

转载请注明链接:[https://blog.csdn.net/feather\\_wch/article/details/81351151](https://blog.csdn.net/feather_wch/article/details/81351151)

详细介绍HashMap的原理、哈希算法、哈希冲突、hashCode、JDK1.8前后的区别、Fail-Fast(快速失败机制)、LinkedHashMap的原理、与HashSet的异同、与Hashtable的异同。

如果有帮助，请点个赞万分感谢！

# HashMap(52题)

版本：2018/8/2-1(17:00)

- [HashMap\(52题\)](#)
  - [HashMap原理分析](#)
    - [构造函数](#)
    - [数据结构](#)
      - [哈希算法](#)
      - [Entry](#)
    - [快速存取](#)
      - [哈希策略](#)
    - [扩容](#)
    - [数据读取](#)
    - [底层数组的长度](#)
    - [Fail-Fast机制](#)
    - [JDK1.8前后区别](#)
  - [LinkedHashMap](#)
    - [LruCache和DiskLruCache](#)
  - [HashTable](#)
  - [HashSet](#)
  - [细节面试题补充](#)
  - [参考资料](#)

## 1、HashMap是什么？

1. Map族中最常用的集合
2. 是Java Collection Framework的重要成员
3. Map是键值对映射的抽象接口，该映射只允许一个键对应一个值
4. HashMap存储的对象是Entry(同时包含Key-Value)
5. HashMap中会根据hash算法来计算key-value的存储位置并进行快速存取
6. HashMap只允许一条Entry的Key为Null(多条会覆盖)，但允许多条Entry的Value为Null
7. HashMap是Map的非同步实现。

## 2、集合存储的是Java对象？

不是：

1. 没有将Java对象放入到容器中
2. 容器仅仅保存这些对象的引用，这些引用指向了实际内存地址中的Java对象。

## HashMap原理分析

### 3、HashMap的实现

1. 继承 `AbstractMap<K, V>`类：提供了Map接口的骨干实现，以最大程度减少实现Map接口所需要的工作。
2. 实现 `Map<K, V>`接口

3. 实现 Cloneable接口：支持浅复制
4. 实现 Serializable接口：支持序列化

## 构造函数

### 4、HashMap具有的构造方法一共有4种？

1. HashMap(): 初始容量-16；负载因子-0.75
2. HashMap(int initialCapacity, float loadFactor): 指定初始容量和负载因子
3. HashMap(int initialCapacity): 指定初始容量；负载因子-0.75
4. HashMap(Map m): 构造与指定Map具有相同映射的HashMap；初始容量 $\geq 16$ ；负载因子-0.75

### 5、HashMap底层数据存储的实现方法是什么？

1. 采用 数组 实现，名为table
2. 数组的每一项都是 一条链表

### 6、初始容量是什么？

1. 哈希表 中桶的数量，也就是table数组的大小

### 7、负载因子是什么？

1. 哈希表在其容量自动扩容前可以达到当前总容量的比例(如：达到 当前容量 \* 负载因子 时会进行自动扩容)
2. 用于衡量的是一个散列表的空间的使用程度
3. 负载因子越大，散列表的装填程度越高

### 8、负载因子在拉链法的哈希表中的意义？

1. 使用 拉链法 的哈希表来说，查找一个元素的平均时间是  $O(1+a)$ ，a 指的是链的长度，是一个常数。
2. 负载因子越大，空间利用越充分，查找效率会越低
3. 负载因子越小，数据越稀疏，空间浪费越严重。
4. 0.75是系统默认在时间和空间上的折中办法，一般不需要修改。

