Java 集合框架在面试中必问

主要内容：

1. Arraylist 与 LinkedList 异同
2. ArrayList 与 Vector 区别
3. HashMap的底层实现
4. HashMap 和 Hashtable 的区别
5. HashMap 的长度为什么是2的幂次方
6. HashSet 和 HashMap 区别
7. ConcurrentHashMap 和 Hashtable 的区别
8. ConcurrentHashMap线程安全的具体实现方式/底层具体实现
9. 集合框架底层数据结构总结

Java Collections 框架中包含了大量集合接口，以及这些接口的实现类和它们的操作方法（例如查找、排序反转、替换、复制、取最小元素、取最大元素等），具体而言，主要提供了列表、集合、栈、Map 等，List、Set、Stack、Queue 都继承自 Collection 接口
Collection 是整个集合框架的基础，里面存储了一组对象，表示不同类型的 Collections，它的作用是只是提供维护一组对象的基本接口而已。

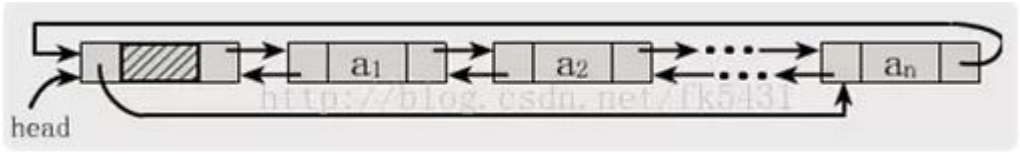
1. Arraylist 与 LinkedList 异同

- **1. 是否保证线程安全：** ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；
- **2. 底层数据结构：** Arraylist 底层使用的是 Object 数组；LinkedList 底层使用的是双向循环链表数据结构；
- **3. 插入和删除是否受元素位置的影响：** ① **ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。**比如：执行 `add(E e)`方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是 O(1)。但是如果要在指定位置 i 插入和删除元素的话（`add(intindex,E element)`）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。 ②**LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 O（1）而数组为近似 O（n）。**

- **4. 是否支持快速随机访问：** **LinkedList 不支持高效的随机元素访问**，而 **ArrayList** 实现了 RandmoAccess 接口，所以**有随机访问功能。**快速随机访问就是通过元素的序号快速获取元素对象(对应于 `get(intindex)`方法)。
- **5. 内存空间占用：**ArrayList 的空间浪费主要体现在在 list 列表的结尾会预留一定的容量空间，而 LinkedList 的空间花费则体现在它的每一个元素都需要消耗比 ArrayList 更多的空间（因为要存放直接后继和直接前驱以及数据）。

补充：数据结构基础之双向链表

双向链表也叫双链表，是链表的一种，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。一般我们都构造双向循环链表，如下图所示，同时下图也是 LinkedList 底层使用的是双向循环链表数据结构。



ArrayList 与 Vector 区别

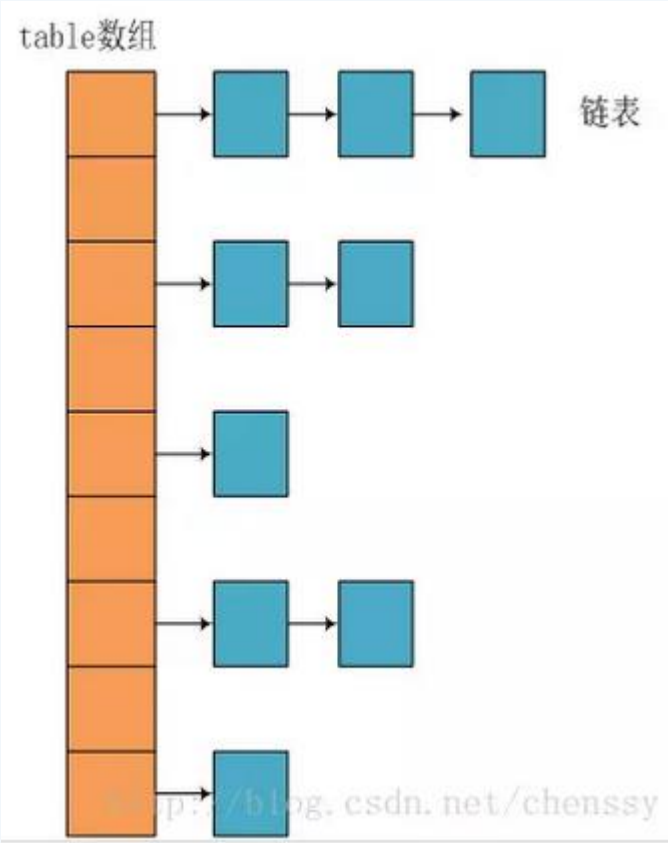
Vector 类的所有方法都是同步的。可以由两个线程安全地访问一个 Vector 对象、但是一个线程访问 Vector 的话代码要在同步操作上耗费大量的时间。Arraylist 不是同步的，所以在**不需要保证线程安全时时建议使用 Arraylist。**

