Java面试-集合：

java 有哪些集合?

- 实现 Collection 接口和 Map 接口的
- Collection 子接口 -> List 接口 和 Set 接口
- List： ArrayList 、 LinkedList 、 Vector (线程安全)
- Set： HashSet 、 LinkedHashSet 、 TreeSet
- Map： HashMap 、 LinkedHashMap 、 HashTable (线程安全)

List 与 Set 、 Map 区别?

- List 和 Set 是存储单列数据的集合。 Map 是存储键值对的。
- List 有序，值允许重复
  - ArrayList 基于数组实现，**增删慢，查找快**;
    - 优点：是随机读取效率很高，原因数组是连续（随机访问性强，查找速度快）

        - 缺点：插入和删除数据效率低，因插入数据，这个位置后面的数据在内存中要往后移的，且大小固定不易动态扩展
    - `LinkedList` 基于链表实现，链表内存是散列的，**增删快，查找慢**；
        - 缺点：不能随机查找，每次都是从第一个开始遍历（查询效率低）。
- `Set` 无序， 值不允许重复
- `Map` 中存储的数据是无序的，它的键是不允许重复的，但是值是允许重复的；
    - `TreeMap` ，能够把它保存的记录根据键排序，默认是**键的升序排序**

数组

- 数组通过角标进行随机访问的时间复杂度为 `O(1)`
    - 为什么数组能支持随机访问呢?
        - **数组占用的内存空间是连续的**
        - 数组中都为**同一类型的元素**（内存地址与基址之间的偏移字节，每次都偏移相同的字节）
    - 为什么数组增加删除慢呢？（增加同理）
        - 数组是一个有序列表，数组是在连续的位置上面储存对象的应用。当我们删除某一个元素的时候在他后面的元素的索引都会左移

列表

- 在内存中，元素的空间可以在任意地方，空间是分散的，不需要连续。 任意位置插入元素和删除元素的速度快，时间复杂度是 `o(1)`
- 链表中的元素有两个属性，一个是元素的值，另一个是指针，此指针标记了下一个元素的地址。每一个数据都会保存下一个数据的内存地址，通过该地址就可以找到下一个数据
- 查找数据时间效率低，时间复杂度是 `o(n)`： 因为链表的空间是分散的，所以**不具有随机访问性**，如果需要访问某个位置的数据，需要从第一个数开始找起，依次往后遍历，直到找到待查询的位置，故可能在查找某个元素时，时间复杂度是 `o(n)`

`ArrayList` 扩容原理

1. `ArrayList` 初始长度为0（这里以jdk17为例）
2. 通过 `add()` 方法添加单个元素时，会先检查容量，看是否需要扩容。如果容量不足需要扩容则调用 `grow()` 扩容方法
    - 当第一次调用 `add` 后，长度变为10 (初始长度为0的话)

    - 判断如果为空数组，则两者取最大值默认容量10（DEFAULT_CAPACITY =10, minCapacity=size+1 =1）
    - minCapacity 代表最小的新容量
- newCapacity = oldCapacity + max(minCapacity-oldCapacity, oldCapacity>>1)，即**至少1.5倍**
    - minGrowth=minCapacity-oldCapacity，代表最小的扩容范围
- 如果新数组长度大于 Integer.MAX_VALUE - 8， 就使用 minGrowth 或 Integer.MAX_VALUE - 8
    a. 如果 minGrowth 的长度溢出了就报错（特殊的JVM接近 Integer.MAX_VALUE 时即便有足够内存也会报错）
    b. 如果 minGrowth 小于 Integer.MAX_VALUE - 8 就返回 Integer.MAX_VALUE - 8
    c. 否则使用 minGrowth： 此时新数组容量大于 Integer.MAX_VALUE - 8 却没有内存溢出，范围在 Integer.MAX_VALUE - 8 和 Integer.MAX_VALUE 之间

