在**[Java](http://lib.csdn.net/base/17" \o "Java EE知识库" \t "http://blog.csdn.net/abcd1430/article/details/_blank)**编程语言中，最基本的结构就是两种，一种是数组，一种是模拟指针(引用),所有的数据结构都可以用这两个基本结构构造，HashMap也一样。当程序试图将多个 key-value 放入 HashMap 中时，以如下代码片段为例：

HashMap<String,Object> m=new HashMap<String,Object>();   
m.put("a", "rrr1");   
m.put("b", "tt9");   
m.put("c", "tt8");   
m.put("d", "g7");   
m.put("e", "d6");   
m.put("f", "d4");   
m.put("g", "d4");   
m.put("h", "d3");   
m.put("i", "d2");   
m.put("j", "d1");   
m.put("k", "1");   
m.put("o", "2");   
m.put("p", "3");   
m.put("q", "4");   
m.put("r", "5");   
m.put("s", "6");   
m.put("t", "7");   
m.put("u", "8");   
m.put("v", "9");

        HashMap 采用一种所谓的“Hash 算法”来决定每个元素的存储位置。当程序执行 map.put(String,Obect)方法 时，系统将调用String的 hashCode() 方法得到其 hashCode 值——每个 Java 对象都有 hashCode() 方法，都可通过该方法获得它的 hashCode 值。得到这个对象的 hashCode 值之后，系统会根据该 hashCode 值来决定该元素的存储位置。源码如下:

1. **public V put(K key, V value) {**
2. **if (key == null)**
3. **return putForNullKey(value);**
4. **int hash = hash(key.hashCode());**
5. **int i = indexFor(hash, table.length);**
6. **for (Entry<K,V> e = table[i]; e != null; e = e.next) {**
7. Object k;
8. //判断当前确定的索引位置是否存在相同hashcode和相同key的元素，如果存在相同的hashcode和相同的key的元素，那么新值覆盖原来的旧值，并返回旧值。
9. //如果存在相同的hashcode，那么他们确定的索引位置就相同，这时判断他们的key是否相同，如果不相同，这时就是产生了hash冲突。
10. //Hash冲突后，那么HashMap的单个bucket里存储的不是一个 Entry，而是一个 Entry 链。
11. //系统只能必须按顺序遍历每个 Entry，直到找到想搜索的 Entry 为止——如果恰好要搜索的 Entry 位于该 Entry 链的最末端（该 Entry 是最早放入该 bucket 中），
12. //那系统必须循环到最后才能找到该元素。
13. **if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {**
14. V oldValue = e.value;
15. e.value = value;
16. **return oldValue;**
17. }
18. }
19. modCount++;
20. addEntry(hash, key, value, i);
21. **return null;**
22. }

       上面程序中用到了一个重要的内部接口：Map.Entry，每个 Map.Entry 其实就是一个 key-value 对。从上面程序中可以看出：当系统决定存储 HashMap 中的 key-value 对时，完全没有考虑 Entry 中的 value，仅仅只是根据 key 来计算并决定每个 Entry 的存储位置。这也说明了前面的结论：我们完全可以把 Map 集合中的 value 当成 key 的附属，当系统决定了 key 的存储位置之后，value 随之保存在那里即可.HashMap程序经过我改造，我故意的构造出了hash冲突现象，因为HashMap的初始大小16,但是我在hashmap里面放了超过16个元素，并且我屏蔽了它的resize()方法。不让它去扩容。这时HashMap的底层数组Entry[]   table结构如下:

       Hashmap里面的bucket出现了单链表的形式，散列表要解决的一个问题就是散列值的冲突问题，通常是两种方法：链表法和开放地址法。链表法就是将相同hash值的对象组织成一个链表放在hash值对应的槽位；开放地址法是通过一个探测算法，当某个槽位已经被占据的情况下继续查找下一个可以使用的槽位。java.util.HashMap采用的链表法的方式，链表是单向链表。形成单链表的核心代码如下：

1. **void addEntry(int hash, K key, V value, int bucketIndex) {**
2. Entry<K,V> e = table[bucketIndex];
3. table[bucketIndex] = **new Entry<K,V>(hash, key, value, e);**
4. **if (size++ >= threshold)**
5. resize(2 \* table.length);
6. bsp;

     上面方法的代码很简单，但其中包含了一个设计：系统总是将新添加的 Entry 对象放入 table 数组的 bucketIndex 索引处——如果 bucketIndex 索引处已经有了一个 Entry 对象，那新添加的 Entry 对象指向原有的 Entry 对象（产生一个 Entry 链），如果 bucketIndex 索引处没有 Entry 对象，也就是上面程序代码的 e 变量是 null，也就是新放入的 Entry 对象指向 null，也就是没有产生 Entry 链。