2. HashMap 的底层实现

JDK1.8 之前

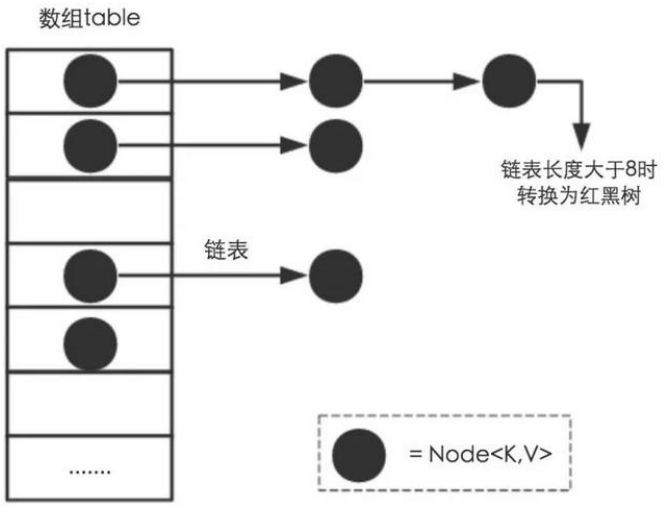
JDK1.8 之前 HashMap 由 **数组+链表** 组成的（**“链表散列”** 即数组和链表的结合体），数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（HashMap 采用 **“拉链法也就是链地址法”** 解决冲突），如果定位到的数组位置不含链表（当前 entry 的 next 指向 null ）,那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度依然为 O(1)，因为最新的 Entry 会插入链表头部，即需要简单改变引用链即可，而对于查找操作来讲，此时就需要遍历链表，然后通过 key 对象的 equals 方法逐一比对查找.

所谓 **“拉链法”** 就是将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。



JDK1.8 之后

相比于之前的版本， JDK1.8 之后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间。



TreeMap、TreeSet 以及 JDK1.8 之后的 HashMap 底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

HashMap 和 Hashtable 的区别

1. **线程是否安全：** HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过 `synchronized` 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；
2. **效率：** 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；
3. **对 Null key 和 Null value 的支持：** HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。

4. **初始容量大小和每次扩充容量大小的不同 ：** ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为 11，之后每次扩充，容量变为原来的 2n+1。HashMap 默认的初始化大小为 16。之后每次扩充，容量变为原来的 2 倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为 2 的幂次方大小。也就是说 HashMap 总是使用 2 的幂次方作为哈希表的大小，后面会介绍到为什么是 2 的幂次方。
5. **底层数据结构：** JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

HashMap 的长度为什么是 2 的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀，每个链表/红黑树长度大致相同。这个实现就是把数据存到哪个链表/红黑树中的算法。

这个算法应该如何设计呢？

我们首先可能会想到采用%取余的操作来实现。但是，重点来了： **“取余(%)操作中如果除数是 2 的幂次则等价于与其除数减一的与(&)操作（也就是说 hash%length==hash&(length-1)的前提是 length 是 2 的 n 次方；）。”** 并且 **采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是 2 的幂次方。**

HashSet 和 HashMap 区别

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，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相对于HashSet较快，因为它是使用唯一的键获取对象	HashSet较HashMap来说比较慢

