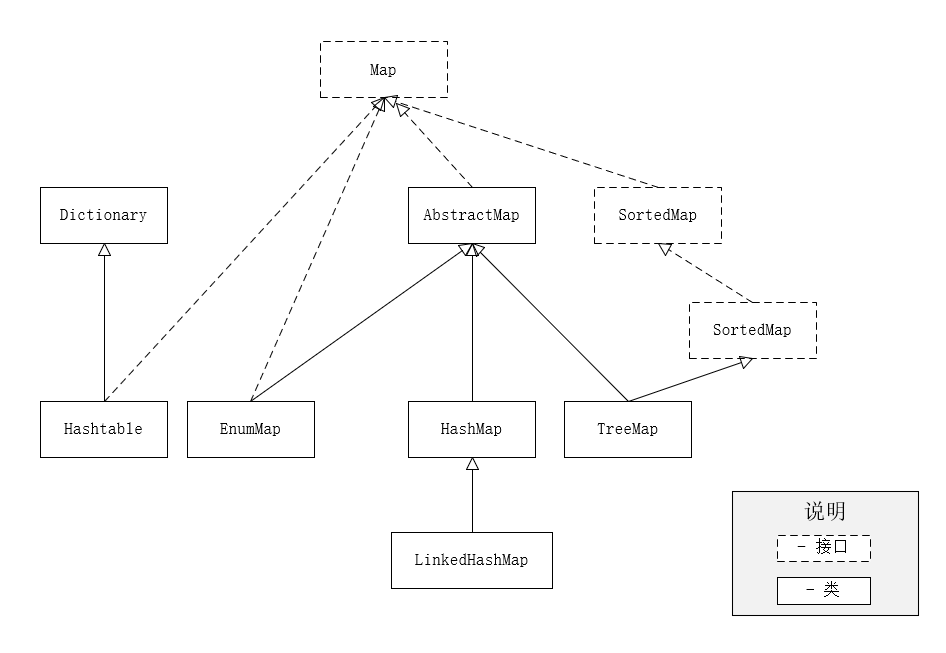
## 集合详解之 Map + 面试题

集合有两个大接口：Collection 和 Map，本文重点来讲解集合中另一个常用的集合类型 Map。

以下是 Map 的继承关系图：



11

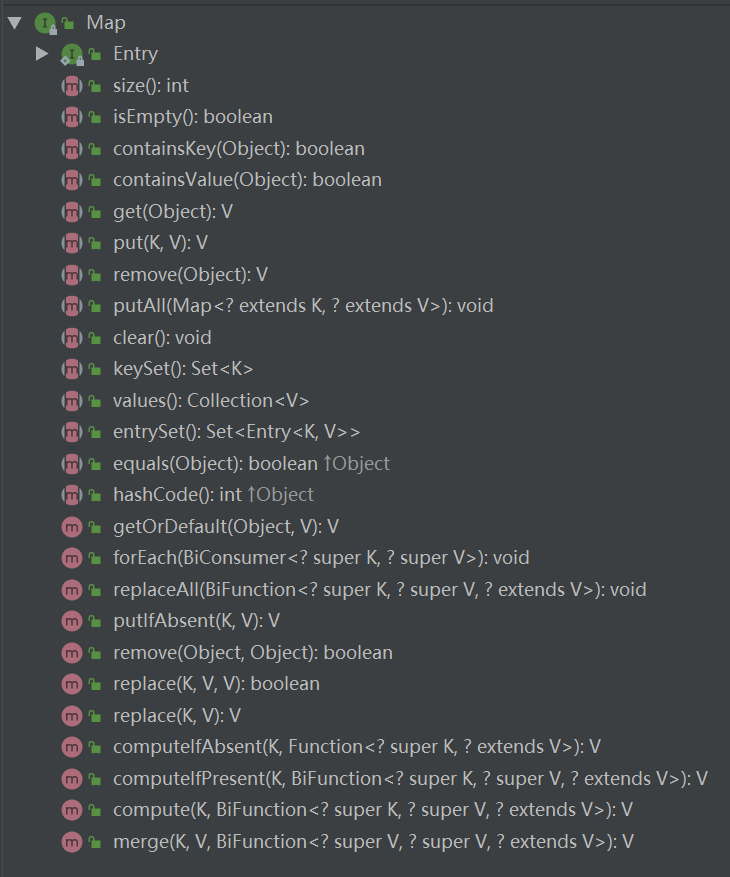
### Map 简介

Map 常用的实现类如下：

* **Hashtable** ：Java 早期提供的一个哈希表实现，它是线程安全的，不支持 null 键和值，因为它的性能不如 ConcurrentHashMap，所以很少被推荐使用。
* **HashMap** ：最常用的哈希表实现，如果程序中没有多线程的需求，HashMap 是一个很好的选择，支持 null 键和值，如果在多线程中可用 ConcurrentHashMap 替代。
* **TreeMap** ：基于红黑树的一种提供顺序访问的 Map，自身实现了 key 的自然排序，也可以指定 Comparator 来自定义排序。
* **LinkedHashMap** ：HashMap 的一个子类，保存了记录的插入顺序，可在遍历时保持与插入一样的顺序。

