# Java面试题

Hashmap在1.7和1.8的区别

是线程不安全的，若要线程安全使用concurrenthashmap ，或用collections里的静态同步方法。或加synchronzed 或lock

集合

synchronzed

多线程

JVM

Spring源码

1. 面向对象四大特征：

多态 继承 抽象 封装

1. 抽象类和接口类什么区别

抽象类：

接口类：

1. Java jdk8的接口新增的一些特性
2. 重载跟重写的区别
3. 父类的private方法不可重写，final修饰的不可重写
4. 多线程用过吗？线程的状态

创建、就绪、运行、等待、时间等待、死亡

1. 线程池用过吗？参数各代表什么含义？
2. 核心线程数已满，最大线程数未满，队列有位置，再来一个任务时会怎么处理？
3. Synchronized用过吗？用在静态方法（锁.class文件）与非静态方法（锁对象）上什么区别？
4. 重入锁与synchronized有什么区别？
5. 可重入锁、公平锁、非公平锁、悲观锁、乐观锁
6. Synchronized是悲观锁还是乐观锁？
7. 数据库ACID
8. 事务隔离级别 四个
9. 可重复读存在什么问题？幻读
10. Mysql索引了解吗？有哪些索引类别
11. **执行引擎了解哪些？**
12. **Mysql怎么做分页？用limit做分页有什么问题？（mysql引擎和分页还没有搞明白）**
13. Redis
14. 哪些数据类型结构，设置过缓存时间吗
15. 缓存穿透是什么？
16. Redis是单线程还是多线程？

单线程，为什么单线程读取速度还这么快？

1. 各种排序算法
2. http熟悉吗
3. 500什么意思 404 302暂时重定向
4. TCP为什么三次握手
5. 项目：：
6. 怎么去维护用户的登录状态？session
7. Session有哪几种实现方式？
8. Session有一个问题就是无法共享到其他服务器，！！！这个好像很重要
9. 如果浏览器的cookie删掉，session还能用吗

## Java集合

1. List，Set，Map三者区别
2. List 顺序的好帮手：存储一组不唯一的有序的对象
3. Set 注重独一无二的性质：不允许重复的集合
4. Map 用key来搜索：使用键值对存储。两个key可以引用相同的对象，但key不能重复。Key一般是String，也可以是任何对象（自定义对象需要重写hashCode和equals方法）
5. Arraylist与LinkedList
6. 线程安全：都是线程不安全的
7. 数据结构：ArrayList使用Object数组，LinkedList使用双向链表（1.6之前为循环链表，1.7取消了循环）
8. 插入删除：二者对于指定位置的插入删除都要耗费O（n），因为前者需要对元素进行移动操作，而后者需要先移动到指定位置再插入。
9. 快速随机访问：LinkedList不支持，而ArrayList支持。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（int index）方法）
10. 内存占用：ArrayList空间主要浪费在list结尾会预留一定的空间，而LinkedList的空间花费在它每个元素都需要消耗额外的空间存储直接前驱和直接后继。
11. ArrayList与Vector区别，为什么用前者取代后者？

Vector类的所有方法都是**Synchronized同步**的，线程安全的，但是单线程访问Vector的话就要在同步操作上耗费大量时间

ArrayList不是同步的，所以不需要保证线程安全时建议使用ArrayList。

1. **ArrayList的扩容机制？？？**<https://www.cnblogs.com/SunArmy/p/9844022.html>

**如果是无参构造，就初始化一个长度为0的空Object数组，调用add方法时，先size+1，然后将元素加入数组的size位置处，如果添加时数组为长度0的空数组，就初始化长度为10的数组。如果需要的长度大于原来的长度时就需要扩容，扩容每次变为原来容量的1.5倍，把老数组copy到新数组中，再进行添加元素操作。**

1. **无参构造，Object[0]**
2. **Add, size+1, 加入元素**
3. **数组为空，初始化为长度10**
4. **长度不够，扩容为1.5**
5. **HashMap和HashTable**
6. **线程是否安全**：HashMap是非线程安全的，HashTable线程安全

效率：因为线程安全的问题，HashMap效率高，另外HashTable基本被淘汰，不建议使用。

1. 对**null**的支持：HashMap中null可以做键或值。但HashTable不允许。

补充：为什么HashTable和ConcurrentHashMap不可以是null？

答：Map不是线程安全的，一般用于单线程环境，而Table用于多线程环境。

map.get(key)返回null，无法判断key对应的值是null，还是根本不存在key。单线程可以用map.contains(key)来判断，而多线程解决这个问题很复杂，必须保证contains与get操作的原子性。因此线程安全的map不允许value为null（也是这个原因，HashTable没有contains方法，错了，HashTable有containsValue和containsKey方法）