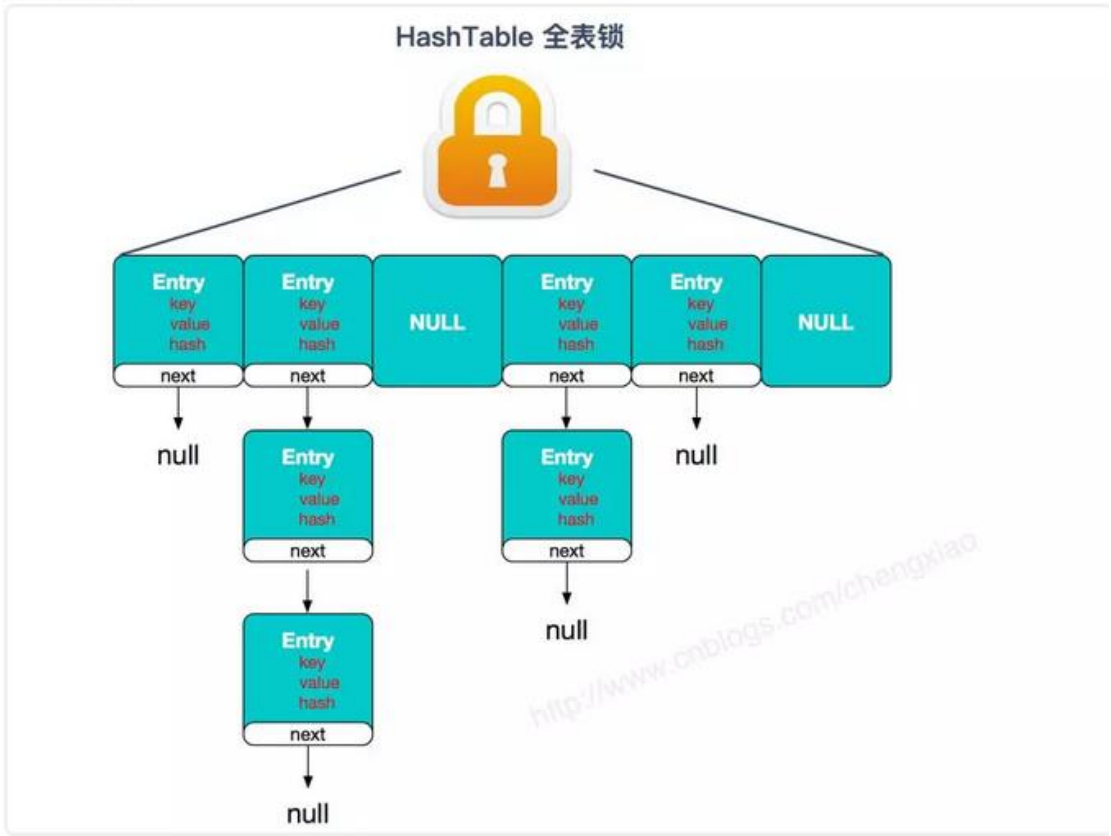
ConcurrentHashMap 和 Hashtable 的区别

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

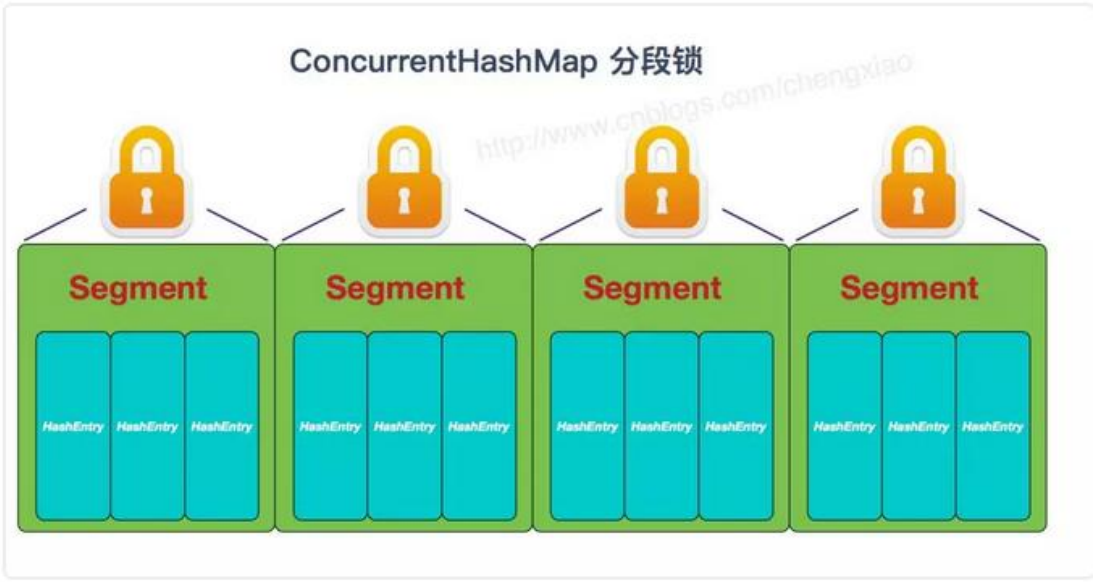
- **底层数据结构：** JDK1.7 的 ConcurrentHashMap 底层采用 **分段的数组+链表** 实现，JDK1.8 采用的数据结构跟 HashMap1.8 的结构类似，**数组+链表/红黑二叉树。** Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 **数组+链表** 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；
- **实现线程安全的方式（重要）： ① 在 JDK1.7 的时候，ConcurrentHashMap（分段锁）** 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。（默认分配 16 个 Segment，比 Hashtable 效率提高 16 倍。） **到了 JDK1.8 的时候已经摒弃了 Segment 的概念，而是直接用 Node 数组+链表/红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6 以后 对 synchronized 