       HashMap里面没有出现hash冲突时，没有形成单链表时，hashmap查找元素很快,get()方法能够直接定位到元素，但是出现单链表后，单个bucket 里存储的不是一个 Entry，而是一个 Entry 链，系统只能必须按顺序遍历每个 Entry，直到找到想搜索的 Entry 为止——如果恰好要搜索的 Entry 位于该 Entry 链的最末端（该 Entry 是最早放入该 bucket 中），那系统必须循环到最后才能找到该元素。

       当创建 HashMap 时，有一个默认的负载因子（load factor），其默认值为 0.75，这是时间和空间成本上一种折衷：增大负载因子可以减少 Hash 表（就是那个 Entry 数组）所占用的内存空间，但会增加查询数据的时间开销，而查询是最频繁的的操作（HashMap 的 get() 与 put() 方法都要用到查询）；减小负载因子会提高数据查询的性能，但会增加 Hash 表所占用的内存空间。

一、HashMap概述

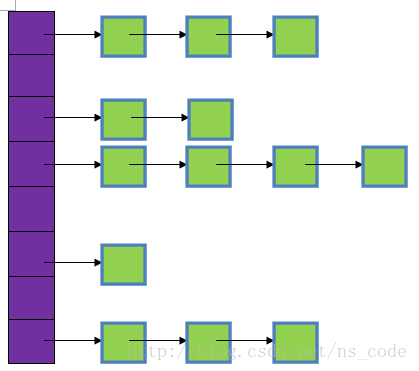
　　HashMap基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了不同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

　　值得注意的是HashMap不是线程安全的，如果想要线程安全的HashMap，可以通过Collections类的静态方法synchronizedMap获得线程安全的HashMap。

Map map = Collections.synchronizedMap(new HashMap());

二、HashMap的数据结构

　　HashMap的底层主要是基于数组和链表来实现的，它之所以有相当快的查询速度主要是因为它是通过计算散列码来决定存储的位置。HashMap中主要是通过key的hashCode来计算hash值的，只要hashCode相同，计算出来的hash值就一样。如果存储的对象对多了，就有可能不同的对象所算出来的hash值是相同的，这就出现了所谓的hash冲突。学过数据结构的同学都知道，解决hash冲突的方法有很多，HashMap底层是通过链表来解决hash冲突的。



 图中，紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中。

其中loadFactor加载因子是表示Hsah表中元素的填满的程度.

若:加载因子越大,填满的元素越多,好处是,空间利用率高了,但:冲突的机会加大了.链表长度会越来越长,查找效率降低。

反之,加载因子越小,填满的元素越少,好处是:冲突的机会减小了,但:空间浪费多了.表中的数据将过于稀疏（很多空间还没用，就开始扩容了）

冲突的机会越大,则查找的成本越高.

因此,必须在 "冲突的机会"与"空间利用率"之间寻找一种平衡与折衷. 这种平衡与折衷本质上是数据结构中有名的"时-空"矛盾的平衡与折衷.

　　如果机器内存足够，并且想要提高查询速度的话可以将加载因子设置小一点；相反如果机器内存紧张，并且对查询速度没有什么要求的话可以将加载因子设置大一点。不过一般我们都不用去设置它，让它取默认值0.75就好了。

当HashMap中的元素越来越多的时候，hash冲突的几率也就越来越高，因为数组的长度是固定的。所以为了提高查询的效率，就要对HashMap的数组进行扩容，数组扩容这个操作也会出现在ArrayList中，这是一个常用的操作，而在HashMap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。

　　那么HashMap什么时候进行扩容呢？当HashMap中的元素个数超过数组大小\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，这是一个折中的取值。也就是说，默认情况下，数组大小为16，那么当HashMap中元素个数超过16\*0.75=12（这个值就是代码中的threshold值，也叫做临界值）的时候，就把数组的大小扩展为 2\*16=32，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知HashMap中元素的个数，那么预设元素的个数能够有效的提高HashMap的性能。

#### **HashMap的工作原理**

HashMap基于hashing原理，我们通过put()和get()方法储存和获取对象。当我们将键值对传递给put()方法时，它调用键对象的hashCode()方法来计算hashcode，让后找到bucket位置来储存值对象。当获取对象时，通过键对象的equals()方法找到正确的键值对，然后返回值对象。HashMap使用链表来解决碰撞问题，当发生碰撞了，对象将会储存在链表的下一个节点中。 HashMap在每个链表节点中储存键值对对象。