为什么key不能是null呢？

答：null不是对象，无法调用equals和hashCode方法，因此无法计算哈希值用以做键。但HashMap对此做了特殊处理。

1. **初始容量**和扩充容量的大小：
2. 不指定初始值：HashTable默认初始大小为11，每次扩充为原来的2\*n+1；HashMap默认初始大小为16，每次扩充容量变为原来的2倍
3. 创建时指定初始值：HashTable会直接使用指定大小，而HashMap会扩充为2的幂次方大小（向下取整）（后面介绍原因）
4. **底层数据结构**：1.8之后HashMap在解决哈希冲突有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转为红黑树，以减少搜索时间。HashTable没有这种机制。
5. HashMap和HashSet的区别

后者的底层就是基于前者实现的（key就是存入的对象，value是一个Object类型的对象，无意义），HashSet自己实现的方法有clone(), writeObject(), readObject(),其他都是调用HashMap中的方法。



1. HashSet如何检查重复（实际上就是HashMap的插入过程）

把对象加入Set时，HashSet会先计算对象的hashcode值来进行下标运算（也就是hash&（n-1）），判断对象加入的位置是否为空，如果为空就直接插入，如果不为空，就与已存在对象的hashcode比较同时进行equals方法比较（&&），如果相同就插入失败。否则，再对已存在对象进行节点判断（是树节点还是普通节点），如果是普通节点，就从头遍历并equals比较，若与要插入的对象与某个已存在对象相同，就插入失败，否则在尾部插入新节点。如果是树节点，它的插入方法没有细看。

Hashcode与equals的相关规定：

1. 两个对象内容相同则hashcode一定相同，equals返回true
2. 两个具有相同hashcode的对象不一定内容相同
3. 将对象放入不可重复的集合（Map的key，或Set）时，需要判断对象的hashcode，进一步判断对象的equals，所以重写equals就必须重写hashcode。

关于hashCode和equals的重写

为什么需要重写？

答：把自定义对象放入Map中做key时就需要重写这两个方法。

当把自定义对象放入某些集合中时，比如让自定义对象作为Map的key。给定一个对象，Map会调用对象的hashCode方法获得该对象的哈希值，放入hash值所指引的内存位置（这里用指引二字，因为hash值并不完全等同于内存地址）。我们希望内容相同的两个对象拥有相同的哈希值，然而如果不重写hashCode方法，就会使用从Object类继承的hashCode方法，得到的是该对象的内存地址（也就是对应的引用类型变量的值），这样就导致即使内容相同的对象也会有不同的hashCode，这是我们不希望的。

为什么重写equals方法？

即使是不同内容的对象也有可能具有相同的hash值，因此HashMap不能只靠hashCode就取出value。当查询k1对应的value时，首先要通过计算k1的hash值得到HashMap中key的存储地址，然后再进一步比较该地址中放入的k0是否equals（k1），所以即使k0和k1的hashCode相同也是不够的。如果不重写自定义对象的equals，那么默认使用的是继承自Object类的equals方法，判断依据还是两个对象的内存地址，即使内容相同的两个对象也是不equals的，因此需要重写，只要两个对象的内容相同就返回true。

1. ==和equals的区别

1） ==可以用于基本类型和引用类型的比较，进行的是值比较（也就是说当比较引用类型时，实际比较是否指向同一个对象）

2）equals只能比较引用类型，一般情况下也是进行值比较（即比较是否指向同一对象，或者说比较引用类型变量存放的内存地址）

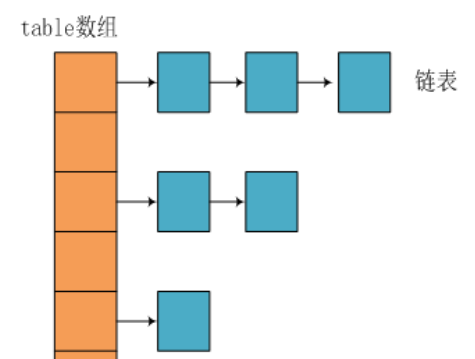
3） 但是File String Date 以及包装类，equals比较的是类型和内容（**原因在于这些类重写了equals方法**）

4）需要注意的是，使用字面量创建对象时（就是不使用new关键字，比如String a=“a”; int b=2;）,这些对象只在常量池中保存一份，所以两个字面量创建的对象只要内容相同就是指向同一对象，但是如果是用new创建对象，new一个就在堆内存中保存一个对象，不管内容是否相同,同时也会在常量池保存一份。（比如 Integer i = new Interger(1); Integer j = new Interger(1); i和j分别指向堆内存中的创建的两个Integer对象，但常量池中也会保存一个1）

