主讲老师：Fox

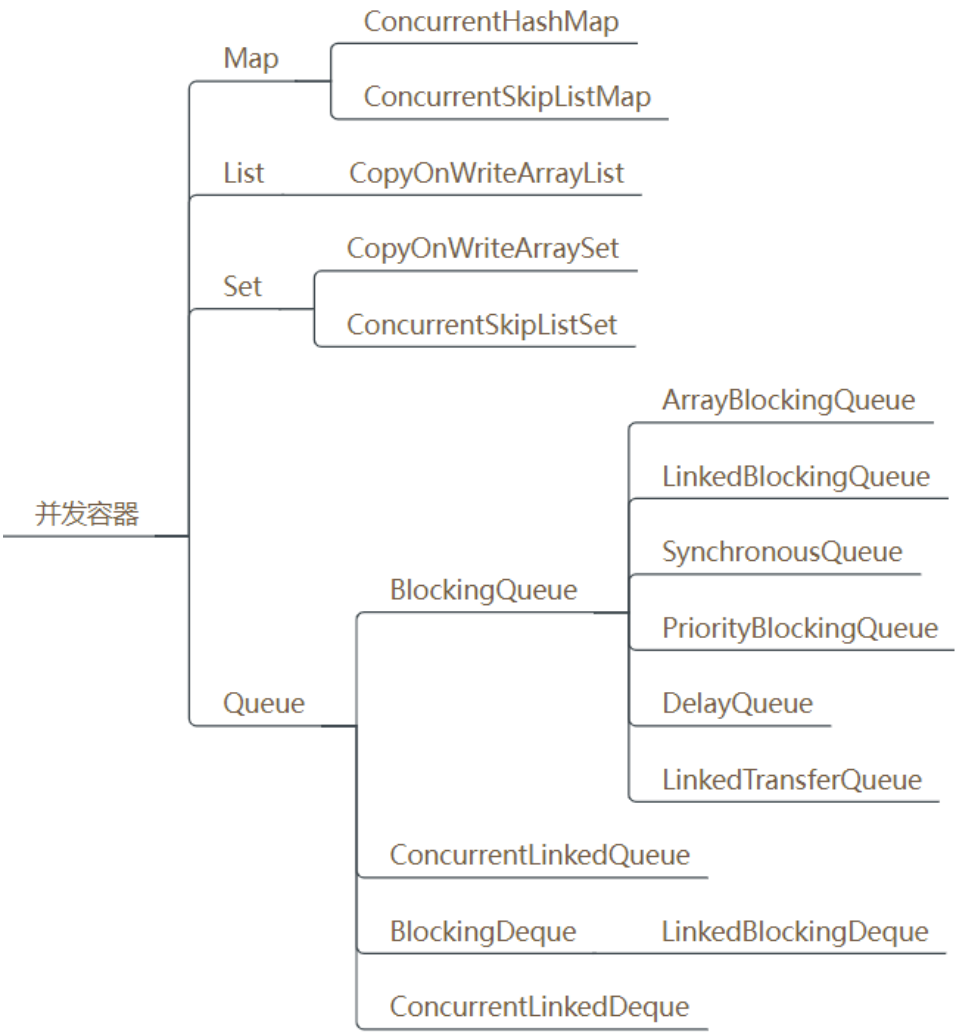
> 有道笔记地址：https://note.youdao.com/s/YctrFhyh

1. JUC包下的并发容器

Java的集合容器框架中，主要有四大类别：List、Set、Queue、Map，大家熟知的这些集合类ArrayList、LinkedList、HashMap这些容器都是非线程安全的。

所以，Java先提供了同步容器供用户使用。同步容器可以简单地理解为通过synchronized来实现同步的容器，比如Vector、Hashtable以及SynchronizedList等容器。这样做的代价是削弱了并发性，当多个线程共同竞争容器级的锁时，吞吐量就会降低。

因此为了解决同步容器的性能问题，所以才有了并发容器。java.util.concurrent包中提供了多种并发类容器：



CopyOnWriteArrayList

对应的非并发容器：ArrayList

目标：代替Vector、synchronizedList

原理：利用高并发往往是读多写少的特性，对读操作不加锁，对写操作，先复制一份新的集合，在新的集合上面修改，然后将新集合赋值给旧的引用，并通过volatile 保证其可见性，当然写操作的锁是必