### Map 常用方法

常用方法包括：put、remove、get、size 等，所有方法如下图：



enter image description here

使用示例，请参考以下代码：

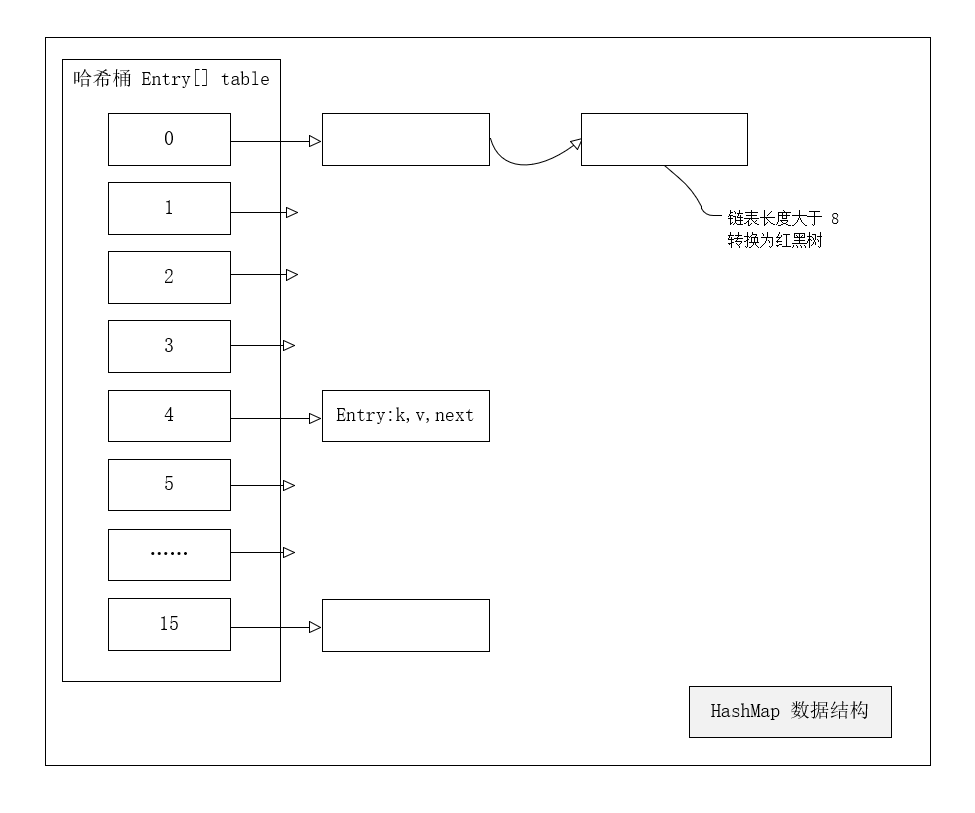
Map hashMap = new HashMap();  
// 增加元素  
hashMap.put("name", "老王");  
hashMap.put("age", "30");  
hashMap.put("sex", "你猜");  
// 删除元素  
hashMap.remove("age");  
// 查找单个元素  
System.out.println(hashMap.get("age"));  
// 循环所有的 key  
for (Object k : hashMap.keySet()) {  
 System.out.println(k);  
}  
// 循环所有的值  
for (Object v : hashMap.values()) {  
 System.out.println(v);  
}