1. **HashMap的底层实现**

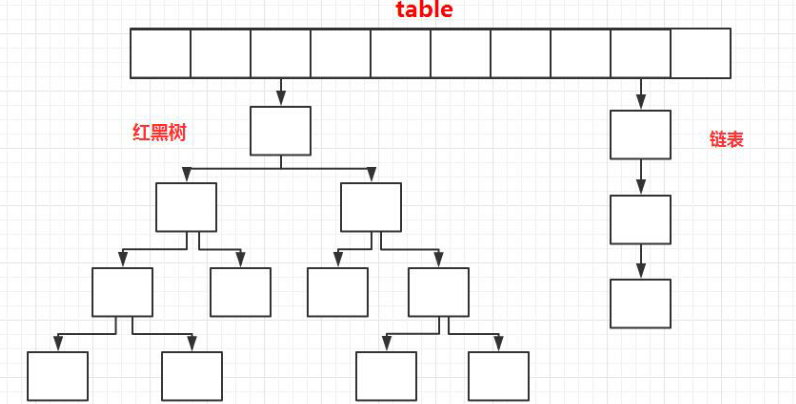
1.8之前

数组+链表（链表散列）。HashMap经过key的hashcode经过扰动函数处理得到hash值，然后通过（n-1）&hash判断当前元素的存放位置（n是数组长度）。如果当前位置已经存放元素，就判断该元素与要存入元素的hash值以及key是否相同，相同则覆盖，不同则拉链法解决。（拉链法就是数组加链表，数组每一格都是一个链表，哈希冲突值放到对应位置的链表中）



1.8之后（包括1.8）

链表长度大于8时转化为红黑树以减少搜索时间。TreeMap、TreeSet以及1.8之后的HashMap都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为某些情况下查找树会退化为线性结构。



1. HashMap的长度为什么是2的幂次方

Hash值的范围是int型的，-2147483648到2147483647，前后加起来大概40亿的映射空间。但一个40亿长度的数组内存是放不下的，（也就是**散列值范围很大而数组不可能这么大，因此无法直接拿来使用**）所以这个散列值不能直接使用，在**使用之前要先对数组的长度做取模运算，得到的余数才能用来代表存放位置，也就是数组下标**。这个数组下标的计算方法是（n-1）&hash，n为数组长度，这也就解释了为什么HashMap长度为2的幂次方。。当然，计算下标的（n-1）&hash中，hash并不是key的hashCode（）直接拿来使用，而是经过扰动函数（高16位与低16位相与）。

**取余（%）操作如果除数是2的幂次则等价于与其除数减一的与（&）操作**（也就是说，如果length是2的n次方，那么hash % length == hash&（length - 1）），**采用二进制与运算比取余效率高，这就解释了为什么HashMap的长度为2的幂次方。**

1. HashMap多线程操作导致死循环问题

主要原因在于并发下rehash会造成元素之间形成一个循环链表，不过1.8之后解决了这个问题，但还是不建议多线程下使用HashMap，因为还是可能会存在数据丢失问题。并发环境建议使用ConcurrentHashMap。

详见：<https://coolshell.cn/articles/9606.html>

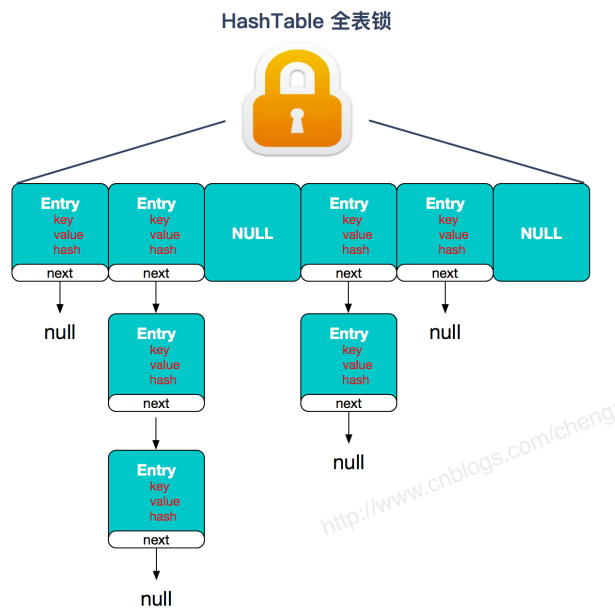
1. **ConcurrentHashMap和HashTable的区别**

主要区别在实现线程安全的方式上不同

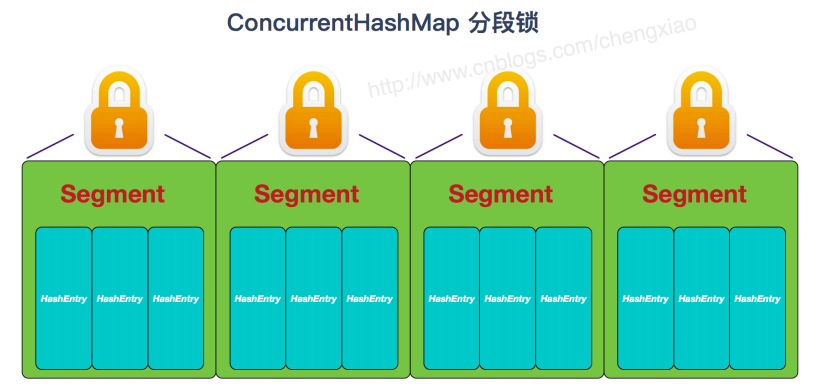
1. 底层数据结构：
   1. 1.7的ConcurrentHashMap采用分段数组+链表，1.8采用数组+链表/红黑树。
   2. HashMap1.8之前用数组+链表，1.8采用数组+链表/红黑树
   3. HashTable用数组+链表
2. 实现线程安全的方式（重要）
   1. ConcurrentHashMap，1.7时采用分段锁，对整个数组进行了分割分段（segment），每一把锁只锁容器中的一部分数据，提高了并发访问效率。1.8摒弃了Segment的概念，直接用Node数组+链表/红黑树的数据结构来实现，并发控制用synchronized和CAS来操作（1.6后对synchronized锁做了很多优化），看起来像是优化过且线程安全的HashMap。
   2. HashTable：使用synchronized(给所有方法（get，put，contains）都加上了Synchronized关键字)来保证线程安全，效率低下。一个线程访问同步方法时，没有获得锁就可能进入阻塞或轮询状态