锁做了很多优化）** 整个看起来就像是优化过且线程安全的 HashMap，虽然在 JDK1.8 中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；

② **Hashtable(同一把锁)** :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争越激烈效率越低。

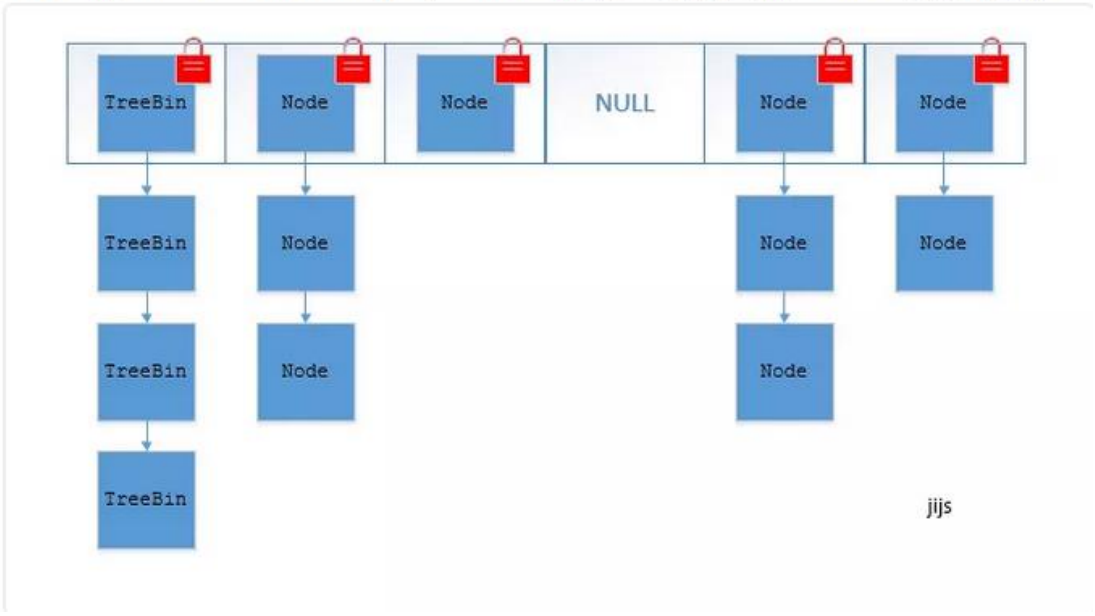
HashTable:

- 

JDK1.7 的 ConcurrentHashMap ：



JDK1.8 的 ConcurrentHashMap （TreeBin: 红黑二叉树节点； Node: 链表节点）：



ConcurrentHashMap 线程安全的具体实现方式/底层具体实现

JDK1.7（上面有示意图）

首先将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也能被其他线程访问。

ConcurrentHashMap 是由 Segment 数组结构和 HahEntry 数组结构组成。

Segment 实现了 ReentrantLock,所以 Segment 是一种可重入锁,扮演锁的角色。HashEntry 用于存储键值对数据。

```
1. static class Segment<K,V> extends ReentrantLock implements Serializable {
2. }
```