不可少的了。

CopyOnWriteArraySet

对应的非并发容器：HashSet

目标：代替synchronizedSet

原理：基于CopyOnWriteArrayList实现，其唯一的不同是在add时调用的是CopyOnWriteArrayList的addIfAbsent方法，其遍历当前Object数组，如Object数组中已有了当前元素，则直接返回，如果没有则放入Object数组的尾部，并返回。

ConcurrentHashMap

对应的非并发容器：HashMap

目标：代替Hashtable、synchronizedMap，支持复合操作

原理：JDK6中采用一种更加细粒度的加锁机制Segment“分段锁”，JDK8中采用CAS无锁算法。

ConcurrentSkipListMap

对应的非并发容器：TreeMap

目标：代替synchronizedSortedMap(TreeMap)

原理：Skip list（跳表）是一种可以代替平衡树的数据结构，默认是按照Key值升序的。

面试题： CopyOnWriteArrayList的底层原理是怎样的?

2. CopyOnWriteArrayList

CopyOnWriteArrayList 是 Java 中的一种线程安全的 List，它是一个可变的数组，支持并发读和写。与普通的 ArrayList 不同，它的读取操作不需要加锁，因此具有很高的并发性能。

2.1 应用场景

CopyOnWriteArrayList 的应用场景主要有两个方面：

1. 读多写少的场景

由于 CopyOnWriteArrayList 的读操作不需要加锁，因此它非常适合在读多写少的场景中使用。例如，一个读取频率比写入频率高得多的缓存，使用 CopyOnWriteArrayList 可以提高读取性能，并减少锁竞争的开销。

2. 不需要实时更新的数据

由于 CopyOnWriteArrayList 读取的数据可能不是最新的，因此它适合于不需要实时更新的数据。例如，在日志应用中，为了保证应用的性能，日志记录的操作可能被缓冲，并不是实时写入文件系统，而是在某个时刻批量写入。这种情况下，使用 CopyOnWriteArrayList 可以避免多个线程之间的竞争，提高应用的性能。

2.2 CopyOnWriteArrayList使用

基本使用

和 ArrayList 在使用方式方面很类似。

```
// 创建一个 CopyOnWriteArrayList 对象
CopyOnWriteArrayList phaser = new CopyOnWriteArrayList();
// 新增
copyOnWriteArrayList.add(1);
// 设置（指定下标）
copyOnWriteArrayList.set(0, 2);
// 获取（查询）
copyOnWriteArrayList.get(0);
// 删除
copyOnWriteArrayList.remove(0);
// 清空
copyOnWriteArrayList.clear();
// 是否为空
copyOnWriteArrayList.isEmpty();
// 是否包含
copyOnWriteArrayList.contains(1);
// 获取元素个数
copyOnWriteArrayList.size();

```

IP 黑名单判定

当应用接入外部请求后，为了防范风险，一般会对请求做一些特征判定，如对请求 IP 是否合法的判定就是一种。IP 黑名单偶尔会被系统运维人员做更新

```
public class CopyOnWriteArrayListDemo {

    private static CopyOnWriteArrayList<String> copyOnWriteArrayList = new
CopyOnWriteArrayList<>();
    // 模拟初始化的黑名单数据
    static {
        copyOnWriteArrayList.add("ipAddr0");
```

```
        copyOnWriteArrayList.add("ipAddr1");
        copyOnWriteArrayList.add("ipAddr2");
    }

    public static void main(String[] args) throws InterruptedException {
        Runnable task = new Runnable() {
            public void run() {
                // 模拟接入用时
                try {
                    Thread.sleep(new Random().nextInt(5000));
                } catch (Exception e) {}

                String currentIP = "ipAddr" + new Random().nextInt(6);
                if (copyOnWriteArrayList.contains(currentIP)) {
                    System.out.println(Thread.currentThread().getName() + " IP " + currentIP + "命中黑名单，拒绝接入处理");
                    return;
                }
                System.out.println(Thread.currentThread().getName() + " IP " + currentIP + "接入处理...");
            }
        };
        new Thread(task, "请求1").start();
        new Thread(task, "请求2").start();
        new Thread(task, "请求3").start();

        new Thread(new Runnable() {
            public void run() {
                // 模拟用时
                try {
                    Thread.sleep(new Random().nextInt(2000));
                } catch (Exception e) {}

                String newBlackIP = "ipAddr3";
                copyOnWriteArrayList.add(newBlackIP);
                System.out.println(Thread.currentThread().getName() + " 添加了新的非法IP " + newBlackIP);
            }
        }, "IP黑名单更新").start();

