2020 面试题总结——数据结构篇

1、ConcurrentHashMap 为何读不加锁。

在 JDK 1.8 中，ConcurrentHashMap 采用 Node+CAS+Synchronized 来保证并发安全，在读操作时没有加锁操作。先看一下 ConcurrentHashMap 读操作的相关源码。

```
transient volatile Node<K,V>[] table;//代表 table 的地址是 volatile，而不是数组元素是 volatile。这处是为了使得 table 在扩容时对其他线程也保持可见性。
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    //用 volatile 修饰 Node 的元素 val 和指针 next，在多线程环境下保持可见性。
    volatile V val;
    volatile Node<K,V> next;
    Node(int hash, K key, V val, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.val = val;
        this.next = next;
    }
    //省略其他代码
}
//在读操作中没有一处加锁。
public V get(Object key) {
    Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
    int h = spread(key.hashCode());//计算 hash 值
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (e = tabAt(tab, (n - 1) & h)) != null) {//读取首节点的 Node 元素
        if ((eh = e.hash) == h) {//如果 Key 相等则直接返回首节点
            if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                return e.val;
        }
        // eh（hash 值）为负值表示正在扩容，这个时候查的是 ForwardingNode 的 find 方法来定位到 nextTable 来
        //eh=-1，说明该节点是一个 ForwardingNode，正在迁移，此时调用 ForwardingNode 的 find 方法去 nextTable 里找。
        //eh=-2，说明该节点是一个 TreeBin，此时调用 TreeBin 的 find 方法遍历红黑树，由于红黑树有可能正在旋转变色，所以 find 里会有读写锁。
        //eh>=0，说明该节点下挂的是一个链表，直接遍历该链表即可。

        else if (eh < 0)
            return (p = e.find(h, key)) != null ? p.val : null;
        while ((e = e.next) != null) {//直接在链表遍历查询
            if (e.hash == h &&
                ((ek = e.key) == key || (ek != null && key.equals(ek))))
                return e.val;
        }
    }
    return null;
}
```

总结：

- 在 JDK 1.8 中，ConcurrentHashMap 的 get 操作不需要加锁，这也是它比其他并发集合比如 HashTable、用 Collections.synchronizedMap()包装的 HashMap 更为安全高效的原因。
- get 操作全程不需要加锁是因为 Node 的成员 val 是用 volatile 修饰的，**和数组用 volatile 修饰没有关系**。
- 数组用 volatile 修饰主要是保证数组在扩容的时候保持可见性。

2、HashMap 如何同步。

HashMap 有很多优点，但也有一个缺点：不是同步的。

当多线程并发访问一个哈希表时，需要在外部进行同步操作，否则会引发数据不同步问题。

当我们需要一个同步的 HashMap 时，有两种选择：

- 选择全局加锁，考虑用 Collections.synchronizedMap 包一层锁，变成个线程安全的 Map：Map m = Collections.synchronizedMap(new HashMap(...));
- 使用 ConcurrentHashMap。这两个选项之间的首选是使用 ConcurrentHashMap，这是因为我们不需要锁定整个对象。

此外，刚才提到 Collections.synchronizedMap，这里单独多说一点，它的原理超级简单，就是内部多加一个锁（在构造函数中初始化为 SynchronizedMap 类本身），每次调用 get、put 等方法时都用 synchronized 方法块来实现同步功能。

```
private static class SynchronizedMap<K,V>
    implements Map<K,V>, Serializable {
    private static final long serialVersionUID = 1978198479659022715L;
    private final Map<K,V> m;     // Backing Map
    final Object      mutex;        // Object on which to synchronize
    SynchronizedMap(Map<K,V> m) {
        this.m = Objects.requireNonNull(m);
        mutex = this;
    }
    SynchronizedMap(Map<K,V> m, Object mutex) {
        this.m = m;
        this.mutex = mutex;
    }
    public int size() {
        synchronized (mutex) {return m.size();}
    }
    public V get(Object key) {
        synchronized (mutex) {return m.get(key);}
    }
    public V put(K key, V value) {
        synchronized (mutex) {return m.put(key, value);}
    }
    public V remove(Object key) {
```

```
        synchronized (mutex) {return m.remove(key);}
    }
        // 省略其他方法
}
```