## 数据结构

### 9、什么是Hash(哈希)？

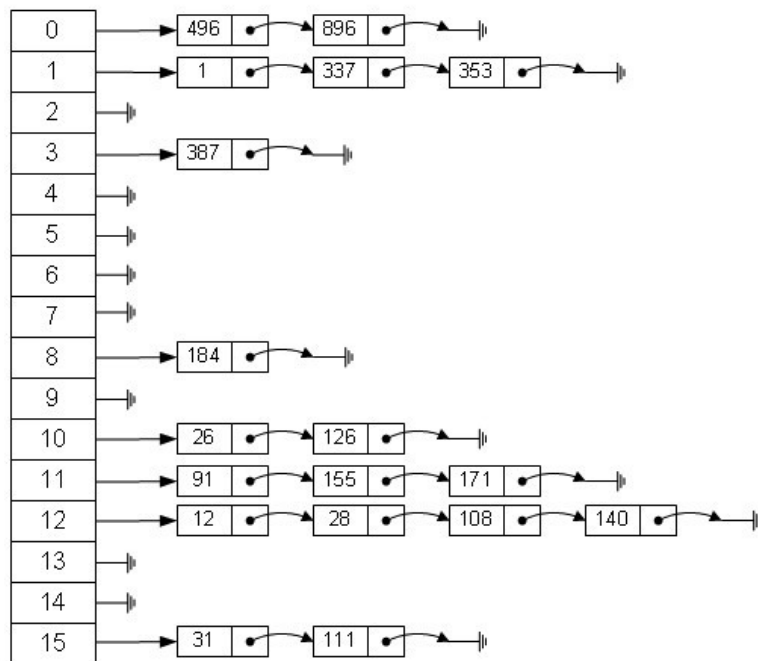
1. Hash就是把任意长度的输入，通过哈希算法，变换成固定长度的输出(一般是整型值)，该输出就是哈希值。
2. 这种转换是一种 压缩映射，散列值的控件通常远小于输入的空间。
3. 不同的输入可能会散列成相同的输出，从而不可能从散列值来唯一的确定输入值。
4. 简而言之，就是一种将任意长度的信息压缩到某一个固定长度的消息摘要函数。

### 10、哈希表的由来？

1. 数组的特点：寻址容易，插入和删除困难
2. 链表的特点：寻址困难，插入和删除容易
3. 哈希表的特点：寻址容易，插入和删除也容易。适合快速查找、插入、删除
4. 哈希表有多种实现方法：最经典的是拉链法。

### 11、拉链法是什么？

1. 可以理解为 链表的数组
2. 会根据元素的特征将其分配到不同的链表中
3. 通过 元素的特征 去计算 元素数组下标 的方法就是 哈希算法



## 12、使用哈希表的两个关键点

1. 哈希算法(Hash函数)的选择：应该针对不同的对象(字符串、整数等)采用具体不同的哈希方法。
2. hash冲突的处理：一种是开放散列（open hashing）/ 拉链法（针对桶链结构）；另一种是封闭散列（closed hashing）/ 开放定址法。

## 哈希算法

### 13、==的作用

1. 基本数据类型，比较值是否相等
2. 引用类型，比较内存地址是否相等。

### 14、equals方法的作用

1. 本意：比较两个对象的内容是否相等
2. Object中equals仅仅是比较两者的引用是否相等。

### 15、String的equals为什么能正确比较？内部的实现方法？

重写了 equals 方法：

1. 先 比较引用是否相同(是否为同一对象\同一个内存地址)
2. 再 判断类型是否一致（是否为同一类型(String))
3. 最后 比较内容是否一致

### 16、重写equals必须遵守的规则？(5)

对称性、自反性、类推性、一致性、对称性

### 17、hashCode是什么？

1. Object类的一个 native 方法
2. 会针对不同的对象返回不同的整数(通过将该对象的内部地址转换成一个整数来实现)
3. hashCode只是在需要用到哈希算法的数据结构中才有用，比如 HashSet, HashMap 和 Hashtable。
4. 本质是系统用来快速检索对象而使用。

### 18、Java的集合(Collections)有哪几类？(3)

1. List: 元素有序；元素可以重复
2. Queue: 元素有序；元素可以重复

3. Set: 元素无序; 元素不可以重复

- Map不属于 Collections

## 19、如何保证元素不重复-equals方法?

1. 元素重复问题可以通过 `Object.equals` 判断
2. `equals`的缺点 在于如果有1000个元素, 第1001个元素加入集合时, 为了保证不重复, 需要调用1000次`equals`方法。性能低下

## 20、hashCode方法如何保证元素不重复?

1. 通过 `hashCode` 计算出元素对应的值, 根据该值计算出元素在数组中的位置。
2. 如果该位置上没有数据, 将该元素存储到该位置。
3. 如果该位置上有数据, 调用`equals`方法将两者比较。
4. 元素相同, 不进行存储。
5. 元素不同, 存储到该位置对应的链表中(`HashSet`, `HashMap` 和 `Hashtable`的实现总将元素放到链表的表头)