以上为 HashMap 的使用示例，其他类的使用也是类似。

### HashMap 数据结构

HashMap 底层的数据是数组被成为哈希桶，每个桶存放的是链表，链表中的每个节点，就是 HashMap 中的每个元素。在 JDK 8 当链表长度大于等于 8 时，就会转成红黑树的数据结构，以提升查询和插入的效率。

HashMap 数据结构，如下图：



enter image description here

### HashMap 重要方法

#### 1）添加方法：put(Object key, Object value)

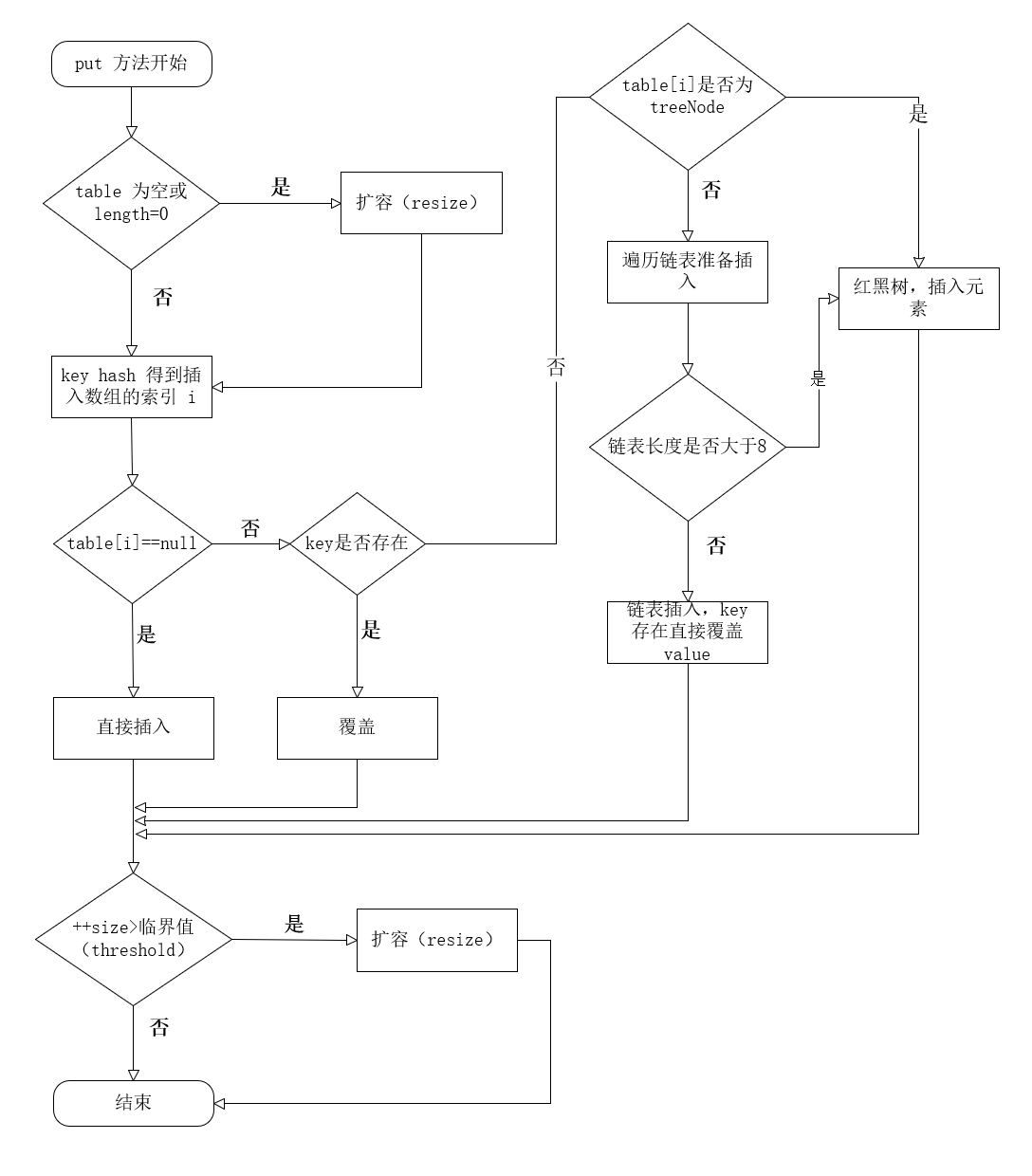
执行流程如下：

* 对 key 进行 hash 操作，计算存储 index；
* 判断是否有哈希碰撞，如果没碰撞直接放到哈希桶里，如果有碰撞则以链表的形式存储；
* 判断已有元素的类型，决定是追加树还是追加链表，当链表大于等于 8 时，把链表转换成红黑树；
* 如果节点已经存在就替换旧值；
* 判断是否超过阀值，如果超过就要扩容。

源码及说明：

public V put(K key, V value) {  
 // 对 key 进行 hash()  
 return putVal(hash(key), key, value, false, true);  
}  
static final int hash(Object key) {  
 int h;  
 // 对 key 进行 hash() 的具体实现  
 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;  
 // tab为空则创建  
 if ((tab = table) == null || (n = tab.length) == 0)  
 n = (tab = resize()).length;  
 // 计算 index，并对 null 做处理  
 if ((p = tab[i = (n - 1) & hash]) == null)  
 tab[i] = newNode(hash, key, value, null);  
 else {  
 Node<K,V> e; K k;  
 // 节点存在  
 if (p.hash == hash &&  
 ((k = p.key) == key || (key != null && key.equals(k))))  
 e = p;  
 // 该链为树  
 else if (p instanceof TreeNode)  
 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  
 // 该链为链表  
 else {  
 for (int binCount = 0; ; ++binCount) {  
 if ((e = p.next) == null) {  
 p.next = newNode(hash, key, value, null);  
 if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st  
 treeifyBin(tab, hash);  
 break;  
 }  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k))))  