对比图：

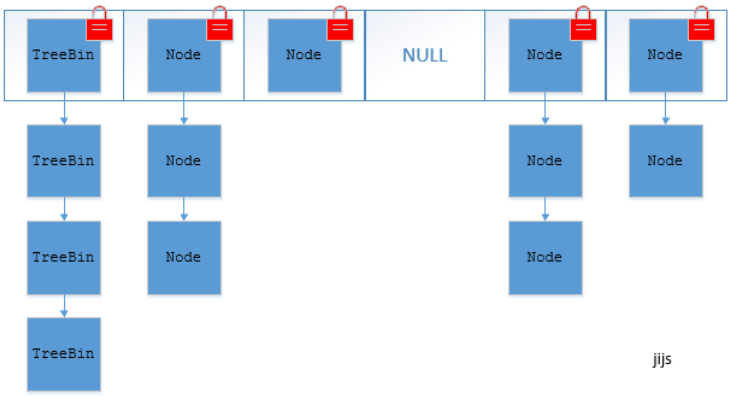
HashTable



1.7 的ConcurrentHashMap



1.8 的ConcurrentHashMap（TreeBin 红黑树节点，Node：链表节点）

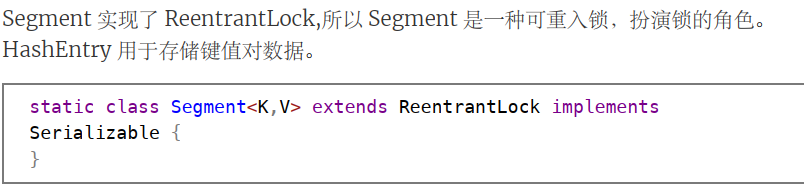


1. **ConcurrentHashMap线程安全的底层具体实现**

1.7

将数据分段存储，每一段数据分配一把锁，当一个线程访问其中一个数据段时，其他段的数据可以被其他线程访问。

**ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。**



一个ConcurrentHashMap包含一个Segment数组。数组中每个Segment的结构和HashMap类似，是数组+链表的结构。一个Segment包含一个HashEntry的数组，每个HashEntry是一个链表结构的元素，每个Segment守护着一个HashEntry数组中所有元素。对HashEntry数组中的数据修改时要先获得对应的Segment的锁。

1.8

取消了分段锁，采用CAS和synchronized保证并发安全。数据结构跟HashMap1.8类似（数组+链表/红黑树）。1.8中当链表达到长度阈值（默认8）时转为红黑树。

Synchronized只锁定当前链表或红黑树节点的首节点，只要hash不冲突，就不会产生并发，效率提升。

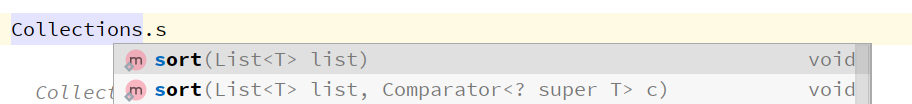
CAS（Compare and Swap）操作包含三个操作数（内存当前值，预期的原值，新值）。如果内存的值和预期的原值是一致，那就转化为新值。CAS是通过不断的循环来获取新值的，如果线程中的值被另一个线程修改了，那么就需要自旋，到下次循环才有可能执行。

Node保存的是key，value，next指针和key的hash值。其中value和next都用volatile修饰，保证并发的可见性。

1. Compareble和Comparator区别
2. Comparable接口出自java.lang包，有一个compareTo（Object obj）方法用来排序
3. Comparator接口出自java.util包，有一个compare（Object obj1，Object obj2）方法用来排序。
4. 这两个接口用于自定义类的实例对象的比较。

使用方法：

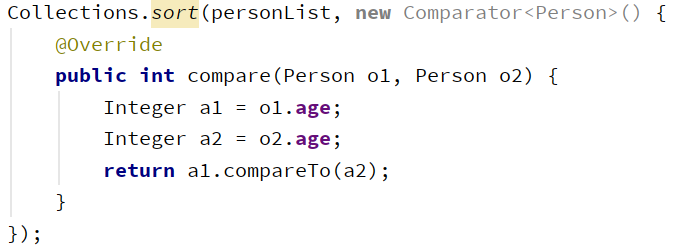
1. 对List集合排序时，可以使用该方法（该方法只能排序List集合或数组）



1. 如果在自定义类实现了Comparable接口，并重写了compareTo方法，如下，就可以使用上图的第一个函数（也就是sort函数只传待排序的List集合即可）；



如果在进行排序时才确定以何种方式对对象排序，那么可以使用第二种sort方法（具有两个参数，第二个参数为new Comparator<待排序对象的类>）（使用如下图）



1. **compareTo方法与compare方法的返回参数：（太绕了）**

有点绕：我们可以认为sort方法一直是按升序排序的，只是我们重写了以上两个方法来重新规定两个需要比较的对象之间的“大小”关系。如果返回1，说明第一个对象比第二个对象“大”，如果返回-1，说明第一个对象比第二个对象“小”，二者“大小”相同则返回0.