## Entry

### 21、Entry是什么?

1. Entry是HashMap的内部类
2. 实现了 `Map.Entry` 接口
3. 包含了 `key`、`value`、下个节点`next`、`hash`值 四个属性
4. 是构成哈希表的基石, 是哈希表所存储元素的具体形式。

## 快速存取

### 22、HashMap如何确保Key的唯一性?

1. `put()` 存值的时候, 会调用 `Key`的`hashCode` 获取到`hash`值, 根据该值找到对应的桶。
2. 如果桶中没有数据, 直接存储该`Key-Value`
3. 如果桶中有数据, 依次调用 `equals` 进行比较(判断`Key`是否存在)
4. `Key`不存在, 直接保存`Key-Value`到链表中。
5. `Key`存在, 使用新`Value`替换旧`Value`, 并且返回旧`Value`值

`equals`只有在哈希碰撞时才会被用到。

### 23、HashMap的put源码

```
public V put(K key, V value) {
    // 1、当key为null时, 调用putForNullKey方法, 并将该键值对保存到table的第一个位置
    if (key == null) return putForNullKey(value);
    // 2、根据key的hashCode计算hash值
    int hash = hash(key.hashCode());
    // 3、 计算该键值对在数组中的存储位置 (哪个桶)
    int i = indexFor(hash, table.length);
    // 4、在table的第i个桶上进行迭代, 寻找 key 保存的位置
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        // 5、判断该条链上是否存在hash值相同且key值相等的映射, 若存在, 则直接覆盖 value, 并返回旧value
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;    // 返回旧值
        }
    }
    // 6、修改次数增加1, 快速失败机制
    modCount++;
    // 7、原HashMap中无该映射, 将该添加至该链的链头
    addEntry(hash, key, value, i);
    return null;
}
```

1. 如果key = null，直接保存到table数组的第一个位置中

## 24、HashMap的putForNullKey

```
private V putForNullKey(V value) {
    // 1、若key==null，则将其放入table的第一个桶，即 table[0]
    for (Entry<K,V> e = table[0]; e != null; e = e.next) {
        // 2、若已经存在key为null的键，则替换其值，并返回旧值
        if (e.key == null) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    // 3、快速失败
    modCount++;
    // 4、否则，将其添加到 table[0] 的桶中
    addEntry(0, null, value, 0);
    return null;
}
```

1. key = null 时一定保存在第一个桶中。
2. 会从第一个桶中去查找key == null的Entry，存在就新Value替换旧Value，不存在就新增Entry到桶中。

## 哈希策略

### 25、HashMap的哈希策略

```
// 1、根据key的hashCode计算hash值
int hash = hash(key.hashCode());
// 2、计算该键值对在数组中的存储位置（哪个桶）
int i = indexFor(hash, table.length);
```

1. 没有直接使用 hashCode，而是通过 hash(xxx) 对哈希值进行打散。
2. indexFor通过 &运算 获取到位于[0, length - 1)区间的数值，该值越均匀，空间利用率越高。

### 26、HashMap的hash方法(为什么能防止产生质量低下的哈希值)?

```
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

1. JDK: hash()对hashCode进行重新计算，是为了防止产生质量低下的Hash值。
2. HashMap 的数组长度都是 2 的幂次方，通过右移可以使低位的数据尽量不相同，从而使hash值分布尽量均匀。

### 27、HashMap的indexFor()作用和技巧?

```
static int indexFor(int h, int length) {
    return h & (length-1); // 作用等价于取模运算，但这种方式效率更高
}
```

1. 为了让元素均匀的分布到table的桶中，常规方法是通过 取模，但是效率低下。
2. 当数组长度为2的n次方时，hash & (length - 1) 就相当于对length取模，而且速度比直接取模要快得多，这是HashMap在速度上的一个优化。

### 28、HashMap性能优化的体现?

1. 对hashCode()返回的哈希值通过hash()进行打散，内部通过右移等位运算，让低位的数据尽量不相同，从而使得hash值分布均匀。
2. 获取到哈希值后，获取数组下标的过程中，不采用取模 的方法，而是进行 &与运算 来达到同效果但性能更高的目的。

### 29、HashMap的键值对添加-addEntry()?

1. HashMap永远都是在链表的表头添加新数据
2. 如果HashMap中元素的个数超过极限值，会进行扩容操作。

## 扩容

### 30、HashMap的扩容

1. 当元素越来越多时，产生碰撞的概率会越来越大，桶中链表的长度也会增加，这样会影响到 HashMap 的存取速度
2. 为了保障效率，在元素数量达到 `threshold(table数组长度 * 负载因子)` 时，会进行扩容操作。
3. 扩容性能损耗很大，如果能提前预知 HashMap 中元素的个数，建议在构造HashMap时指定。能够提高效率

### 31、HashMap的扩容方法resize()源码