 break;  
 p = e;  
 }  
 }  
 // 写入  
 if (e != null) { // existing mapping for key  
 V oldValue = e.value;  
 if (!onlyIfAbsent || oldValue == null)  
 e.value = value;  
 afterNodeAccess(e);  
 return oldValue;  
 }  
 }  
 ++modCount;  
 // 超过load factor\*current capacity，resize  
 if (++size > threshold)  
 resize();  
 afterNodeInsertion(evict);  
 return null;  
}

put() 执行流程图如下：



enter image description here

#### 2）获取方法：get(Object key)

执行流程如下：

* 首先比对首节点，如果首节点的 hash 值和 key 的 hash 值相同，并且首节点的键对象和 key 相同（地址相同或 equals 相等），则返回该节点；
* 如果首节点比对不相同、那么看看是否存在下一个节点，如果存在的话，可以继续比对，如果不存在就意味着 key 没有匹配的键值对。

源码及说明：

public V get(Object key) {  
 Node<K,V> e;  
 return (e = getNode(hash(key), key)) == null ? null : e.value;  
}  
/\*\*  
\* 该方法是 Map.get 方法的具体实现  
\* 接收两个参数  
\* @param hash key 的 hash 值，根据 hash 值在节点数组中寻址，该 hash 值是通过 hash(key) 得到的  
\* @param key key 对象，当存在 hash 碰撞时，要逐个比对是否相等  
\* @return 查找到则返回键值对节点对象，否则返回 null  
\*/  
final Node<K,V> getNode(int hash, Object key) {  
 Node<K,V>[] tab; Node<K,V> first, e; int n; K k; // 声明节点数组对象、链表的第一个节点对象、循环遍历时的当前节点对象、数组长度、节点的键对象  
 // 节点数组赋值、数组长度赋值、通过位运算得到求模结果确定链表的首节点  
 if ((tab = table) != null && (n = tab.length) > 0 &&  
 (first = tab[(n - 1) & hash]) != null) {  
 if (first.hash == hash && // 首先比对首节点，如果首节点的 hash 值和 key 的 hash 值相同，并且首节点的键对象和 key 相同（地址相同或 equals 相等），则返回该节点  
 ((k = first.key) == key || (key != null && key.equals(k))))  
 return first; // 返回首节点  
  