3. 通过 Arrays.copyOf 方法把原数组的内容放到更大容量的数组里面

```
private Object[] grow() {
    return grow(size + 1);
}

private Object[] grow(int minCapacity) {
    int oldCapacity = elementData.length;
    if (oldCapacity > 0 || elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        int newCapacity = ArraysSupport.newLength(oldCapacity,
                minCapacity - oldCapacity, /* minimum growth */
                oldCapacity >> 1           /* preferred growth */);
        return elementData = Arrays.copyOf(elementData, newCapacity);
    } else {
        return elementData = new Object[Math.max(DEFAULT_CAPACITY, minCapacity)];
    }
}

// 限制可能取决于某些 JVM （例如 HotSpot）实现特定的特征，例如对象标头大小。所以选择保守的大小。
public static final int SOFT_MAX_ARRAY_LENGTH = Integer.MAX_VALUE - 8;

public static int newLength(int oldLength, int minGrowth, int prefGrowth) {
    // preconditions not checked because of inlining
    // assert oldLength >= 0
    // assert minGrowth > 0

    int prefLength = oldLength + Math.max(minGrowth, prefGrowth); // might overflow
    if (0 < prefLength && prefLength <= SOFT_MAX_ARRAY_LENGTH) {
        return prefLength;
    } else {
        // put code cold in a separate method
        return hugeLength(oldLength, minGrowth);
    }
}

private static int hugeLength(int oldLength, int minGrowth) {
    int minLength = oldLength + minGrowth;
    if (minLength < 0) { // overflow
        throw new OutOfMemoryError(
            "Required array length " + oldLength + " + " + minGrowth + " is too large");
    } else if (minLength <= SOFT_MAX_ARRAY_LENGTH) {
        return SOFT_MAX_ARRAY_LENGTH;
    } else {
        return minLength;
```

```
        }
    }
```

LinkedList

LinkedList 底层的实现基于**双向表**。每个 Node 实例除了保存节点的真实值(真实数据)外，还保存了这个节点的前一个节点的引用和后一个节点的引用。底层是 Node 节点 next 指向下一个 node 的地址。 Prev 指向上一个 node 。

根据索引查找 node(int index)

所有根据索引的查找操作都是按照双向链表的需要执行的，根据索引从前或从后开始搜索，并且**从最靠近索引的一端开始**。这样做的目的可以提升查找效率。那如何做到这一点呢？在 LinkedList 内部有一个 node(int index) 方法，它会**判断从头或者从后开始查找比较快**。

通过比较 index 和容量右移一位（相当于除以2）的大小

```
    Node<E> node(int index) {
        // assert isElementIndex(index);

        if (index < (size >> 1)) {
            Node<E> x = first;
            for (int i = 0; i < index; i++)
                x = x.next;
            return x;
        } else {
            Node<E> x = last;
            for (int i = size - 1; i > index; i--)
                x = x.prev;
            return x;
        }
    }
```

添加 add(E e)

add(E e) 方法实际上调用的是 linkLast(E e) 方法，意思是把元素**加到链表的最后**。

1. first 和 last ，分别保存了当前链表**第一个节点和最后一个节点**的引用。
2. 在新增一个节点之前，首先把指向最后一个节点的引用（即变量 last ）保存起来（即变量 l）

3. 然后新建一个节点，把**新节点的前驱设为链表最后一个节点**，把**指向最后一个节点的引用（`last`）指向新建的节点。**
4. 然后把**链表最后一个节点的后继节点设为新建的节点**
5. 最后把整个链表的总数加1，完成了新增一个节点的操作。

图示：

1. `a ⇆ b(last)`
2. `a ⇆ b(last, l) ← c`
3. `a ⇆ b(l) ← c(last)`
4. `a ⇆ b(l) ⇆ c(last)`
5. `size++;`
6. `modCount++;` 并发控制

```
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
public boolean add(E e) {
    linkLast(e);
    return true;
}
```

`ArrayList` 和 `LinkedList` 中的 `transient` 关键字和序列化

`transient` 关键字的作用，简单地说，就是让某些被修饰的成员属性变量**不被序列化。**

- `arratlsit` 中的 `elementData` 设置为了 `transient` 。
  - `elementData` **不总是满的**，每次都序列化，**会浪费时间和空间**
  - 重写了 `writeObject` 保证序列化的时候虽然不序列化全部，但是**有的元素都序列化。**不是不序列化而是不全部序列化
- `LinkedList` 中将 `first` 和 `last` 修饰成 `transient` 是为了节省空间和重新连接链表。

HashMap

HashMap 为什么使用红黑树，而不是其他的树？

- 为什么不使用二叉排序树？原因：二叉排序树在添加元素的时候**极端情况下会出现线性结构。**
  - 由于二叉排序树左子树所有节点的值均小于根节点的特点，如果我们添加的元素都比根节点小，会导致左子树线性增长，这样就失去了用树型结构替换链表的初衷，导致查询时间增长。
- 为什么不使用平衡二叉树呢？红黑树**不追求"完全平衡"。**它能**保证在最坏的情况**下，基本动态集合操作时间为 O(logn) .是功能、性能、空间开销的**折中结果。**

