个键可以映射到最多一个值
- 允许将映射内容视为一组键、值集合或键值映射集合
- `key` 不要求有序，不可以重复。 `value` 也不要求有序，但可以重复
- 当使用对象作为 `key` 时，要重写 `equals` 和 `hashCode` 方法

抽象方法：

方法	返回值	描述
clear()	void	删除所有的映射
containsKey(Object key)	boolean	Map中是有没有这个key
containsValue(Object value)	boolean	Map有没有这个value
entrySet()	Set<Map.Entry<K,V>>	返回包含的映射的Set
get(Object key)	V	根据key返回对应的value
isEmpty()	boolean	Map是不是空的
keySet()	Set<K>	返回Map中所有key的Set
put(K key, V value)	V	向Map添加映射
putAll(Map<? extends K,? extends V> m)	void	将指定Map中的映射复制到此映射
remove(Object key)	V	如果key存在，删除映射
remove(Object key, Object value)	boolean	当key、value映射存在时，删除
replace(K key, V value)	boolean	当key存在时，替换内容

方法	返回值	描述
size()	int	Map中映射的数量
values()	Collection<V>	返回所有value的集合

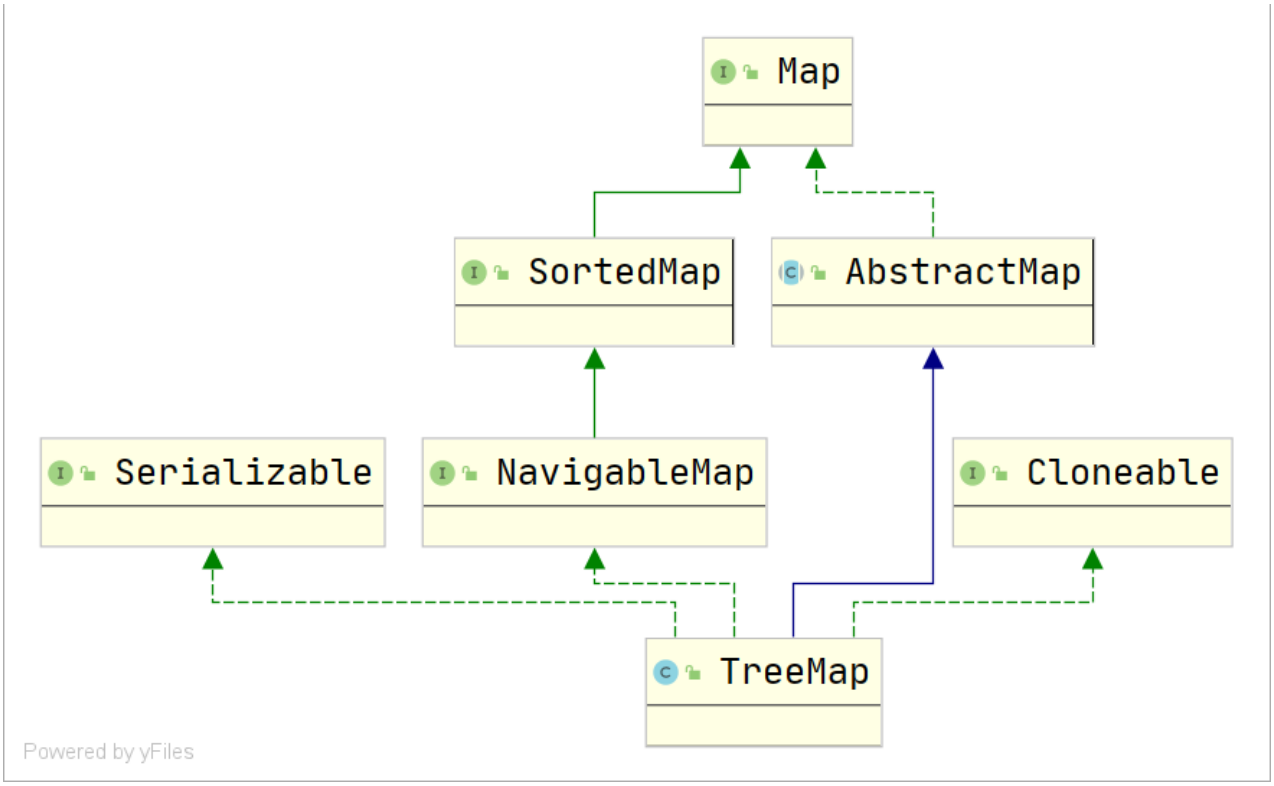
```
`JDK9` 提供了创建不可修改 Map 对象的方法:
    1. Map.of
    2. Map.ofEntries
    3. Map.copyOf
```