3、说说常见的集合有哪些吧。

Map 接口和 Collection 接口是所有集合类框架的父接口：

Collcetion 接口的子接口包括：Set 接口和 List 接口。

- Map 接口的实现类主要有：HashMap、HashTable、ConcurrentHashMap 以及 Properties 等。
- Set 接口的实现类主要有：HashSet、TreeSet、LinkedHashSet 等。
- List 接口的实现类主要有：ArrayList、LinkedList、Stack 以及 Vector 等。

如此图所示（网上找的，有点丑，凑付看吧）。

4、HashMap 与 HashTable 的区别。

主要有以下几点区别。

- HashMap 没有考虑同步，是线程不安全的；HashTable 在关键方法（put、get、contains、size 等）上使用了 Synchronized 关键字，是线程安全的。
- HashMap 允许 Key/Value 都为 null，后者 Key/Value 都不允许为 null。
- HashMap 继承自 AbstractMap 类；而 HashTable 继承自 Dictionary 类（较为陈旧）。
- 在 jdk1.8 中，HashMap 的底层结构是数组+链表+红黑树，而 HashTable 的底层结构是数组+链表。
- HashMap 对底层数组采取的懒加载，即当执行第一次 put 操作时才会创建数组；而 HashTable 在初始化时就创建了数组。
- HashMap 中数组的默认初始容量是 16，并且必须是 2 的指数倍数，扩容时 `newCapacity=2*oldCapacity`；而 HashTable 中默认的初始容量是 11，并且不要求必须是 2 的指数倍数，扩容时 `newCapacity=2*oldCapacity+1`。
- 在 hash 取模计算时，HashTable 的模数一般为素数，简单的做除取模结果会更为均匀，`int index = (hash & 0x7FFFFFFF) % tab.length;`
- HashMap 的模数为 2 的幂，直接用位运算来得到结果，效率要大大高于做除法，`i = (n - 1) & hash`；当然，为了解决哈希分布不均匀情况，所以在 hash 算法中引入了扰动计算，这又是后话了。

5、HashMap 的 put 方法的具体流程。

在 jdk1.8 以前的 HashMap 的实现是数组+链表，即使哈希函数取得再好，也很难达到元素百分百均匀分布。当 HashMap 中有大量的元素都存放在同一个桶中时，这个桶下有一条长长的链表，这个时候 HashMap 就相当于一个单链表，假如单链表有 n 个元素，遍历的时间复杂度就是 O(n)，完全失去了它的优势。

针对这种情况，JDK1.8 引入了红黑树（查找时间复杂度为 O(logn)）来优化这个问题。

下面具体看 put 方法的源码。

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // 1 如果 table 为空或者长度为 0，即没有元素，则使用 resize()方法扩容
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 2 计算插入存储的数组索引 i，如果数组当前位置为空，则不存在 Hash 冲突，可以直接插入元素。
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    // 3 插入新元素时，如果发生 Hash 冲突，则依次往下判断。
    else {
        Node<K,V> e; K k;
        // 3.1 判断 table[i]的元素的 key 是否与需要插入的 key 相同，若相同则直接用新的 value 覆盖掉旧的 value，判断元素相等原则是 equals()，所以 key 对象需要重写该方法。
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        // 3.2 判断需要插入的数据结构是红黑树还是链表，如果是红黑树，则直接在树中通过 putTreeVal()方法插入/更新键值对。
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        // 3.3 如果是链表，则在链表中插入/更新键值对
        else {
            // 3.3.1 遍历 table[i]，判断 key 是否已经存在，采用 equals 对比当前遍历结点的 key 与需要插入数据的 key，如果存在相同，则直接覆盖。
            // 3.3.2 遍历完毕后如果没有发现相同的 key，直接在链表尾部插入新的节点元素。
            // 插入完成后判断链表长度是否>TREEIFY_THRESHOLD（8），若是，则把链表转换为红黑树。
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
```

```
                p = e;
            }
        }
        // 3.4  如果 e 不为空，证明 key 已经存在，直接用新的 value 覆盖旧的 value 即可。
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    // 4 插入成功后，判断实际存在的键值对数量 size>最大容量，如果大于则进行扩容。
    if (++size > threshold)
        resize();
    // 5 插入成功时会调用的方法（默认实现为空）
    afterNodeInsertion(evict);
    return null;
}
```

简单的图片总结为：

HashMap 中的红黑树操作具体可以参考：Java 集合深入理解（17）：HashMap 在 JDK 1.8 后新增的红黑树结构

6、HashMap 的扩容操作是怎么实现的。

在 HashMap 中，通过 resize()方法来进行扩容或者初始化操作。直接上源码。