当两个不同的键对象的hashcode相同时会发生什么？ 它们会储存在同一个bucket位置的链表中。键对象的equals()方法用来找到键值对。

### **HashMap和Hashtable的区别**

HashMap和Hashtable都实现了Map接口，但决定用哪一个之前先要弄清楚它们之间的分别。主要的区别有：线程安全性，同步(synchronization)，以及速度。

1. HashMap几乎可以等价于Hashtable，除了HashMap是非synchronized的，并可以接受null(HashMap可以接受为null的键值(key)和值(value)，而Hashtable则不行)。
2. HashMap是非synchronized，而Hashtable是synchronized，这意味着Hashtable是线程安全的，多个线程可以共享一个Hashtable；而如果没有正确的同步的话，多个线程是不能共享HashMap的。Java 5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。
3. 另一个区别是HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。这条同样也是Enumeration和Iterator的区别。
4. 由于Hashtable是线程安全的也是synchronized，所以在单线程环境下它比HashMap要慢。如果你不需要同步，只需要单一线程，那么使用HashMap性能要好过Hashtable。
5. HashMap不能保证随着时间的推移Map中的元素次序是不变的。

### **我们能否让HashMap同步？**

HashMap可以通过下面的语句进行同步：  
Map m = Collections.synchronizeMap(hashMap);

### **HashMap和HashSet的区别**

HashMap和HashSet的区别是Java面试中最常被问到的问题。如果没有涉及到Collection框架以及多线程的面试，可以说是不完整。而Collection框架的问题不涉及到HashSet和HashMap，也可以说是不完整。HashMap和HashSet都是collection框架的一部分，它们让我们能够使用对象的集合。collection框架有自己的接口和实现，主要分为Set接口，List接口和Queue接口。它们有各自的特点，Set的集合里不允许对象有重复的值，List允许有重复，它对集合中的对象进行索引，Queue的工作原理是FCFS算法(First Come, First Serve)。

首先让我们来看看什么是HashMap和HashSet，然后再来比较它们之间的分别。

### **什么是HashSet**

HashSet实现了Set接口，它不允许集合中有重复的值，当我们提到HashSet时，第一件事情就是在将对象存储在HashSet之前，要先确保对象重写equals()和hashCode()方法，这样才能比较对象的值是否相等，以确保set中没有储存相等的对象。如果我们没有重写这两个方法，将会使用这个方法的默认实现。

public boolean add(Object o)方法用来在Set中添加元素，当元素值重复时则会立即返回false，如果成功添加的话会返回true。

### **什么是HashMap**

HashMap实现了Map接口，Map接口对键值对进行映射。Map中不允许重复的键。Map接口有两个基本的实现，HashMap和TreeMap。TreeMap保存了对象的排列次序，而HashMap则不能。HashMap允许键和值为null。HashMap是非synchronized的，但collection框架提供方法能保证HashMap synchronized，这样多个线程同时访问HashMap时，能保证只有一个线程更改Map。

public Object put(Object Key,Object value)方法用来将元素添加到map中。

### **HashSet和HashMap的区别**

|  |  |
| --- | --- |
| \*HashMap\* | \*HashSet\* |
| HashMap实现了Map接口 | HashSet实现了Set接口 |
| HashMap储存键值对 | HashSet仅仅存储对象 |
| 使用put()方法将元素放入map中 | 使用add()方法将元素放入set中 |
| HashMap中使用键对象来计算hashcode值 | 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，所以equals()方法用来判断对象的相等性，如果两个对象不同的话，那么返回false |
| HashMap比较快，因为是使用唯一的键来获取对象 | HashSet较HashMap来说比较慢 |

**“你用过HashMap吗？” “什么是HashMap？你为什么用到它？”**

几乎每个人都会回答“是的”，然后回答HashMap的一些特性，譬如HashMap可以接受null键值和值，而Hashtable则不能；HashMap是非synchronized;HashMap很快；以及HashMap储存的是键值对等等。这显示出你已经用过HashMap，而且对它相当的熟悉。但是面试官来个急转直下，从此刻开始问出一些刁钻的问题，关于HashMap的更多基础的细节。面试官可能会问出下面的问题：

**“你知道HashMap的工作原理吗？” “你知道HashMap的get()方法的工作原理吗？”**

你也许会回答“我没有详查标准的Java API，你可以看看Java源代码或者Open JDK。”“我可以用Google找到答案。”