```
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    // 1、若 oldCapacity 已达到最大值，直接将 threshold 设为 Integer.MAX_VALUE
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return; // 直接返回
    }
    // 2、否则，创建一个更大的数组
    Entry[] newTable = new Entry[newCapacity];
    // 3、将每条Entry重新哈希到新的数组中
    transfer(newTable);
    // 4、设置新数组，设置新的极限值
    table = newTable;
    threshold = (int)(newCapacity * loadFactor); // 重新设定 threshold
}
```

### 32、重哈希方法transfer()的作用？

1. 重新计算原HashMap中的元素在新table数组中的位置
2. 并进行复制处理

## 数据读取

### 33、HashMap中数据的读取

1. 计算出 key的hash值
2. 查找出对应table数组中的那个桶
3. 遍历桶中的链表，查找并返回 key 对应的 value

### 34、HashMap对Key=null的情况提供了专门方法getForNullKey()

1. 会直接从 `table[0]` 也就是第一个桶中查找
2. 效率更高

### 35、HashMap的get方法返回Null意味着什么？

1. key不存在
2. value = null

### 36、如何判断Key是否存在？

1. 不能通过get的返回值是否为null来判断
2. 需要通过 `containsKey` 进行判断

## 底层数组的长度

### 37、HashMap的底层数组长度为何需要是2的n次方？

1. 减少发生碰撞的概率，使得数据在table数组中更均匀，空间利用率高，查询速度更快(length = 15时，哈希值从0~15，会出现8次hash冲突；当length = 16是，哈希值从0~15，不会出现hash冲突。)
2. hash & (length - 1)效果和取模相用，但是速度、效率更高。

## Fail-Fast机制

38、快速失败机制(Fail-Fast)是什么？

1. HashMap 不是线程安全的
2. 使用迭代器的过程中，如果其他地方修改了Map就会抛出 ConcurrentModificationException 异常
3. 在自身迭代器遍历的循环中，如果往 map 中去添加数据，也会导致抛出异常。

39、快速失败机制的原理

1. HashMap内部有一个 modCount 值，对map的修改会增加该值(+1)
2. 迭代器初始化时会把 modCount 的值赋予 expectedModCount，
3. 迭代过程中如果两个值不相等，表明Map内容已经被修改，就会抛出异常。

40、快速失败实例分析与抛出异常的地方？