```
final Node<K,V>[] resize() {
        // 1 复制一份扩容前的数组（当前数组）
    Node<K,V>[] oldTab = table;
        // 2 保存旧的数组长度，阈值
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
        // 3 判断扩容前的数组容量
    if (oldCap > 0) {
        // 3.1 若扩容前的数组容量超过最大值，则不再扩容。
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        // 3.2 若没有超过最大值，则扩容为原来二倍。
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    // 4 如果旧容量为 0，并且旧阈值>0，说明之前创建了哈希表但没有添加元素，初始化容量等于阈值。
    else if (oldThr > 0) // initial capacity was placed in threshold
```

```
            newCap = oldThr;
        // 5 如果旧容量、旧阈值都为 0，说明还没创建哈希表，容量为默认值，阈值为容量*加载因子。
        else {               // zero initial threshold signifies using defaults
            newCap = DEFAULT_INITIAL_CAPACITY;
            newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
        }
            // 6 如果新的阈值为 0，则用新容量*加载因子 重新计算一次。
        if (newThr == 0) {
            float ft = (float)newCap * loadFactor;
            newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                      (int)ft : Integer.MAX_VALUE);
        }
        // 7 更新阈值。
        threshold = newThr;
        // 8 创建新链表数组，容量是原来两倍。
        @SuppressWarnings({"rawtypes","unchecked"})
            Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        table = newTab;
        // 9 接下来是遍历复制。
        if (oldTab != null) {
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
                if ((e = oldTab[j]) != null) {
                    // 9.1 旧的桶置为空
                    oldTab[j] = null;
                    // 9.2 如果当前桶只有一个元素，则直接赋值给新 table 的对应位置。
                    if (e.next == null)
                        newTab[e.hash & (newCap - 1)] = e;
                    // 9.3 如果当前桶是树形结构，则要把新 table 当前位置桶也变成树结构。
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                    // 9.4 保留旧哈希表桶中链表元素的顺序
                    else { // preserve order
                        Node<K,V> loHead = null, loTail = null;//按原始链表顺序，过滤出来扩容后位置不变的元素（低位=0），放在一起。
                        Node<K,V> hiHead = null, hiTail = null;//按原始链表顺序，过滤出来扩容后位置改变到（index+oldCap）的元素（高位=0），放在一起。
                        Node<K,V> next;
                        // 9.4.1 do-while 循环赋值给新哈希表
                        do {
                            next = e.next;
                            // 9.4.2 通过计算(e.hash & oldCap) == 0 分成两条链表，再将两条链表散列到新数组的不同位置。
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            else {
                                if (hiTail == null)
                                    hiHead = e;
                                else
```