但一些面试者可能可以给出答案，“HashMap是基于hashing的原理，我们使用put(key, value)存储对象到HashMap中，使用get(key)从HashMap中获取对象。当我们给put()方法传递键和值时，我们先对键调用hashCode()方法，返回的hashCode用于找到bucket位置来储存Entry对象。”这里关键点在于指出，HashMap是在bucket中储存键对象和值对象，作为Map.Entry。这一点有助于理解获取对象的逻辑。如果你没有意识到这一点，或者错误的认为仅仅只在bucket中存储值的话，你将不会回答如何从HashMap中获取对象的逻辑。这个答案相当的正确，也显示出面试者确实知道hashing以及HashMap的工作原理。但是这仅仅是故事的开始，当面试官加入一些Java程序员每天要碰到的实际场景的时候，错误的答案频现。下个问题可能是关于HashMap中的碰撞探测(collision detection)以及碰撞的解决方法：

**“当两个对象的hashcode相同会发生什么？”** 从这里开始，真正的困惑开始了，一些面试者会回答因为hashcode相同，所以两个对象是相等的，HashMap将会抛出异常，或者不会存储它们。然后面试官可能会提醒他们有equals()和hashCode()两个方法，并告诉他们两个对象就算hashcode相同，但是它们可能并不相等。一些面试者可能就此放弃，而另外一些还能继续挺进，他们回答“因为hashcode相同，所以它们的bucket位置相同，‘碰撞’会发生。因为HashMap使用链表存储对象，这个Entry(包含有键值对的Map.Entry对象)会存储在链表中。”这个答案非常的合理，虽然有很多种处理碰撞的方法，这种方法是最简单的，也正是HashMap的处理方法。但故事还没有完结，面试官会继续问：

**“如果两个键的hashcode相同，你如何获取值对象？”** 面试者会回答：当我们调用get()方法，HashMap会使用键对象的hashcode找到bucket位置，然后获取值对象。面试官提醒他如果有两个值对象储存在同一个bucket，他给出答案:将会遍历链表直到找到值对象。面试官会问因为你并没有值对象去比较，你是如何确定确定找到值对象的？除非面试者直到HashMap在链表中存储的是键值对，否则他们不可能回答出这一题。

其中一些记得这个重要知识点的面试者会说，找到bucket位置之后，会调用keys.equals()方法去找到链表中正确的节点，最终找到要找的值对象。完美的答案！

许多情况下，面试者会在这个环节中出错，因为他们混淆了hashCode()和equals()方法。因为在此之前hashCode()屡屡出现，而equals()方法仅仅在获取值对象的时候才出现。一些优秀的开发者会指出使用不可变的、声明作final的对象，并且采用合适的equals()和hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择。

如果你认为到这里已经完结了，那么听到下面这个问题的时候，你会大吃一惊。**“如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？”**除非你真正知道HashMap的工作原理，否则你将回答不出这道题。默认的负载因子大小为0.75，也就是说，当一个map填满了75%的bucket时候，和其它集合类(如ArrayList等)一样，将会创建原来HashMap大小的两倍的bucket数组，来重新调整map的大小，并将原来的对象放入新的bucket数组中。这个过程叫作rehashing，因为它调用hash方法找到新的bucket位置。

如果你能够回答这道问题，下面的问题来了：**“你了解重新调整HashMap大小存在什么问题吗？”**你可能回答不上来，这时面试官会提醒你当多线程的情况下，可能产生条件竞争(race condition)。

当重新调整HashMap大小的时候，确实存在条件竞争，因为如果两个线程都发现HashMap需要重新调整大小了，它们会同时试着调整大小。在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，HashMap并不会将元素放在链表的尾部，而是放在头部，这是为了避免尾部遍历(tail traversing)。如果条件竞争发生了，那么就死循环了。这个时候，你可以质问面试官，为什么这么奇怪，要在多线程的环境下使用HashMap呢？：）

1. **为什么String, Interger这样的wrapper类适合作为键？** String, Interger这样的wrapper类作为HashMap的键是再适合不过了，而且String最为常用。因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还有其他的优点如线程安全。如果你可以仅仅通过将某个field声明成final就能保证hashCode是不变的，那么请这么做吧。因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能。
2. **我们可以使用自定义的对象作为键吗？** 这是前一个问题的延伸。当然你可能使用任何对象作为键，只要它遵守了equals()和hashCode()方法的定义规则，并且当对象插入到Map中之后将不会再改变了。如果这个自定义对象时不可变的，那么它已经满足了作为键的条件，因为当它创建之后就已经不能改变了。
3. **我们可以使用CocurrentHashMap来代替Hashtable吗？**这是另外一个很热门的面试题，因为ConcurrentHashMap越来越多人用了。我们知道Hashtable是synchronized的，但是ConcurrentHashMap同步性能更好，因为它仅仅根据同步级别对map的一部分进行上锁。ConcurrentHashMap当然可以代替HashTable，但是HashTable提供更强的线程安全性。