经典问题：你能说一下HashMap吗？

jdk1.7和1.8的区别？

jdk1.7中：

- 底层数据结构：数组+链表
- 扩容：capacity和Loadfactor—0.75
  - 扩容：创建一个新的Entry空数组，长度是原来数组长度的2倍
  - ReHash：遍历元Entry数组，把所有的Entry重新Hash到新数组上
- 头插法：
- 线程不安全：**当我们要开始扩容时，会调用resize方法，进而调用transfer方法，多线程操作HashMap时可能引起死循环，原因是扩容转移后前后链表顺序倒置，在转移过程中修改了原来链表中节点的引用关系。**

jdk1.8中：

- 底层数据结构：数组+链表+红黑树
- 扩容：创建一个新的Node节点，长度是原来的2倍
  - 当数组长度达到0.75*capacity时，会进行扩容，同时当链表长度大于8但是数组长度小于64时，也会进行扩容
  - 当链表长度大于8，数组长度大于64时，链表会转化为红黑树
- 尾插法
- 线程不安全：在多线程环境下，执行put操作时会发生数据覆盖的情况。

为什么我们重写equals方法时，还要建议我们重写hashcode方法？

- 为了保证相同的对象有相同的hash值，不同的对象有不同的hash值
- 在HashMap中，当有相同的hashcode值会被放在一条链表上，如果没有重写equals方法，我们就取不到我们想要的value值。

1 HashMap的初始容量是2的次幂及扩容是2倍形式？

- HashMap初始值是2的次幂，扩容也是2倍形式进行扩容的，是因为容量是2的次幂时，可以使得添加的元素可以均匀的分布在HashMap中的数组上，减少**hash碰撞**，避免形成链表的结构，使得查询效率降低。

2 HashMap的hash算法可以用%取余运算吗？

&运算时二进制逻辑运算符，是计算机能直接执行的操作符，而%是Java处理整形浮点型所定义的操作符，底层也是这些逻辑运算符的实现，效率的差别可想而知，效率相差大概10倍。

2 HashMap的加载因子

加载因子为什么是0.75？

- 负载因子是0.75的时候，空间利用率比较高，而且避免了相当多的Hash冲突，使得底层的链表或者是红黑树的高度比较低，提升了空间效率。

加载因子可以调整吗？
可以调整,hashmap运行用户输入一个加载因子

```
public HashMap(int initialCapacity, float loadFactor) {
}
```

加载因子为0.5或者1，会怎么样？能大于1吗

- 加载因子小，那么我们扩容的频率就会变高，但是hash碰撞的概率会低很多，相应的**链表长度就普遍很低**，那么我们的查询速度变快了，但是**内存消耗确实大**了。
- 加载因子大，hash碰撞的概率变高，每个**链表长度都很长**，**查询速度变慢**，但是由于我们不怎么扩容，内存是节省了不少，毕竟扩容一次就翻一倍。

如果在实际开发中，内存非常充裕，可以将加载因子调小。如果内存非常吃紧，可以稍微调大一点。

3 ConcurrentHashMap（读无锁，写同步）

在jdk1.7中数据结构：数组+链表（分段锁实现，并发度就是Segment数组的大小）

- ConcurrentHashMap定义了一个内部类Segment数组，一个Segment里面包含一个HashEntry数组，每个HashEntry数组是一个链表的数据结构。
- 其中Segment继承ReentrantLock，当我们要执行**put**操作时，每当一个线程占用锁访问一个Segment时，不会影响到其他的Segment。
- 当我们执行**get**操作时，此时没有加锁，而是把共享变量都用了volatile进行修饰，保证了内存可见性。保证每个线程读取的值是最新值。
- 具体put和get的步骤：
  - put操作
    - 将key通过hash算法定位到一个Segment，尝试自旋获取锁
    - 如果重试次数达到了一定阈值，则改为阻塞锁进行获取。
  - get操作
    - 只需要将 Key 通过 Hash 之后定位到具体的 Segment ，再通过一次 Hash 定位到具体的元素上。

在jdk1.8中数据结构：数组+链表+红黑树（采用CAS+synchronized实现线程安全）

- ConcurrentHashMap定义了一个内部类Node节点，类似于jdk1.8的HashMap。
- 当我们进行put操作时，刚开始是没有加锁的，使用cas无锁操作进行put
- 具体put和get的步骤：
  - put操作
    - 首先根据key，计算出hash值，并判断是否需要初始化Node数组
    - 根据当前的key定位到一个Node节点，如果为空，则用CAS尝试写入，失败则自旋保证成功。
    - 再判断Node数组是否需要扩容
    - 如果都不满足，则用Sychronized锁进行写入数据
    - 如果数量大于红黑树的阈值，则要转换为红黑树
  - get操作
    - 只需要将 Key 通过 Hash 之后定位到具体的 Node节点，遍历链表或者红黑树获取元素。