 // 如果首节点比对不相同、那么看看是否存在下一个节点，如果存在的话，可以继续比对，如果不存在就意味着 key 没有匹配的键值对   
 if ((e = first.next) != null) {  
 // 如果存在下一个节点 e，那么先看看这个首节点是否是个树节点  
 if (first instanceof TreeNode)  
 // 如果是首节点是树节点，那么遍历树来查找  
 return ((TreeNode<K,V>)first).getTreeNode(hash, key);   
  
 // 如果首节点不是树节点，就说明还是个普通的链表，那么逐个遍历比对即可   
 do {  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k)))) // 比对时还是先看 hash 值是否相同、再看地址或 equals  
 return e; // 如果当前节点e的键对象和key相同，那么返回 e  
 } while ((e = e.next) != null); // 看看是否还有下一个节点，如果有，继续下一轮比对，否则跳出循环  
 }  
 }  
 return null; // 在比对完了应该比对的树节点 或者全部的链表节点 都没能匹配到 key，那么就返回 null

### 相关面试题

#### 1.Map 常见实现类有哪些？

答：Map 的常见实现类如下列表：

* Hashtable：Java 早期提供的一个哈希表实现，它是线程安全的，不支持 null 键和值，因为它的性能不如 ConcurrentHashMap，所以很少被推荐使用；
* HashMap：最常用的哈希表实现，如果程序中没有多线程的需求，HashMap 是一个很好的选择，支持 null 键和值，如果在多线程中可用 ConcurrentHashMap 替代；
* TreeMap：基于红黑树的一种提供顺序访问的 Map，自身实现了 key 的自然排序，也可以指定的 Comparator 来自定义排序；
* LinkedHashMap：HashMap 的一个子类，保存了记录的插入顺序，可在遍历时保持与插入一样的顺序。

#### 2.使用 HashMap 可能会遇到什么问题？如何避免？

答：HashMap 在并发场景中可能出现死循环的问题，这是因为 HashMap 在扩容的时候会对链表进行一次倒序处理，假设两个线程同时执行扩容操作，第一个线程正在执行 B→A 的时候，第二个线程又执行了 A→B ，这个时候就会出现 B→A→B 的问题，造成死循环。  
解决的方法：升级 JDK 版本，在 JDK 8 之后扩容不会再进行倒序，因此死循环的问题得到了极大的改善，但这不是终极的方案，因为 HashMap 本来就不是用在多线程版本下的，如果是多线程可使用 ConcurrentHashMap 替代 HashMap。

#### 3.以下说法正确的是？

A：Hashtable 和 HashMap 都是非线程安全的  
B：ConcurrentHashMap 允许 null 作为 key  
C：HashMap 允许 null 作为 key  
D：Hashtable 允许 null 作为 key  
答：C  
题目解析：Hashtable 是线程安全的，ConcurrentHashMap 和 Hashtable 是不允许 null 作为键和值的。

#### 4.TreeMap 怎么实现根据 value 值倒序？

答：使用 Collections.sort(list, new Comparator<Map.Entry<String, String>>() 自定义比较器实现，先把 TreeMap 转换为 ArrayList，在使用 Collections.sort() 根据 value 进行倒序，完整的实现代码如下。