1. String类和包装类已经实现了Comparable接口重写了compareTo方法（默认升序），所以List集合里面如果是这两种类型的对象就不用再重写了。倒序可以用Collections.reverse()方法。
2. **ConcurrentHashMap为什么读不加锁？**

**在HashTable中get方法都是使用Synchronized加锁的。而ConcurrentHashMap中get方法不加锁，这是因为：**

**Get源码：计算hash值定位索引位置，如果首节点符合则返回；遇到扩容的时候，会调用标志正在扩容节点的ForwardingNode的find方法，查找该节点，匹配则返回；以上不符合的话就遍历节点看是否匹配，匹配则返回，否则返回null。**

**Volatile可见性原理：使用volatile会强制将修改的值立即写入主存；使用volatile关键字，当线程2修改时，会使线程1中的工作内存中的缓存无效，线程1读取时需要去主存读取。（volatile修饰基本数据类型时保证值可见，修饰引用类型变量时，保证引用对象的地址可见，而不保证对象内容可见）**

**Get操作无锁是因为Node元素的value和next指针都是volatile修饰的，多线程环境下修改节点的value或新增节点，其他线程都是可见的。另外Node数组也用volatile修饰，这是为了数组扩容时保持可见性。**

1. **Volatile vs Synchronized**

* **前者修饰变量，后者修饰方法，代码块**
* **前者的实现变量的修改可见性，不保证原子性；后者可以保证可见性和原子性**
* **前者不造成线程阻塞，后者可能会造成线程阻塞**
* **前者的本质是告诉jvm，变量在主内存中的不确定性，而后者是锁定当前变量，只有当前线程可以访问，其他线程阻塞。**

1. **一段模拟面试：**

**参考：**[**https://open-hl.toutiao.com/a6804793505479655939/?utm\_campaign=open&utm\_medium=webview&utm\_source=o\_llq\_api&req\_id=202003170819060100140360750467CA40&dt=OPPO+R9sk&label=open\_o\_channel&a\_t=4115924741504904790338914b4&gy=e3b466f9cbf4d75c46ceb660d1615cdab3d7bc19a180f3f4864f04805c7cd621728ef4290a5526bdbafa96e3d0df445db44d92f12359226bbb9114f315aa3a657b5f258f917359f2b7fc79e159f6e92db4d081be41aa6a3f93901f76db5bbb3a11ee2ff9b174de3bf21c1564ebcfa905ce6984e93eec344488c32cfbf4e982f66d0e2873b4c18a57afbd2c559a3eb7e31447f92ee3eb083bb7b5c8d1d11079cfc577b0e509d0db8691aa92f0e8e1cc316c4a9c0f16d1ba387853e13da19d7987&crypt=614&item\_id=6804793505479655939&\_\_docId\_\_=6804793505479655939&\_\_barStyle\_\_=1\_1&\_\_fromId\_\_=\_\_all\_\_&\_\_statParams\_\_=sourceMedia&\_\_source\_\_=toutiao&sourceMedia=toutiao&\_\_cmtCnt\_\_=21&\_\_styleType\_\_=2&\_\_publisher\_id\_\_=81362283758&openv=&\_\_pf\_\_=detail**](https://open-hl.toutiao.com/a6804793505479655939/?utm_campaign=open&utm_medium=webview&utm_source=o_llq_api&req_id=202003170819060100140360750467CA40&dt=OPPO+R9sk&label=open_o_channel&a_t=4115924741504904790338914b4&gy=e3b466f9cbf4d75c46ceb660d1615cdab3d7bc19a180f3f4864f04805c7cd621728ef4290a5526bdbafa96e3d0df445db44d92f12359226bbb9114f315aa3a657b5f258f917359f2b7fc79e159f6e92db4d081be41aa6a3f93901f76db5bbb3a11ee2ff9b174de3bf21c1564ebcfa905ce6984e93eec344488c32cfbf4e982f66d0e2873b4c18a57afbd2c559a3eb7e31447f92ee3eb083bb7b5c8d1d11079cfc577b0e509d0db8691aa92f0e8e1cc316c4a9c0f16d1ba387853e13da19d7987&crypt=614&item_id=6804793505479655939&__docId__=6804793505479655939&__barStyle__=1_1&__fromId__=__all__&__statParams__=sourceMedia&__source__=toutiao&sourceMedia=toutiao&__cmtCnt__=21&__styleType__=2&__publisher_id__=81362283758&openv=&__pf__=detail)