`Map` 的实现类较多，在此我们关注 `HashMap` 、 `TreeMap` 、 `HashTable` 、 `LinkedHashMap`

TreeMap

```
public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

- 继承 `AbstractMap` ,一个红黑树基于 `NavigableMap` 实现
- 非线程安全的
- `key` 不能存 `null` ,但是 `value` 可以存 `null`
- `key` 必须是可比较的 (实现 `Comparable` 接口，传递一个 `Comparator` 比较器)



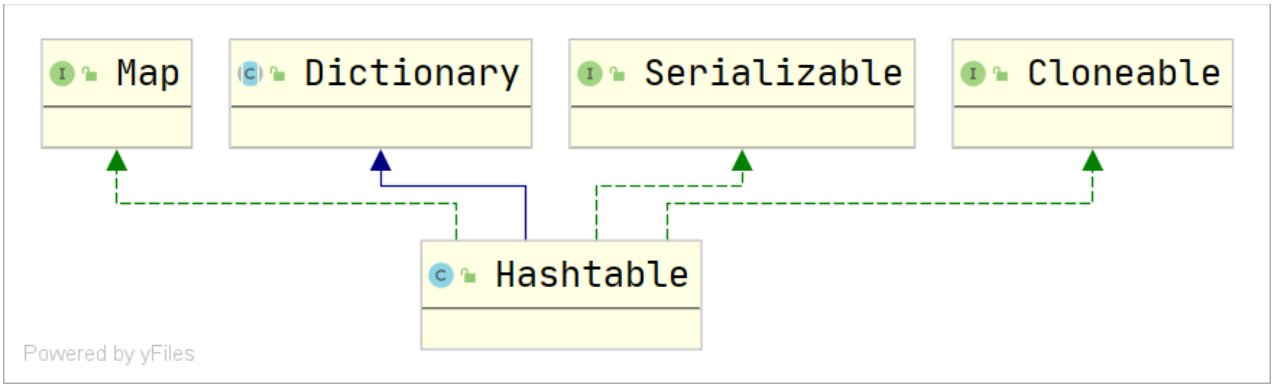
方法

(略)

Hashtable

```
public class Hashtable<K,V>
extends Dictionary<K,V>
implements Map<K,V>, Cloneable, Serializable
```

- 该类实现了一个哈希表，它将键映射到值
- 不允许 `null` 作为键和值
- 默认初始容量(`initialCapacity`)为 `11` ，默认负载因子(`loadFactor`)为 `0.75f`
- 同步的(线程安全的)
- 不保证顺序
- 扩容方式是旧容量的2倍 +1
  - 为什么hashtable的扩容方式选择为2n+1
  - 为了均匀分布，降低冲突率
- 数组 + 链表方式存储实现Hash表存储
- 添加值时
  - 如果 `hash` 一样 `equals` 为 `false` 则将当前值添加到链表头
  - 如果 `hash` 一样 `equals` 为 `true` 则使用当前值替换原来的值 （ `key` 相同）



构造方法

方法	描述
Hashtable()	构造一个新的，空的散列表，默认初始容量（11）和负载因子（0.75）。
Hashtable(int initialCapacity)	构造一个新的，空的哈希表，具有指定的初始容量和默认负载因子（0.75）。
Hashtable(int initialCapacity, float loadFactor)	构造一个新的，空的哈希表，具有指定的初始容量和指定的负载因子
Hashtable(Map<? extends K,? extends V> t)	构造一个与给定Map相同的映射的新哈希表

常用方法，大多实现自 `Map` 所以不重复讲解

方法名	返回值	描述
toString()	String	返回此 Hashtable对象的字符串表示形式，其括在大括号中，并以ASCII字符“ , ”（逗号和空格）分隔

`Hashtable` put 的过程