HashMap 的实现原理

map.put(k,v) 实现原理：

1. 首先将 k,v 封装到 Node 对象当中（节点）
2. 调用 K 的 hashCode() 方法得出 hash 值，将 hash 值转换成**数组的下标**
3. 下标位置上如果没有任何元素，就把 Node 添加到这个位置上
4. 下标对应的位置上如果有链表。此时，就会拿着 k 和链表上每个节点的 k 进行 equal
   - 如果所有的 equals 方法返回都是 false ， 那么这个新的节点将被添加到**链表的末尾。**
   - 如其中有一个 equals 返回了 true ， 那么这个节点的 value 将会被**覆盖。**

map.get(k) 实现原理：

1. 调用 k 的 hashCode() 方法得出哈希值，并通过哈希算法转换成**数组的下标。**
2. 通过数组下标快速定位到某个位置上
   - 如果这个位置上**什么都没有，则返回 null 。**
   - 如果这个位置上有**单向链表**，那么它就会拿着参数 K 和单向链表上的每一个节点的 K 进行 equals
     - 如果都返回 false ， 则 get 方法返回 null 。
     - 如果其中一个节点的 K 和参数 K 进行 equals 返回 true ， 那么 get 方法返回这个要找的 value

HashMap 的扩容原理

hashmap 集合的默认初始化容量为 16 ， 默认加载因子为 0.75 ， 也就是说这个默认加载因子是当 hashMap 集合底层数组的容量达到 75% 时，数组就开始扩容。为了达到**散列均匀**，提高 hashmap 集合的存取效率, hashmap 集合初始化容量是 2 的倍数。

JDK8 之后，如果哈希表单向链表中元素超过 8 个，那么单向链表这种数据结构会变成**红黑树**数据结构。当红黑树上的节点数量小于 6 个，会重新把红黑树变成**单向链表**数据结构。红黑树的阈值为什么是 8？

- 首先和 hashcode 碰撞次数的泊松分布有关，主要是为了寻找一种**时间和空间的平衡**。在负载因子 0.75 （ HashMap 默认）的情况下，**单个 hash 槽内元素个数为 8 的概率小于百万分之一。**
- 红黑树转链表的阈值为 6 ，主要是因为，如果也将该阈值设置于 8 ，那么当 hash 碰撞在 8 时，会发生**链表和红黑树的不停相互转换**，白白浪费资源。

HashMap 扩容大小为什么是 2 的幂?

- 2的幂可以保证数组的长度是一个偶数，这样可以避免一些奇偶性的问题。
- 2的幂可以让 HashMap 的散列函数更简单高效，只需要用 hash 值和数组长度减一做按**位与运算**，就可以得到元素在数组中的位置。这样可以减少乘法和取模的开销，提高性能。
- 2的幂可以让 HashMap 的**扩容更方便**，只需要将数组长度乘以2，就可以得到新的容量。这样可以保证**元素在扩容后的位置要么不变，要么在原来的基础上加上数组的旧长度**。这样可以**减少元素的重新散列**，提高效率。

Hash 计算:

- 使高 16 位也参与到 hash 的运算能减少冲突
- 索引： `(n - 1) & hash`
  - n 永远是 2 的次幂，所以 n-1 通过 二进制表示，永远都是**尾端以连续 1 的形式**表示
  - 当 (n - 1) 和 hash 做与运算时，会保留 hash 中 后 x 位的 1