TreeMap<String, String> treeMap = new TreeMap();  
treeMap.put("dog", "dog");  
treeMap.put("camel", "camel");  
treeMap.put("cat", "cat");  
treeMap.put("ant", "ant");  
// map.entrySet() 转成 List  
List<Map.Entry<String, String>> list = new ArrayList<>(treeMap.entrySet());  
// 通过比较器实现比较排序  
Collections.sort(list, new Comparator<Map.Entry<String, String>>() {  
 public int compare(Map.Entry<String, String> m1, Map.Entry<String, String> m2) {  
 return m2.getValue().compareTo(m1.getValue());  
 }  
});  
// 打印结果  
for (Map.Entry<String, String> item : list) {  
 System.out.println(item.getKey() + ":" + item.getValue());  
}

程序执行结果：

dog:dog  
cat:cat  
camel:camel  
ant:ant

#### 5.以下哪个 Set 实现了自动排序？

A：LinedHashSet  
B：HashSet  
C：TreeSet  
D：AbstractSet

答：C

#### 6.以下程序运行的结果是什么？

Hashtable hashtable = new Hashtable();  
hashtable.put("table", null);  
System.out.println(hashtable.get("table"));

答：程序执行报错：java.lang.NullPointerException。Hashtable 不允许 null 键和值。

#### 7.HashMap 有哪些重要的参数？用途分别是什么？

答：HashMap 有两个重要的参数：容量（Capacity）和负载因子（LoadFactor）。

* 容量（Capacity）：是指 HashMap 中桶的数量，默认的初始值为 16。
* 负载因子（LoadFactor）：也被称为装载因子，LoadFactor 是用来判定 HashMap 是否扩容的依据，默认值为 0.75f，装载因子的计算公式 = HashMap 存放的 KV 总和（size）/ Capacity。

#### 8.HashMap 和 Hashtable 有什么区别？

答：HashMap 和 Hashtable 区别如下：

* Hashtable 使用了 synchronized 关键字来保障线程安全，而 HashMap 是非线程安全的；
* HashMap 允许 K/V 都为 null，而 Hashtable K/V 都不允许 null；
* HashMap 继承自 AbstractMap 类；而 Hashtable 继承自 Dictionary 类。

#### 9.什么是哈希冲突？

答：当输入两个不同值，根据同一散列函数计算出相同的散列值的现象，我们就把它叫做碰撞（哈希碰撞）。

#### 10.有哪些方法可以解决哈希冲突？

答：哈希冲突的常用解决方案有以下 4 种。

* 开放定址法：当关键字的哈希地址 p=H（key）出现冲突时，以 p 为基础，产生另一个哈希地址 p1，如果 p1 仍然冲突，再以 p 为基础，产生另一个哈希地址 p2，循环此过程直到找出一个不冲突的哈希地址，将相应元素存入其中。
* 再哈希法：这种方法是同时构造多个不同的哈希函数，当哈希地址 Hi=RH1（key）发生冲突时，再计算 Hi=RH2（key），循环此过程直到找到一个不冲突的哈希地址，这种方法唯一的缺点就是增加了计算时间。
* 链地址法：这种方法的基本思想是将所有哈希地址为 i 的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第 i 个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。
* 建立公共溢出区：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

#### 11.HashMap 使用哪种方法来解决哈希冲突（哈希碰撞）？

答：HashMap 使用链表和红黑树来解决哈希冲突，详见本文 put() 方法的执行过程。

#### 12.HashMap 的扩容为什么是 2^n ？

答：这样做的目的是为了让散列更加均匀，从而减少哈希碰撞，以提供代码的执行效率。

#### 13.有哈希冲突的情况下 HashMap 如何取值？

答：如果有哈希冲突，HashMap 会循环链表中的每项 key 进行 equals 对比，返回对应的元素。相关源码如下：

do {  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k)))) // 比对时还是先看 hash 值是否相同、再看地址或 equals  
 return e; // 如果当前节点 e 的键对象和 key 相同，那么返回 e  
} while ((e = e.next) != null); // 看看是否还有下一个节点，如果有，继续下一轮比对，否则跳出循环