```
public synchronized V put(K key, V value) {
    // 值不能为 null
    if (value == null) {
        throw new NullPointerException();
    }

    // 确认 key 不在 hashtable 中存在
    Entry<?,?> tab[] = table;
    // 计算 key 的 hash
    int hash = key.hashCode();
    // 找 key 应在存放在 数组中的位置
    int index = (hash & 0x7FFFFFFF) % tab.length;
    // index 位置的元素的 key 和 当前的 key 不一样
    @SuppressWarnings("unchecked")
    Entry<K,V> entry = (Entry<K,V>)tab[index];
    for(; entry != null ; entry = entry.next) {
        // 判断 key 是否相同
        if ((entry.hash == hash) && entry.key.equals(key)) {
            // 当 key 一样时，替换值
            V old = entry.value;
```

```
            entry.value = value;
            return old;
        }
    }
    // 当 key 在 hashtable 中不存在时，添加
    addEntry(hash, key, value, index);
    return null;
}

private void addEntry(int hash, K key, V value, int index) {
    Entry<?,?> tab[] = table;
    // 判断是否需要 "扩容"
    if (count >= threshold) {
        // 对数组进行扩容 （2n + 1），将原有元素重新计算 hash
        rehash();

        tab = table;
        // 将当前的 key 也重新计算 hash
        hash = key.hashCode();
        index = (hash & 0x7FFFFFFF) % tab.length;
    }

    // 原来数组中的 entry 对象
    @SuppressWarnings("unchecked")
    Entry<K,V> e = (Entry<K,V>) tab[index];
    // 创建一个 新的 entry 对象，  放到 数组中
    tab[index] = new Entry<>(hash, key, value, e);
    // 元素个数 +1
    count++;
    // 修改次数 +1
    modCount++;
}
```

HashMap

```
public class HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

- 基于哈希表的实现的 `Map` 接口
- 允许 `null` 的值和 `null` 键
- 非线程安全

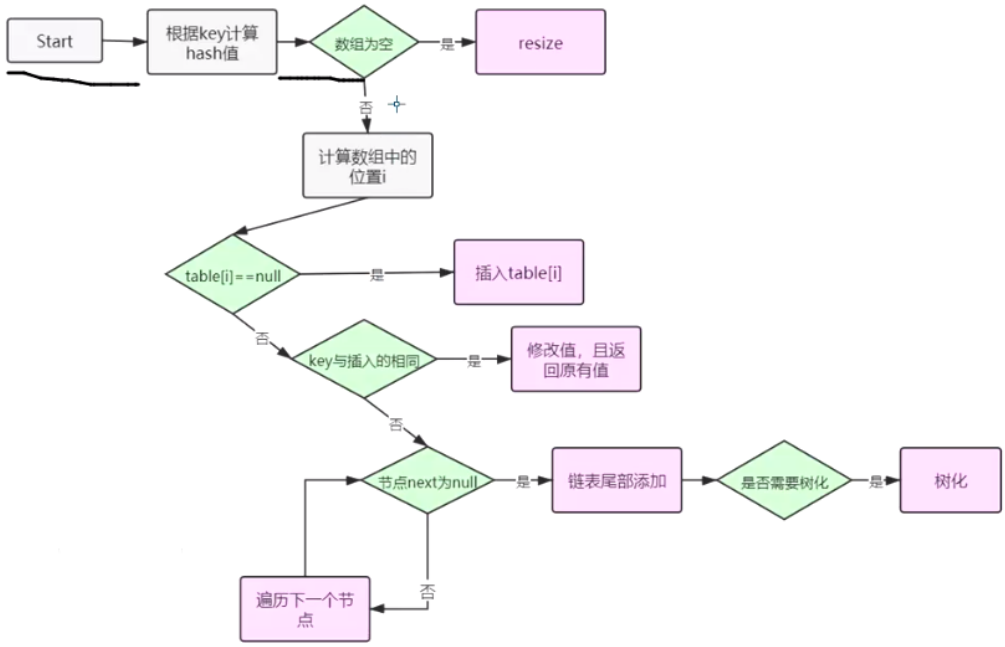
- 默认容量 16，默认负载因子 0.75
  - **HashMap容量为什么是2的n次方**
- 扩容是 2 倍旧的容量
- 在存储数据时， `key` 的 `hash` 计算调用的是 `HashMap` 中的 `hash` 方法
- 添加值时，如果 `hash` 一样添加到链表尾部