1. HashMap的数据结构？

1.8使用数组+链表/红黑树

1. HashMap的数据插入原理

判断数组是否为空，为空则进行初始化为16大小：

不为空则计算k的hash值（拿到k的hashCode（）再进行扰动函数），通过（n-1）&hash计算应存放的数组下标index；查看table[index]是否存在数据，不存在就构造一个Node节点存放在该位置，如果存在数据说明发生了hash冲突，（存在两个节点的key的hash值相同），继续判断key是否相等（使用key的equals方法），相等则替换，不相等则进行插入操作，插入时先判断当前节点是不是树形节点，如果是树形节点就创建树形节点插入，否则创建普通Node加入链表（需要遍历判断是否equals链表中某个已存在的key，没有则插入链表尾部），判断链表长度是否大于8，大于则转换为红黑树。插入完成后判断当前节点数是否大于阈值，如果大于就扩容为原数组的二倍。

1. 刚才提到HashMap的初始化，那它是怎么设定初始容量大小的？

如果 new HashMap<>（）不传值，默认大小是16，负载因子是0.75，如果传入初始大小k，则初始化大小为大于k的2的整数次方。（HashMap当前size大于等于负载因子乘总容量时进行扩容，扩容时进行rehash来确定各元素在新数组中的位置）

1. HashMap的hash函数是怎么设计的？

先拿到key的hashCode，（通过调用key的hashCode方法），得到32位的int值，然后让hashcode的高16位与低16位进行异或操作。

**这个顺序到底是怎么样的？拿到key的hashcode，再经过扰动函数得到新的hash，再（n-1）&hash得到下标值，是这样吗？是的！**

1. 为什么这样设计呢？

这个也叫扰动函数，这么设计原因有二：

一定要尽可能减少hash碰撞，越分散越好；算法一定要尽可能高效，因为这是高频操作，所以采用位运算。

1. 为什么采用高16位与低16位异或能降低hash碰撞？hash能不能直接使用key的hashcode？

* 因为key.hashCode()函数调用的是key自身类型自带的哈希函数，返回int型散列值，范围很大（内存根本放不下40亿长度的数组），而HashMap数组的大小没有这么大，所以hashcode不能直接使用，使用之前需要对数组长度做取模运算，得到的余数才能用来访问数组下标。如果Map大小n是2的整数次幂，那么hash%n就等价于（n-1）&hash，而后者使用位运算效率更高，所以Map的容量为2的整数次幂。
* 直接取余也会有问题：就算散列值（key的hashcode）再分散，要是只取最后几位的话碰撞也会很严重。如果散列做的不好，分布上成等差数列的漏洞，如果正好让最后几个低位呈现规律性重复，就很容易碰撞。

这时扰动函数的价值就体现出来了。高位与低位异或就是为了**混合原始哈希码的高位和低位，以此来加大低位的随机性**，而且混合后的低位也掺杂了高位的部分特征，这样高位信息也变相的保留下来。（另外1.7做了4次移位和异或，1.8只做了一次，可能是考虑到效率和实际效果）

1. **1.8还有其他优化吗**
2. 数组➕链表改成了数组➕链表/红黑树。
3. 链表的插入方式从头插法改成了尾插法。
4. 扩容的时候1.7需要对原数组中的元素进行重新hash定位在在新数组中的位置，而1.8采用更简单的判断逻辑，位置不变，或变为原位置+旧容量大小。
5. 在插入时，1.7先判断是否需要扩容，1.8是先插入再判断是否需要扩容。
6. 为什么做这些优化？
7. 数据结构改变是为了减少hash冲突时的时间复杂度，由n降为logn
8. 头插法扩容时会使链表发生反转，多线程环境下会产生环：为什么？<https://blog.csdn.net/dgutliangxuan/article/details/78779448>
9. 扩容时为什么1.8不用重新rehash就可以直接定位原节点在新数组中的位置？这是因为扩容是扩大为原数组大小的2倍，回想hash函数计算方式：（n-1）&hash，（n为2的整数次幂）如果n扩大为2倍，那么n-1的二进制仅仅是由0001111变为0011111，也就是1多了一位，与hash进行&运算时，（hash是key自身计算得到的hashcode，rehash的过程中不会变化），得到的结果要么是原结果，要么是原结果+原数组大小（取决于hash值对应位上是1是0）
10. HashMap是线程安全的吗？怎么解决

不是。Java中有HashTable，Collections.synchronizedMap方法以及ConcurrentHashMap可以实现线程安全的Map。

HashTable是直接在操作方法上加synchronized关键字，锁住整个数组，粒度大，Collections.synchronizedMap是使用Collections集合工具的内部类，通过传入Map封装出一个SynchronizedMap对象，内部定义了一个对象锁，方法内通过对象锁实现；ConcurrentHashMap使用分段锁，降低了锁粒度，使并发度大大提高。

1. 那你知道ConcurrentHashMap的分段锁的实现原理吗？

ConcurrentHashMap的成员变量用volatile修饰，免除了指令重排序，同时保证内存可见性，另外使用CAS操作和synchronized结合实现赋值操作，多线程操作只会锁住当前操作索引的节点。