```
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    // 9.4.3 放到原索引位置不变的桶中。
                    if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
                    }
                    // 9.4.4 放到原索引+oldCap 位置的桶中。
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
    }
    return newTab;
}
```

有一点注意区别，JDK1.7 中 resize 的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但在 JDK1.8 不会倒置，链表元素还是按照之前顺序。

7、HashMap 是怎么解决哈希冲突的。

哈希冲突：当两个不同的输入值，根据同一散列函数计算出相同的散列值的现象，我们就把它叫做哈希碰撞。

在 Java 中，保存数据有两种比较简单的数据结构：数组和链表。数组的特点是：寻址容易，插入和删除困难。链表的特点是：寻址困难，但插入和删除容易。所以我们将数组和链表结合在一起，发挥两者的优势，使用一种叫做链地址法的方式来解决哈希冲突。

这样我们就可以拥有相同哈希值的对象组织成的一个链表放在 hash 值对应的 bucket 下，但相比 Key.hashCode()返回的 int 类型，我们 HashMap 初始的**容量大小 DEFAULT_INITIAL_CAPACITY = 1 << 4（即 2 的四次方为 16）要远小于 int 类型的范围**，所以我们如果只是单纯的使用 hashcode 取余来获取对应位置的 bucket，这将会大大增加哈希碰撞的几率，并且最坏情况下还会将 HashMap 变成一个单链表。所以肯定要对 hashCode 做一定优化。

来看 HashMap 的 hash()函数。上面提到的问题，主要是因为如果使用 hashCode 取余，那么相当于参与运算的只有 hashCode 的低位，高位是没有起到任何作用的，所以我们的思路就是让**hashCode 取值出的高位也参与运算，进一步降低 hash 碰撞的概率，使得数据分布更平均，我们把这样的操作称为扰动，**在 JDK 1.8 中的 hash()函数如下：

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

相比在 JDK 1.7 中的 4 次位运算，5 次异或运算（9 次扰动），在 1.8 中，只进行了 1 次位运算和 1 次异或运算（2 次扰动），更为简洁了。两次扰动已经达到了高低位同时参与运算的目的，提高了对应数组存储下标位置的随机性和均匀性。

通过上面的链地址法（使用散列表）和扰动函数，数据分布更为均匀，哈希碰撞也减少了。但是当 HashMap 中存在大量的数据时，假如某个 bucket 下对应的链表中有 n 个元素，那么遍历时间复杂度就变成了 O(n)，针对这个问题，JDK 1.8 在 HashMap 中新增了红黑树的数据结构，进一步使得遍历复杂度降低至 O(logn)。

简单**总结**一下 HashMap 是如何有效解决哈希碰撞的：

- 使用链地址法（散列表）来链接拥有相同 hash 值的元素；
- 使用 2 次扰动（hash 函数）来降低哈希冲突的概率，使得概率分布更为均匀；
- 引入红黑树进一步降低遍历的时间复杂度。

参考链接：全网把 Map 中的 hash()分析的最透彻的文章，别无二家。

8、HashMap 为什么不直接使用 hashCode()处理后的哈希值之直接作为 table 的下标。

hashCode()方法返回的是 int 整数类型，其范围为-(2 ^ 31)~(2 ^ 31 - 1)，约有 40 亿个映射空间，而 HashMap 的容量范围是在 16（初始化默认值）~2 ^ 30，HashMap 通常情况下是取不到最大值的，并且设备上也难以提供这么多的存储空间，从而导致通过 hashCode()计算出的哈希值可能不在数组大小范围内，进而无法匹配存储位置。

所以解决方法为：

- 在 HashMap 中实现了自己的 hash()方法，通过两次扰动使得元素对象的哈希值高低位自行进行异或运算，降低哈希碰撞概率也使得数据分布更平均。（具体看上一题）
- 在保证数组长度为 2 的幂次方的时候，使用 hash()运算之后的值与运算（&）（数组长度 - 1）来获取数组下标的方式进行存储，这样一来是比取余操作更加有效率，二来也是因为只有当数组长度为 2 的幂次方时，h&(length-1)才等价于 h%length，三来解决了“哈希值与数组大小范围不匹配”的问题。

为什么数组长度要保证 2 的幂次方：

- 只有当数组长度为 2 的幂次方时，h&(length-1)才等价于 h%length，可以用位运算来代替做除取模运算，实现了 key 的定位，2 的幂次方也可以减少冲突次数，提高 HashMap 的查询效率；
- 如果 length 为 2 的次幂 则 length-1 转化为二进制必定是 11111……的形式，在于 h 的二进制与操作效率会非常的快，而且空间不浪费；如果 length 不是 2 的次幂，比如 length 为 15，则 length - 1 为 14，对应的二进制为 1110，在于 h 与操作，最后一位都为 0 ，而 0001，0011，0101，1001，1011，0111，1101 这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！这样就会造成空间的浪费。
- 当然，HashTable 就没有采用 2 的幂作为数组长度，而是采用素数。素数的话是用简单做除取模方法来获取下标 index，而不是位运算，效率低了不少，但分布也很均匀。

9、为什么 HashMap 中 String、Integer 这样的包装类适合作为 Key。

String、Integer 等包装类的特性能够保证 Hash 值的不可更改性和计算准确性，能够有效的减少 Hash 碰撞的几率。

- 都是 final 类型，即不可变性，保证 key 的不可更改性，不会存在获取 hash 值不同的情况。
- 内部已经重写 equals()、hashCode()等方法，遵守了 HashMap 内部的规范，不容易出现 Hash 值计算错误的情况。

如果想要自己写的类作为 Key，需要重写 hashCode()和 equals()方法。

- 重写 HashCode()是因为需要计算存储数据的存储位置，需要注意，不要试图从散列码计算中排除掉一个对象的关键部分来提高性能，这样虽然能更快但是可能会导致更多的 Hash 碰撞。
- 重写 equals()方法，需要遵守自反性、对称性、传递性、一致性以及对于任务非 null 的引用之 x，x.equals(null)必须返回 false 这几个特性，目的就是为了保证 Key 在哈希表中的唯一性。

10、ConcurrentHashMap 和 HashMap 的区别。

ConcurrentHashMap 结合了 HashMap 和 HashTable 二者的优势。HashMap 没有考虑同步，HashTable 却要在每次同此执行时都要锁住整个结构（用 Synchronized 关键字）。ConcurrentHashMap 锁的方式是稍微细粒度的。

在 JDK 1.8 中，ConcurrentHashMap 放弃了臃肿的分段锁设计，取而代之的是采用 Node+CAS+Synchronized 来保证并发安全进行实现。

以插入元素（put 方法）为例。