> `HashMap` 类大致相当于 `Hashtable` ，除了它是不同步的，并允许null

内部采用 `数组` + `链表实现` ，`JDK 8` 及以后版本增加红黑树的支持。

`HashMap` 的 `put` 过程:

- if与else
- 过程
- 总结



存储时，根据内部的 `hash` 方法计算 `key` ,来确定 `value` 的存储位置（ `Map` 的桶bucket），如果 `hash` 相同，在计算下标。如果产生没有碰撞（ `key` 不相同）,直接放到桶中，由于 `hash` 相同，所以放到一个桶中。放的时候，如果没有超过阈值(8)，以链表的形式放到后边，长度超过阈值且数组长度大于等于64将链表转换成红黑树存储。

- 删除元素时，如果时以红黑树存储的如果节点小于 6 个将会变为链表存储

如果产生碰撞（节点已经存在）就替换值

```java
public V put(K key, V value) {
    // 根据 key 计算 hash 值后调用 putVal 方法
    return putVal(hash(key), key, value, false, true);
}

// HashMap 中计算 hash 的方法
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}


final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // 判断是否是第一个元素
    if ((tab = table) == null || (n = tab.length) == 0)
        // 初始化 数组
        n = (tab = resize()).length;
    // (n - 1) & hash 找key在数组中的位置。 判断这个位置是否有元素
    if ((p = tab[i = (n - 1) & hash]) == null)
        // 这个位置没元素，直接放到这个位置上
        tab[i] = newNode(hash, key, value, null);
    else {
        // i 这个位置上有元素
        Node<K,V> e; K k;
        // 判断当前hash和 i这个位置上的元素的hash是否一样
        if (p.hash == hash &&
            // key 是否相同
            ((k = p.key) == key || (key != null && key.equals(k))))
            // 已经存在同样的 key， 将原来 key 对象赋值给 e
            e = p;
        else if (p instanceof TreeNode) // 是否是红黑树
            // 添加红黑树节点
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            // 链表结构
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) { // 最后一个节点
                    // 当前对象挂载链表最后
                    p.next = newNode(hash, key, value, null);
                    // 当链表节点数 大于等于 8 时
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        // 将链表转换为 红黑树 来存储
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
```

```
            }
        }
        if (e != null) { // existing mapping for key 已经存在这个 key
            // 替换
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

使用，主要是操作方法，demo如下：

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

        HashMap map = new HashMap();
        map.put(new Person("张三", 23), 1);
        map.put(new Person("李四", 22), 2);
        map.put(new Person("张三", 23), 3);

        Person p = new Person("王五",22);
        map.put(p, 4);
        if (map.containsKey(p)){
            System.out.println("已经存在王五了");
        }else {
            map.put(p, 4);
        }

        if(map.containsValue(4)) {
            System.out.println("已经有value是4的了");
        }else {
            System.out.println("还没有value是4的");
        }

        System.out.println("map.size() --> " + map.size()); //2
        System.out.println(map.toString());

        System.out.println("======================");