1. 你前面提到链表转红黑树是链表长度达到阈值，这个阈值是多少？

8，红黑树转链表阈值是6

1. 为什么？

在hash函数设计合理的情况下，发生hash碰撞8次（也就是有8个hash相同的情况）的概率为百万分之6，所以8够用了。如果hash碰撞次数在8附近徘徊，会一直发生链表和红黑树的转化。为了避免这种反复转化的情况所以二者的阈值不同。

1. HashMap是有序的吗

是无序的，会根据hash值随机插入。

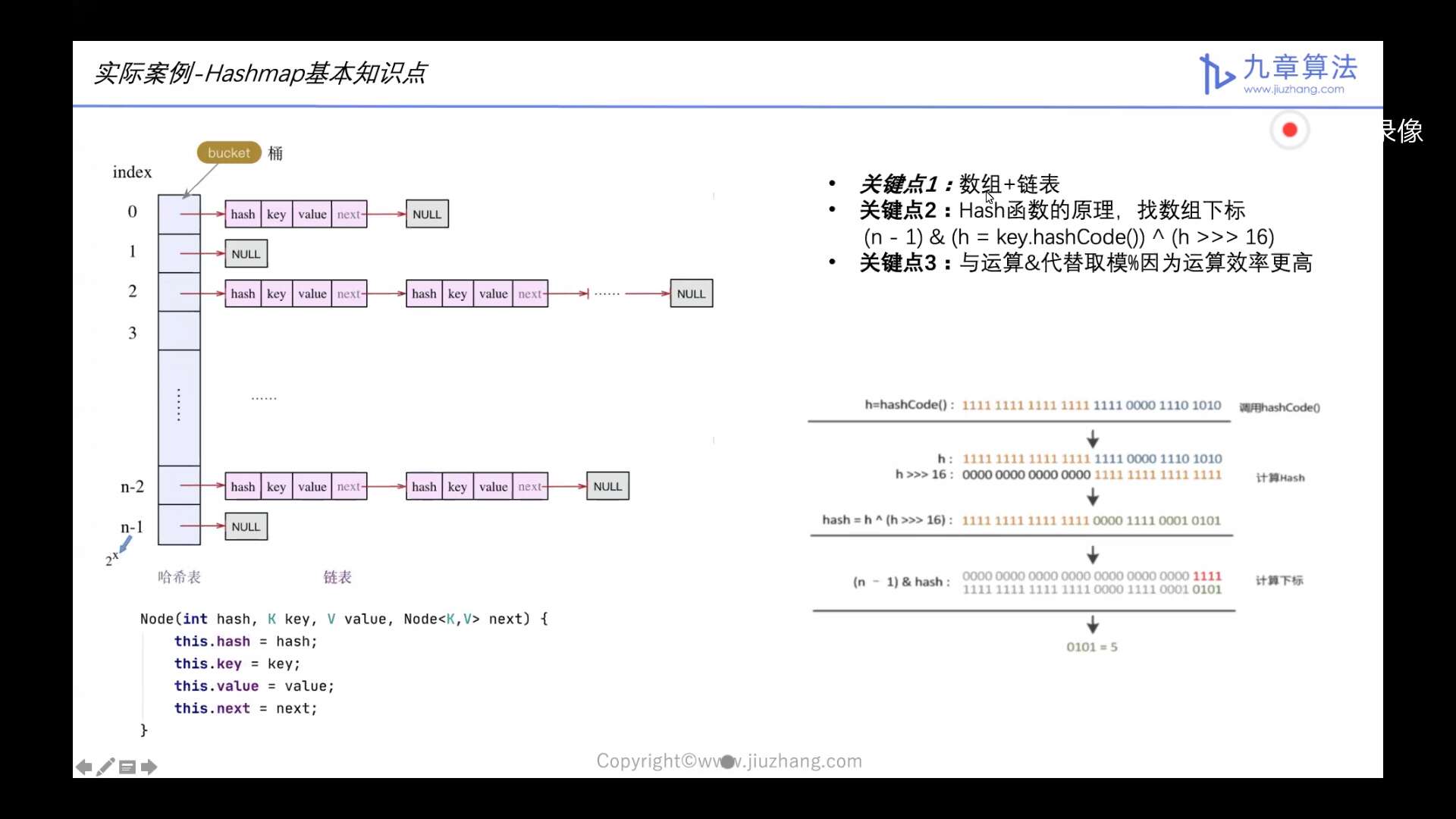
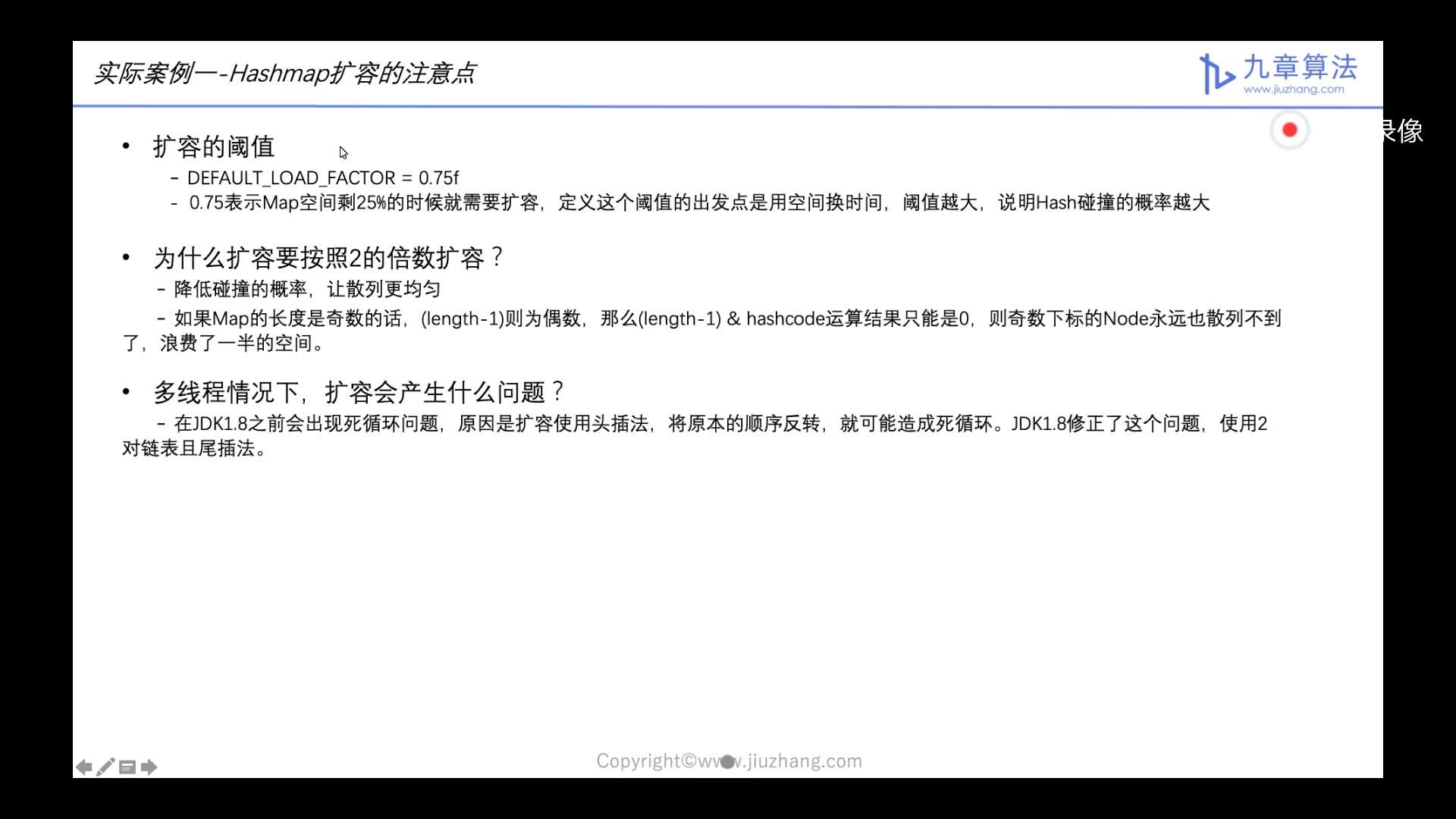
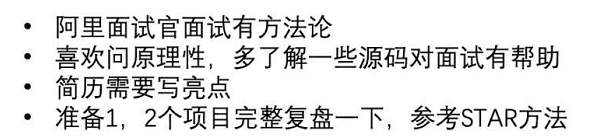
1. 有没有有序的Map？