#### 14.以下程序会输出什么结果？

class Person {  
 private Integer age;  
 public boolean equals(Object o) {  
 if (o == null || !(o instanceof Person)) {  
 return false;  
 } else {  
 return this.getAge().equals(((Person) o).getAge());  
 }  
 }  
 public int hashCode() {  
 return age.hashCode();  
 }  
 public Person(int age) {  
 this.age = age;  
 }  
 public void setAge(int age) {  
 this.age = age;  
 }  
 public Integer getAge() {  
 return age;  
 }  
 public static void main(String[] args) {  
 HashMap<Person, Integer> hashMap = new HashMap<>();  
 Person person = new Person(18);  
 hashMap.put(person, 1);  
 System.out.println(hashMap.get(new Person(18)));  
 }  
}

答：1  
题目解析：因为 Person 重写了 equals 和 hashCode 方法，所有 person 对象和 new Person(18) 的键值相同，所以结果就是 1。

#### 15.为什么重写 equals() 时一定要重写 hashCode()？

答：因为 Java 规定，如果两个对象 equals 比较相等（结果为 true），那么调用 hashCode 也必须相等。如果重写了 equals() 但没有重写 hashCode()，就会与规定相违背，比如以下代码（故意注释掉 hashCode 方法）：

class Person {  
 private Integer age;  
 public boolean equals(Object o) {  
 if (o == null || !(o instanceof Person)) {  
 return false;  
 } else {  
 return this.getAge().equals(((Person) o).getAge());  
 }  
 }  
// public int hashCode() {  
// return age.hashCode();  
// }  
 public Person(int age) {  
 this.age = age;  
 }  
 public void setAge(int age) {  
 this.age = age;  
 }  
 public Integer getAge() {  
 return age;  
 }  
 public static void main(String[] args) {  
 Person p1 = new Person(18);  
 Person p2 = new Person(18);  
 System.out.println(p1.equals(p2));  
 System.out.println(p1.hashCode() + " : " + p2.hashCode());  
 }  
}

执行的结果：

true  
21685669 : 2133927002

如果重写 hashCode() 之后，执行的结果是：

true  
18 : 18

这样就符合了 Java 的规定，因此重写 equals() 时一定要重写 hashCode()。

#### 16.HashMap 在 JDK 7 多线程中使用会导致什么问题？

答：HashMap 在 JDK 7 中会导致死循环的问题。因为在 JDK 7 中，多线程进行 HashMap 扩容时会导致链表的循环引用，这个时候使用 get() 获取元素时就会导致死循环，造成 CPU 100% 的情况。

#### 17.HashMap 在 JDK 7 和 JDK 8 中有哪些不同？

答：HashMap 在 JDK 7 和 JDK 8 的主要区别如下。

* 存储结构：JDK 7 使用的是数组 + 链表；JDK 8 使用的是数组 + 链表 + 红黑树。
* 存放数据的规则：JDK 7 无冲突时，存放数组；冲突时，存放链表；JDK 8 在没有冲突的情况下直接存放数组，有冲突时，当链表长度小于 8 时，存放在单链表结构中，当链表长度大于 8 时，树化并存放至红黑树的数据结构中。
* 插入数据方式：JDK 7 使用的是头插法（先将原位置的数据移到后 1 位，再插入数据到该位置）；JDK 8 使用的是尾插法（直接插入到链表尾部/红黑树）。

### 总结

通过本文可以了解到：

* Map 的常用实现类 Hashtable 是 Java 早期的线程安全的哈希表实现；
* HashMap 是最常用的哈希表实现，但它是非线程安全的，可使用 ConcurrentHashMap 替代；
* TreeMap 是基于红黑树的一种提供顺序访问的哈希表实现；
* LinkedHashMap 是 HashMap 的一个子类，保存了记录的插入顺序，可在遍历时保持与插入一样的顺序。

HashMap 在 JDK 7 可能在扩容时会导致链表的循环引用而造成 CPU 100%，HashMap 在 JDK 8 时数据结构变更为：数组 + 链表 + 红黑树的存储方式，在没有冲突的情况下直接存放数组，有冲突，当链表长度小于 8 时，存放在单链表结构中，当链表长度大于 8 时，树化并存放至红黑树的数据结构中。

[点击此处下载本文源码](https://github.com/vipstone/java-%20interview/tree/master/interview-code/src/main/java/com/interview)

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持