```

```
44          Thread.sleep(1000000);
45      }
46  }
```

2.3 CopyOnWriteArrayList原理

CopyOnWriteArrayList 内部使用了一种称为“写时复制”的机制。当需要进行写操作时，它会创建一个新的数组，并将原始数组的内容复制到新数组中，然后进行写操作。因此，读操作不会被写操作阻塞，读操作返回的结果可能不是最新的，但是对于许多应用场景来说，这是可以接受的。此外，由于读操作不需要加锁，因此它可以支持更高的并发度。

CopyOnWriteArrayList 的缺陷

CopyOnWriteArrayList 有几个缺点：

- 由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致 young gc 或者 full gc
- 不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个 set 操作后，读取到数据可能还是旧的，虽然 CopyOnWriteArrayList 能做到最终一致性，但是还是没法满足实时性要求；
- CopyOnWriteArrayList 合适读多写少的场景，不过这类慎用。因为谁也没法保证 CopyOnWriteArrayList 到底要放置多少数据，万一数据稍微有点多，每次 add/set 都要重新复制数组，这个代价实在太高昂了。在高性能的互联网应用中，这种操作分分钟引起故障。

思考：ArrayList存在什么问题，为什么CopyOnWriteArrayList要设计写操作时拷贝数组的方案?

2.4 扩展知识：迭代器的 fail-fast 与 fail-safe 机制

在 Java 中，迭代器（Iterator）在迭代的过程中，如果底层的集合被修改（添加或删除元素），不同的迭代器对此的表现行为是不一样的，可分为两类：Fail-Fast（快速失败）和 Fail-Safe（安全失败）。

fail-fast 机制

fail-fast 机制是java集合(Collection)中的一种错误机制。当多个线程对同一个集合的内容进行操作时，就可能会产生 fail-fast 事件。例如：当某一个线程A通过 iterator 去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程A访问集合时，就会抛出 ConcurrentModificationException异常，产生 fail-fast 事件。

在 java.util 包中的集合，如 ArrayList、HashMap 等，它们的迭代器默认都是采用 Fail-Fast 机制。

fail-fast解决方案

- 方案一：在遍历过程中所有涉及到改变modCount 值的地方全部加上synchronized 或者直接使用Collection#synchronizedList，这样就可以解决问题，但是不推荐，因为增删造成的同步锁可能会阻塞遍历操作。
- 方案二：使用CopyOnWriteArrayList 替换 ArrayList，推荐使用该方案（即fail-safe）。

fail-safe机制

任何对集合结构的修改都会在一个复制的集合上进行，因此不会抛出ConcurrentModificationException。在 java.util.concurrent 包中的集合，如CopyOnWriteArrayList、ConcurrentHashMap 等，它们的迭代器一般都是采用 Fail-Safe 机制。

缺点：

- 采用 Fail-Safe 机制的集合类都是线程安全的，但是它们无法保证数据的实时一致性，它们只能保证数据的最终一致性。在迭代过程中，如果集合被修改了，可能读取到的仍然是旧的数据。
- Fail-Safe 机制还存在另外一个问题，就是内存占用。由于这类集合一般都是通过复制来实现读写分离的，因此它们会创建出更多的对象，导致占用更多的内存，甚至可能引起频繁的垃圾回收，严重影响性能。

2. ConcurrentHashMap

ConcurrentHashMap 是 Java 中线程安全的哈希表，它支持高并发并且能够同时进行读写操作。在JDK1.8之前，ConcurrentHashMap使用分段锁以在保证线程安全的同时获得更大的效率。JDK1.8开始舍弃了分段锁，使用自旋+CAS+synchronized关键字来实现同步。官方的解释中：一是节省内存空间，二是分段锁需要更多的内存空间，而大多数情况下，并发粒度达不到设置的粒度，竞争概率较小，反而导致更新的长时间等待（因为锁定一段后整个段就无法更新了）三是提高GC效率。

2.1 应用场景

ConcurrentHashMap 的应用场景包括但不限于以下几种：

1. 共享数据的线程安全：在多线程编程中，如果需要进行共享数据的读写，可以使用 ConcurrentHashMap 保证线程安全。
2. 缓存：ConcurrentHashMap 的高并发性能和线程安全能力，使其成为一种很好的缓存实现方案。在多线程环境下，使用 ConcurrentHashMap 作为缓存的数据结构，能够提高程序的并发性能，同时保证数据的一致性。