```
public V put(K key, V value) {
    return putVal(key, value, false);
}
final V putVal(K key, V value, boolean onlyIfAbsent) {
    // 1 key 和 value 不能为 null。
    if (key == null || value == null) throw new NullPointerException();
    // 2 通过 spread 函数获取 hash 值（与 HashMap 类似，也加入了扰动）
    int hash = spread(key.hashCode());
    int binCount = 0;//用来计算在这个桶总共有多少元素，用来控制扩容或者转换为红黑树。
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        // 3 table 为空，则先初始化数组。
        if (tab == null || (n = tab.length) == 0)
            tab = initTable();
        // 4 如果 table 对应位置为空，则调用 CAS 插入相应的数据，注意这个地方没有加锁。
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
        // 5 如果取出来的节点的 hash 值是 MOVED(-1)的话，则表示当前正在对这个数组进行扩容，复制到新的数组，则当前线程也去帮助复制。
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            // 6 如果这个位置有元素的话，则对该节点加 synchronized 锁。
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    // 7 如果是链表的话(hash 大于 0)，就对这个链表的所有元素进行遍历。
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            // 8 若找到 Key 和 Key 的 hash 值都一样的节点，则用新的 value 值替换旧值。
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            Node<K,V> pred = e;
                            // 9 若没有找到相同 Key 节点的话，则在链表后面添加新的节点。
```

```
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key,
                                                          value, null);
                                break;
                            }
                        }
                    }
                    // 10 当前桶的数据结构为红黑树时，调用 putTreeVal 方法将元素
添加到树中。
                    else if (f instanceof TreeBin) {
                        Node<K,V> p;
                        binCount = 2;
                        if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                                       value)) != null) {
                            oldVal = p.val;
                            if (!onlyIfAbsent)
                                p.val = value;
                        }
                    }
                }
            }
            // 11 当在同一个节点的数目达到 TREEIFY_THRESHOLD(8)个的时候，则扩张
数组或将给节点的数据转为树。
            if (binCount != 0) {
                if (binCount >= TREEIFY_THRESHOLD)
                    treeifyBin(tab, i);
                if (oldVal != null)
                    return oldVal;
                break;
            }
        }
    }
    // 12 执行 addCount()方法尝试更新元素个数 baseCount。
    addCount(1L, binCount);
    return null;
}
```

11、Java 集合的快速失败机制“fail-fast”，以及安全失败“fail-safe”。

“fail-fast”是 java 集合的一种错误检测机制，当多个线程对集合进行结构上的改变的操作时，有可能会产生 fail-fast 机制。

例如：假设存在两个线程（线程 1、线程 2），线程 1 通过 Iterator 在遍历集合 A 中的元素，在某个时候线程 2 修改了集合 A 的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生 fail-fast 机制。

原因：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变 modCount 的值。每当迭代器使用 hashNext()/next()遍历下一个元素之前，都会检测 modCount 变量是否为 expectedmodCount 值，是的话就返回遍历；否则抛出异常 ConcurrentModification，终止遍历。

来看一下 ArrayList 源码部分，在 next 方法执行时，会先执行 checkForComodification 方法来检查。