```

```
27          Object obj = map.get(new Person("李四", 22));
28          System.out.println(obj);//2
29
30          System.out.println("map.isEmpty() --> " + map.isEmpty());
31
32
33          HashMap map1 = new HashMap();
34          map1.put("a", "a1");
35          System.out.println("map1.size() --> " + map1.size());// 1
36          map1.putAll(map);
37          System.out.println("map1.size() --> " + map1.size()); //4
38          System.out.println(map1);
39
40          Object obj1 = map1.remove("a");
41          System.out.println(obj1); //a1
42
43          boolean flag = map1.remove(new Person("张三", 23), 3);
44          System.out.println(flag);
45
46
47          System.out.println("============= replace start ===============");
48
49          Object obj2 = map1.replace(new Person("李四", 22), 3);//原来的value
50
51          boolean flag1 = map1.replace(new Person("李四", 22), 3, 2);
52          System.out.println(flag1);
53          System.out.println(map1);
54
55          //迭代
56          System.out.println("======== 1. keySet ============");
57          /**
58           * 1. keySet
59           *   首先获取map.keySet(), 拿到key的集合
60           *   然后迭代key集合, 用get方法获取value
61           */
62          //for-each-loop
63          for (Object key : map.keySet()) {
64              System.out.println(map.get(key));
65          }
66          //Iterator
67          Iterator iter = map.keySet().iterator();
68          while(iter.hasNext()) {
69              Object key = iter.next();
70              System.out.println(map.get(key));
71          }
72
73          System.out.println("======= 2.values ==========");
74          /**
75           * 2.values
76           *  values就是value的集合, map.values()可以拿到
77           *  然后迭代values就行了
```

```
78          */
79         for (Object o : map.values()) {
80             System.out.println(o);
81         }
82 
83         Iterator iterator = map.values().iterator();
84         while(iterator.hasNext()) {
85             System.out.println(iterator.next());
86         }
87 
88         System.out.println("======= 3.entrySet ==========");
89 
90         /**
91          * 3.entrySet
92          *  map中每一个key对应一个value，这样映射就是一个entry
93          *  然后根据entry获取key和value，entry.getKey()/entry.getValue()
94          */
95         Set set = map.entrySet();
96 
97         for (Object object : set) {
98             //object --> entry
99             Entry entry = (Entry) object;
100            Object key = entry.getKey();
101            Object value = entry.getValue();
102            System.out.println(key + "--->" + value);
103        }
104 
105        Iterator  i = set.iterator();
106        while (i.hasNext()) {
107            Entry entry = (Entry) i.next();
108            System.out.println(entry.getValue());
109 
110        }
111    }
```

LinkedHashMap

```
1 public class LinkedHashMap<K,V>
2     extends HashMap<K,V>
3     implements Map<K,V>
```

- 哈希表和双向链表实现的 `Map` 接口
- 具有可预测的迭代次序(有序)
- 非线程安全
- 允许空元素

构造方法，只说有区别的

```java
/**
 initialCapacity - 初始容量
  loadFactor - 负载因子
  accessOrder - 排序模式 - true的访问顺序,  false的插入顺序 默认是false
 */
public LinkedHashMap(int initialCapacity,
                     float loadFactor,
                     boolean accessOrder)
    //accessOrder表示的是存取顺序，true表示访问模式，false插入模式
    //插入模式按照插入的顺序排放，始终不变
    //访问模式是访问某个节点后，会将此节点放到链表的后边去
```

常用方法没特别需要讲解的，自行了解吧。

ConcurrentHashMap

`java.util.concurrent.ConcurrentHashMap`

```java
public class ConcurrentHashMap<K,V> extends AbstractMap<K,V>
    implements ConcurrentMap<K,V>, Serializable
```

是 `Java` 从 `JDK1.5` 开始提供的一个 `HashMap` 线程安全的实现类。且 `java.util.concurrent` 包下所有的类都是线程安全的

Properties

```java
public
class Properties extends Hashtable<Object,Object>
```

`Properties` 类表示一组持久的属性。 `Properties` 可以保存到流中或从流中加载。 属性列表中的每个键及其对应的值都是一个字符串。

内部使用 `ConcurrentHashMap` 存储

操作和 `Hashtable` 基本一致，主要用于从流中记载或保存到流中去。(后期 `JDBC` 模块会使用到)