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和 HashMap 类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护着一个 HashEntry 数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 的锁。

JDK1.8 （上面有示意图）

ConcurrentHashMap 取消了 Segment 分段锁，采用 CAS 和 synchronized 来保证并发安全。数据结构跟 HashMap1.8 的结构类似，数组+链表/红黑二叉树。

synchronized 只锁定当前链表或红黑二叉树的首节点，这样只要 hash 不冲突，就不会产生并发，效率又提升 N 倍。

集合框架底层数据结构总结

Collection

1. List（有序 Collection，按照对象进入的顺序保存对象，所以能对列表中每个元素的插入和删除的位置进行精准的控制）

- **Arraylist：** Object 数组

- **Vector：** Object 数组
- **LinkedList：** 双向循环链表

2. Set（数学意义上的集合，特点存入的元素不能重复，因此需要重写 equals 方法保证元素的唯一性）

- **HashSet（无序，唯一）：**基于 HashMap 实现的，底层采用 HashMap 来保存元素。HashMap 底层数据结构见下。
- **LinkedHashSet：** LinkedHashSet 继承与 HashSet，并且其内部是通过 LinkedHashMap 来实现的。有点类似于我们之前说的 LinkedHashMap 其内部是基于 Hashmap 实现一样，不过还是有一点点区别的。
- **TreeSet（有序，唯一）：** 红黑树(自平衡的排序二叉树。)实现了 SortedSet 接口

Map（提供一种从键映射到值的数据结构，其中值是可以重复的，但是键是不能重复的）

- **HashMap：** JDK1.8 之前 HashMap 由数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）.JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，将链表转化为红黑树，以减少搜索时间
- **LinkedHashMap:** LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外，LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。详细可以查看：

《 LinkedHashMap 源 码 详 细 分 析 （ JDK1.8 ） 》 ： https://www.imooc.com/article/22931

- **HashTable:** 数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的
- **TreeMap:** 红黑树（自平衡的排序二叉树）

关于 HashMap 和 Hashtable 存储键值方面的思考

使用 HashMap 和 HashTable 这两个容器时有一些限制：不能用来存储重复的键。即每个键只能唯一的映射一个值，当有重复的键的时候，不会创建新的映射关系，而会使用先前的键值

向 hashmap 中添加<key,value>需要经过以下步骤：

调用 key 的 hashCode（）生成一个 hash 值 h1，如果这个 h1 在 hashmap 中不存在，那么直接把<key,value>添加到 HashMap 中；如果 h1 存在，那么找出 HashMap 中所有 hash 值为 h1 的 key，然后调用 key 的 equals（）方法判断当前的 key 是否与已经存在的 key 值相同。如果 equals（）返回 true，说明当前需要添加的 key 已经存在，那么 HashMap 会使用新的 value 值覆盖原来旧的 value 值；如果返回为 false，说明新增加的 key 在 HashMap 中不存在，那么就新建新的映射关系。

-

附加：**HashSet 如何检查重复**

当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashcode 值来判断对象加入的位置，同时也会与其他加入的对象的 hashcode 值作比较，如果没有相符的 hashcode ，HashSet 会假设对象没有重复出现。但是如果发现有相同 hashcode 值的对象，这时会调用 equals（）方法来检查 hashcode 相等的对象是否真的相同。如果两者相同，HashSet 就不会让加入操作成功。

hashCode（）与 equals（）的相关规定

1. 如果两个对象相等，则 hashcode 一定也是相同的

2. 两个对象相等,对两个 equals 方法返回 true

3. 两个对象有相同的 hashcode 值，它们也不一定是相等的

4. 综上，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖

5. hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode()，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

==与 equals 的区别

==：它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象。

equals() ：它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

- 情况 1：类没有覆盖 equals()方法。则通过 equals()比较该类的两个对象时，等价于通过“==”比较这两个对象。

- 情况 2：类覆盖了 equals()方法。一般，我们都覆盖 equals()方法来两个对象的内容相等；若它们的内容相等，则返回 true(即，认为这两个对象相等)。