LinkedHashMap和TreeMap是有序的。

1. 怎么实现有序的呢

前者（继承了HashMap）内部维护了一个双向链表，定义了头尾结点，节点名为Entry，继承了HashMap的Node类，Entry维护了before和After保存前置节点和后置节点。

后者是按照key的自然顺序或Comparator的顺序排序，内部是通过红黑树实现。所以要么key所属的类实现了Comparable接口（重写了compareTo方法），要么自定义一个实现了Comparator接口的比较器传给TreeMap用于key的比较。

1. HashMap的问题：
2. 
3. 扩容的问题
4. 
5. 追问系列：
6. 如何解决HashMap线程不安全的问题：1，使用Collections.synchronizedXxx转换，2.使用ConcurrentHashMap，3. 使用Synchronized同步代码块，锁住HashMap对象。
7. ConcurrentHashMap如何做到线程安全
8. ConcurrentHashMap在1.8下解决ABA的问题
9. 为什么引入红黑树而不是平衡二叉树？红黑树的原理？为什么阈值是8？
10. 项目：STAR：什么场景下？什么任务（达到什么效果）？如何做的（方案有几个，为什么选择这个）？结果和收获，改进之处，遇到的问题，如何解决？（展示，说出你的思考，所以需要复盘）
11. 
12. 阿里近一年上机做题是必须的，伯乐在线（给候选人四选一去做，考察规范，思路，复杂度）