```
public E next() {
    checkForComodification();
    int i = cursor;
    if (i >= size)
        throw new NoSuchElementException();
    Object[] elementData = ArrayList.this.elementData;
    if (i >= elementData.length)
        throw new ConcurrentModificationException();
    cursor = i + 1;
    return (E) elementData[lastRet = i];
}
final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
```

java.util 包下的集合类都是快速失败机制，不能在多线程下发生并修改（迭代过程中被修改）。

解决办法：

- 1 在遍历过程中，所有涉及到改变 modCount 值得地方全部加上 synchronized。
- 使用 CopyOnWriteArrayList 来替换 ArrayList。

与“fail-fast”对应的是“fail-safe”。

采用安全失败机制的集合容器,在遍历时不是直接在集合内容上访问的,而是先 copy 原有集合内容,在拷贝的集合上进行遍历。由于迭代时是对原集合的拷贝的值进行遍历,所以在遍历过程中对原集合所作的修改并不能被迭代器检测到,所以不会触发 ConcurrentModificationException 异常。

基于拷贝内容的迭代虽然避免了 ConcurrentModificationException 异常，但同样地，迭代器并不能访问到修改后的内容，简单来说，迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器行为是不知道的。

java.util.concurrent 包下的容器都是安全失败的,可以在多线程下并发使用,并发修改。

12、ArrayList 和 CopyOnWriteArrayList 的区别。

ArrayList 底层就是数组实现的，查询的时候直接根据索引就可以很快的找到对应的元素，改也是如此，找到 index 对应元素直接替换。而增加和删除就涉及到数据元素的移动，会比较慢。

需要注意的是，ArrayList 扩容时，newCapacity = oldCapacity*1.5。看一下 grow 函数。

```
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);//1.5 倍
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

主要看 CopyOnWriteArrayList，主要是读写分离，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时，读取操作可以安全地执行，看一下源码。

```
private transient volatile Object[] array;//保存元素对象的动态数组，注意是用 volatile transient 声明，只能通过 setArray()/getArray()访问。
final transient ReentrantLock lock = new ReentrantLock();//类内唯一一把锁
final Object[] getArray() {
    return array;
}
final void setArray(Object[] a) {
    array = a;
}
//添加操作
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    lock.lock();//获得锁
    try {
        Object[] elements = getArray();//复制一个新的数组
        int len = elements.length;
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        newElements[len] = e;//插入新值
        setArray(newElements);//将新的数组指向原来的引用
        return true;
    } finally {
        lock.unlock();//释放锁
    }
}
//删除操作
public E remove(int index) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        E oldValue = get(elements, index);
        int numMoved = len - index - 1;
        if (numMoved == 0)//如果删除的是最后一个元素，直接复制出来前面 len-1 元素即可。
            setArray(Arrays.copyOf(elements, len - 1));
        else {//如果删除的是中间元素，就需要分别复制前后元素了。
            Object[] newElements = new Object[len - 1];
```

```
            System.arraycopy(elements, 0, newElements, 0, index);
            System.arraycopy(elements, index + 1, newElements, index,
                             numMoved);
            setArray(newElements);
        }
        return oldValue;
    } finally {
        lock.unlock();
    }
}
//读操作，不需要加锁。
public E get(int index) {
    return get(getArray(), index);
}
private E get(Object[] a, int index) {
    return (E) a[index];
}
```

CopyOnWriteArrayList 增删改都需要获得锁，并且只有一把锁，而读操作不需要锁，支持并发。注意， CopyOnWriteArrayList **没有扩容机制**，每次增删操作都是新建数组操作。再解释一下 CopyOnWriteArrayList 为什么是“fail-safe”，因为在迭代器遍历开始时，获取的是当前数组，而后续的增删改，都是新建数组操作，不会影响到迭代器。

另外，CopyOnWriteArrayList 的并发安全性能比 vector 要好，vector 是增删改查方法都加了 synchronized 关键字，但是每个方法执行的时候都要去获得锁，性能就会大大下降，而 CopyOnWriteArrayList 只是在增删改上加锁，读不加锁，在读方面的性能好于 vector，CopyOnWriteArrayList 支持读多写少的并发情况。

但需要注意的点有

- 由于 CopyOnWriteArrayList 写操作的时候，需要拷贝数组，会消耗内存。如果原数组内容比较多的情况下，可能会导致 young gc 或者 full gc。
- 不能用于实时读写的场景，像拷贝数组、新增元素都需要时间。虽然 CopyOnWriteArrayList 能做到最终一致性,但是还是没法满足实时性要求。