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

putVal(hash(key), key, value, false, evict);
final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {
    // ... 索引为 (n - 1) & hash
    tab[i = (n - 1) & hash]
    // ...
}
```

HashMap 在 jdk1.7 和 jdk1.8 的区别

- 1.7采用数组+单链表，1.8在单链表超过一定长度后改成**红黑树**存储
- 1.7插入元素到单链表中采用头插入法，1.8采用的是**尾插入法**。

    - 在 jdk1.7 中采用头插入法，扩容时会改变链表中元素原本的顺序，以至于**在并发场景下导致链表成环的问题**。而在 jdk1.8 中采用尾插入法，在扩容时会保持链表元素原本的顺序，就不会出现链表成环的问题了。
- 1.7**扩容时**需要重新计算哈希值和索引位置，消耗性能，而且多线程环境下会造成死锁。**1.8并不重新计算哈希值，巧妙地采用 hash 和扩容后容量减一进行&操作来计算新的索引位置**。
    - 索引是 (n - 1) & hash，当数组长度扩大一倍时，相当于在最高位加了一个 0，那么原来的 hash 值和新的数组长度减一做按位与运算，**结果要么和原来一样，要么多了一个旧容量的值**。
    - 比如旧容量： n=0100；新容量： n=1000； 则旧 (n - 1) = 0011； 新 (n - 1) = 0111
        - 如果 Key 的 hash 是 0011，则 0011 & 0011 == 0011 & 0111，位置不变
        - 如果 Key 的 hash 是 1111，则 1111 & 0011 != 1111 & 0111
            - 新位置增加了 0100 ,相当于旧数组的长度

HashMap 多线程有什么问题

1. 程序经常占了100%的CPU。CPU利用率过高，查看堆栈，你会发现程序都Hang在了 HashMap.get() 这个方法上了。那为什么会出现在get方法上呢?
    - 因为多线程情况下，对同一个 HashMap 做 put 操作可能导致两个或以上线程同时做 rehash 动作，就可能导致循环链表出现，一旦出现线程将无法终止，持续占用CPU，导致CPU使用率居高不下
2. 多线程 put 的时候可能导致元素丢失
    - 主要问题出在 addEntry 方法的 new Entry (hash, key, value, e)，如果两个线程都同时取得了 e ,则他们下一个元素都是 e，然后赋值给 table 元素的时候有一个成功有一个丢失。

什么叫 rehash？ Hash 表这个容器当有数据要插入时，都会检查容量有没有超过设定的 thredhold，如果超过，需要增大 Hash 表的尺寸，但是这样一来，整个 Hash 表里的元素都需要被重算一遍。这叫 rehash

解决方案:

1. Hashtable 替换 HashMap
2. Collections.synchronizedMap 将 HashMap 包装起来
3. ConcurrentHashMap 替换 HashMap

ConcurrentHashMap

ConcurrentHashMap 是 HashMap 的一个线程安全的、支持高效并发的版本。在默认理想状态下， ConcurrentHashMap 可以支持 16 个线程执行并发写操作及任意数量线程的读操作。

底层数据结构:

- <jdk1.7>： 使用 Segment 数组 + HashEntry 数组 + 链表
- <jdk1.8>： 使用 Node 数组 + 链表 + 红黑树

效率:

- <jdk1.7>： ConcurrentHashMap 使用的**分段锁**，如果一个线程占用一段，别的线程可以操作别的部分
- <jdk1.8>：简化结构， put 和 get 不用二次哈希，**一把锁只锁住一个链表或者一棵树**，并发效率更加提升

通过什么保证线程安全:

- <JDK1.7>: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。
- <jdk1.8>: 使用的是 synchronized 关键字（优化后效率很高）同步代码块和 cas (乐观锁)操作了维护并发。

CouncurrentHashMap <jdk1.7>

底层一个 Segments 数组，存储一个 Segments 对象，（其实就相当于一段一段的）一个 Segments 中储存一个 Entry 数组，存储的每个 Entry 对象又是一个链表头结点。

图示(相当于HashMap嵌套HashMap，只不过第一层是加锁的，分段锁思想，所以需要两次Hash计算才能确定位置):

```
|Segments Arr| -> |Segment(Lock) -> Entry List| -> |Entry -> NodeList| -> |KV Node|
```

- Segment 内部类，继承 ReentrantLock
- get 方法:
  i. 第一次哈希找到对应的 Segment 段，调用 Segment 中的 get 方法
  ii. 再次哈希找到对应的链表
  iii. 最后在链表中查找
- put 方法:

i. 首先确定段的位置
ii. 调用 Segment 中的 put 方法
    a. 加锁 lock()；
    b. 检查当前 Segment 数组中包含的 HashEntry 节点的个数，如果超过阈值就重新 hash (扩容)
        - 只是对 Segments 对象中的 Hashentry 数组进行重哈希
    c. 然后再次 hash 确定放的链表。
    d. 在对应的链表中查找是否相同节点，如果有直接覆盖，如果没有将其放置链表尾部

CouncurrentHashMap <jdk1.8>

putVal：

1. 如果数组"空"，进行数组初始化
2. 找该 hash 值对应的数组下标，得到第一个节点 f
    - 如果数组该位置为空，用一次 CAS 操作将新 new 出来的 Node 节点放入数组 i 下标位置
    - 如果不为空。用 synchronized 修饰

Collections 工具类

Collections 是集合工具类，用来对集合进行操作。 Collections 是一个操作 Set 、 List 和 Map 等集合的工具类。 Collections 中提供了一系列静态的方法对集合元素进行排序、查询和修改等操作，还提供了对集合对象设置不可变、对集合对象实现同步控制等方法：

- Collections.synchronizedXxx：同步集合
- emptyXxx：空不可变
- singletonXxx：单元素不可变
- unmodifiableXxx：不可变
- addAll：将所有指定元素添加到指定 collection 中
- reverse：反转指定列表 List 中元素的顺序。
- shuffle：集合元素进行随机排序，类似洗牌
- replaceAll：使用新值替换 List 对象的所有旧值

如何将集合里的对象进行去重？ 需要重写比较对象的 HashCode 与 equal 方法

- Stream 流式去重: distinct
- HashSet： 通过调用元素内部的 hashCode 和 equals 方法实现去重，首先调用 hashCode 方法，比较两个元素的哈希值，如果哈希值不同，直接认为是两个对象，停止比较。如果哈希值相同，再去调用 equals 方法，返回 true ， 认为是一个对象。返回 false ， 认为是两个对象。