```
HashMap<String, String> map = new HashMap();
map.put("0", "1");
map.put("1", "x");
map.put("2", "x");
Iterator<String> i = map.keySet().iterator();
while (i.hasNext()) {
    if (map.get(i.next()).equals("x")) {
        map.put("ok", "true");
    }
}
```

1. 迭代器的hasNext()最终会执行到HashMap的内部迭代器HashIterator的 nextEntry()
2. nextEntry()中会比较modCount和expectedModCount。不相等就会抛出异常。

41、哪些场景不会触发Fail-Fast机制？

1. 改变Map内容后，直接break等操作导致没有执行hasNext()方法就不会触发。
2. 调用 Iterator的remove 不会触发，内部会在删除后进行赋值： expectedModCount = modCount

## JDK1.8前后区别

42、HashMap在JDK1.8前后的区别？


1. Java 8中的HashMap采用Node数组存储数据，可能是链表也可能是红黑树。
2. Java 8中一个桶中key<=8采用链表，如果key>8并且Map容量超过了64采用红黑树。
3. Java 8中使用HashMap对于 Key 对象需要正确的实现 Comparable接口(例如compareTo方法不能返回0) ---两者性能相差近150倍。

43、HashMap在Java 8中必须要Key对象正确实现Comparable的原因


1. 没有正确实现Comparable接口的Key对象，也就是没有定义与其他Key比较的方法，会导致调用 tieBreakOrder() 方法进行比较，效率低下
2. 正确实现Comparable接口的情况下，直接通过 compareTo 去比较，性能大幅度提高。

44、JDK1.8和JDK1.7性能对比

Hash较均匀的时候：性能提升15%~500%

 Hash均匀

Hash极其不均匀：性能提升15%~100%

 Hash极其不均匀

# LinkedHashMap

45、LinkedHashMap是什么？

- 1. HashMap的直接子类
- 2. 继承所有HashMap的特性
- 3. 额外维护一个双向链表，保持了有序性、

## LruCache和DiskLruCache

46、LruCache和DiskLruCache是什么？

- 1. LruCache 是内存缓存。
- 2. DiskLruCache 是磁盘缓存。
- 3. 均基于 Lru算法 和 LinkedHashMap 实现
- 4. 应用：ImageLoader等

47、LruCache的原理

- 1. 利用 LinkedHashMap 持有对象的强引用，按照Lru算法进行对象淘汰。
- 2. 从表尾访问数据，表头删除数据。
- 3. 访问的数据存在时，就将数据移动到表尾；不存在就在表尾新建数据。
- 4. 链表容量达到阈值后，从表头移除数据。

48、为什么选择会选择LinkedHashMap？

- 1. LinkedHashMap 的特性决定。
- 2. 构造方法中有一个标志位，=true时，会按照访问顺序进行排序；否则按照插入顺序进行排序。

# HashTable

49、Hashtable和HashMap的区别

- 1. 作者： HashMap 的作者比 Hashtable 的作者多了一个人： Doug Lea 写了 util.concurrent 包并且著有 并发编程圣经： Concurrent Programming in Java
- 2. 诞生时间： HashMap 产生于 JDK1.2 相比于 Hashtable 更晚。
- 3. 弃用状况： Hashtable 基本上已经被弃用： 1- Hashtable 是 线程安全 ， 效率比较低。 2- Hashtable 没有遵循 驼峰命名法
- 4. 父类： HashMap 继承自 AbstractMap ， Hashtable 继承自 Dictionary
- 5. 接口数量： Hashtable 比 HashMap 多剔红了 两个接口 ： elements和contains
- 6. key和value是否为null： Hashtable 不允许 key和value 为 NULL ， HashMap 支持： key=null的键只能有一个 ， get()返回为null，可能是value为null，也可能是没有该key，需要通过containsKey来判断是否具有某个key
- 7. 线程安全性- Hashtable 是 线程安全(每个方法都加入Synchronized) ， HashMap 是 非线程安全 的。 HashMap 效率比 Hashtable 高很多，而且需要自己进行同步处理。如果需要 线程安全 可以使用 ConcurrentHashMap ， 也比 Hashtable 效率高很多倍。
- 8. 遍历方式- Hashtable 使用老旧的 Enumeration 的方式， HashMap 使用 Iterator迭代器
- 9. 初始容量和扩容方式： Hashtable初始为11， 扩容是  $2 * n + 1$  ， HashMap初始为16， 扩容是  $2 * n$
- 10. 计算 hash值 的方式不同： HashMap 比 Hashtable 的计算效率更高。（涉及到位运算，以及通过额外计算打散数据来减少hash冲突的问题）  
相同1： 两者都实现了： Cloneable(可复制)、Serializable(可序列化)

# HashSet

50、HashMap和HashSet的相似与区别

HashMap	HashSet
Java Collection Framework重要成员	Java Collection Framework重要成员



HashMap	HashSet
实现了Map接口	实现Set接口
存储键值对	仅存储对象
调用put（）向map中添加元素	调用add（）方法向Set中添加元素
HashMap使用键（Key） 计算HashCode	HashSet使用成员对象来计算hashCode值，对于两个对象来说hashCode可能相同 (因此使用前要确保重写hashCode（）方法和equals（）方法)
HashMap较快， 因为它是使用唯一的键获取对象	HashSet较慢
非线程安全	非线程安全
底层Hash存储机制相同	底层Hash存储机制相同

## 细节面试题补充

51、最容易导致HashMap中Hash冲突的方法是什么？

创建一个类，让你挂起哈希函数返回一个最糟糕的结果---常数。

52、哈希方法返回常数会造成什么情况？

- 1. 造成极度频繁的Hash冲突。
- 2. 数据会都集中在某一桶的链表中，导致链表很长。
- 3. 插入、删除都会在该桶中进行遍历比较Key(equals)，效率极其低下。

如果有帮助，请点个赞万分感谢！

## 参考资料

- 1. [Map 综述一: 彻头彻尾理解 HashMap](#)
- 2. [Java 中的 ==、equals、hashCode 的区别与联系](#)
- 3. [解决hash冲突的三个方法](#)
- 4. [HashMap中hash方法分析](#)
- 5. [JDK1.8 重新认识HashMap](#)
- 6. [Java 8 HashMap键与Comparable接口](#)