13、ArrayList 和 LinkedList 的区别。

主要有以下几点区别：

- LinkedList 实现了 List 和 Deque 接口，一般称为双向链表；ArrayList 实现了 List 接口，是动态数组。
- LinkedList 在插入和删除数据时效率更高，ArrayList 在查找某个 index 的数据时效率更高。
- LinkedList 比 ArrayList 需要更多内存。

14、HashSet 是如何保证数据不可重复的。

HashSet 的底层其实就是 HashMap，只不过 HashSet 是实现了 Set 接口，并且把数据作为 Key 值，而 Value 值一直使用一个相同的虚值来保存。

```
private transient HashMap<E,Object> map;//底层实现的 HashMap
private static final Object PRESENT = new Object();//虚值
public HashSet() {
    map = new HashMap<>();
}
public boolean add(E e) {
    return map.put(e, PRESENT)==null;//如果插入重复 Value，map 会返回上次插入的旧值 PRENET，就不为 null，这句话就返回 false，代表插入失败。
}
```

由于 HashMap 的 K 值本身就不允许重复，并且在 HashMap 中如果 K/V 相同时，会用新的 V 覆盖掉旧的 V，然后返回旧的 V，那么在 HashSet 中执行这一句话始终会返回一个 false，导致插入失败，这样就保证了数据的不可重复性。

15、BlockingQueue 是什么。

java.util.concurrent.BlockingQueue 是一个队列，在进行检索或移除一个元素的时候，它会等待队列变为非空；当添加一个元素时，它会等待队列中的可用空间。

BlockingQueue 接口是 java 集合框架的一部分，主要用于实现生产者-消费者模式。这样我们就不需要担心等待生产者有可用的空间，以及消费者有可用的对象。因为它们都在 BlockingQueue 的实现类中被处理了。

Java 提供了几种 BlockingQueue 的实现，比如 ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue,、SynchronousQueue 等。

16、HashMap 为什么选用红黑树。

HashMap 在 jdk1.8 之后引入了红黑树的概念，表示若桶中链表元素超过 8 时，会自动转化成红黑树；若桶中元素小于等于 6 时，树结构还原成链表形式。

因为红黑树的平均查找长度是 log(n)，长度为 8 的时候，平均查找长度为 3，如果继续使用链表，平均查找长度为 8/2=4，这才有转换为树的必要。链表长度如果是小于等于 6，6/2=3，虽然速度也很快的，但是转化为树结构和生成树的时间并不会太短。

并且，红黑树不像平衡树那样追求绝对的平衡，省去了很多没有必要的调平衡操作，插入删除元素等操作效率提高了很多。

另外，上下阈值选择 6 和 8 的情况下，中间有个差值 7 可以防止链表和树之间频繁的转换。假设一下，如果设计成链表个数超过 8 则链表转换成树结构，链表个数小于 8 则树结构转换成链表，如果一个 HashMap 不停的插入、删除元素，链表个数在 8 左右徘徊，就会频繁的发生树转链表、链表转树，效率会很低。

17、Iterator 中是否存在 Add 方法。

Iterator 是没有 add 方法的，但是有 remove 方法。Iterator.remove()是唯一安全的方式来在迭代过程中修改集合；如果在迭代过程中以任何其它的方式修改了基本集合将会产生未知的行为。而且每调用一次 next()方法，remove()方法只能被调用一次，如果违反这个规则将抛出一个异常。

ListIterator 是一个功能更强大的迭代器，继承于 Iterator，可以通过调用 listIterator()方法产生一个指向 List 开始处的 ListIterator。它内部实现了一个双向链表，是具有 add 方法的。

两者的主要区别如下：

- ListIterator 有 add()方法，可以在遍历的时候向 List 中添加对象，而 Iterator 不可以。
- ListIterator 和 Iterator 都有 hasNext()和 next()方法，可以实现顺序往后遍历，但是 ListIterator 有 hasPrevious()和 previous()方法，可以实现逆向遍历，Iterator 就不可以。
- ListIterator 可以定位当前的索引位置，nextIndex()和 previousIndex()可以实现。Iterator 没有这个功能。
- 都可以实现删除对象，但是 ListIterator 可以实现对象的修改，set()方法可以实现。Iterator 仅能遍历，不能修改。