2.2 ConcurrentHashMap使用

基本用法

```java
// 创建一个 ConcurrentHashMap 对象
ConcurrentHashMap<Object, Object> concurrentHashMap = new ConcurrentHashMap<>();
// 添加键值对
concurrentHashMap.put("key", "value");
// 添加一批键值对
concurrentHashMap.putAll(new HashMap());
// 使用指定的键获取值
concurrentHashMap.get("key");
// 判定是否为空
concurrentHashMap.isEmpty();
// 获取已经添加的键值对个数
concurrentHashMap.size();
// 获取已经添加的所有键的集合
concurrentHashMap.keys();
// 获取已经添加的所有值的集合
concurrentHashMap.values();
// 清空
concurrentHashMap.clear();

```

其他方法：

1. V putIfAbsent(K key, V value)

如果 key 对应的 value 不存在，则 put 进去，返回 null。否则不 put，返回已存在的 value。

2. boolean remove(Object key, Object value)

如果 key 对应的值是 value，则移除 K-V，返回 true。否则不移除，返回 false。

3. boolean replace(K key, V oldValue, V newValue)

如果 key 对应的当前值是 oldValue，则替换为 newValue，返回 true。否则不替换，返回 false。

统计文件中英文字母出现的总次数

```java
public class ConcurrentHashMapDemo {

    private static ConcurrentHashMap<String, AtomicLong> concurrentHashMap = new ConcurrentHashMap<>();
    // 创建一个 CountDownLatch 对象用于统计线程控制
    private static CountDownLatch countDownLatch = new CountDownLatch(3);
    // 模拟文本文件中的单词
    private static String[] words = {"we", "it", "is"};

```

```
    public static void main(String[] args) throws InterruptedException {
        Runnable task = new Runnable() {
            public void run() {
                for(int i=0; i<3; i++) {
                    // 模拟从文本文件中读取到的单词
                    String word = words[new Random().nextInt(3)];
                    // 尝试获取全局统计结果
                    AtomicLong number = concurrentHashMap.get(word);
                    // 在未获取到的情况下，进行初次统计结果设置
                    if (number == null) {
                        // 在设置时发现如果不存在则初始化
                        AtomicLong newNumber = new AtomicLong(0);
                        number = concurrentHashMap.putIfAbsent(word, newNumber);
                        if (number == null) {
                            number = newNumber;
                        }
                    }
                    // 在获取到的情况下，统计次数直接加1
                    number.incrementAndGet();

                    System.out.println(Thread.currentThread().getName() + ":" + word + " 出现" + number + " 次");
                }
                countDownLatch.countDown();
            }
        };

        new Thread(task, "线程1").start();
        new Thread(task, "线程2").start();
        new Thread(task, "线程3").start();

        try {
            countDownLatch.await();
            System.out.println(concurrentHashMap.toString());
        } catch (Exception e) {}
    }
}

```

2.3 数据结构

HashTable的数据结构

JDK1.7 中的ConcurrentHashMap

在jdk1.7及其以下的版本中，结构是用Segments数组 + HashEntry数组 + 链表实现的

JDK1.8中的ConcurrentHashMap

jdk1.8抛弃了Segments分段锁的方案，而是改用了和HashMap一样的结构操作，也就是数组 + 链表 + 红黑树结构，比jdk1.7中的ConcurrentHashMap提高了效率，在并发方面，使用了cas + synchronized的方式保证数据的一致性

链表转化为红黑树需要满足2个条件:

- 链表的节点数量大于等于树化阈值8
- Node数组的长度大于等于最小树化容量值64

```
#树化阈值为8
static final int TREEIFY_THRESHOLD = 8;
#最小树化容量值为64
static final int MIN_TREEIFY_CAPACITY = 64;
```

2.4 ConcurrentHashMap源码分析

https://vip.tulingxueyuan.cn/p/t_pc/goods_pc_detail/goods_detail/p_60339636e4b029faba19895b

3. ConcurrentSkipListMap

ConcurrentSkipListMap 是 Java 中的一种线程安全、基于跳表实现的有序映射（Map）数据结构。它是对 TreeMap 的并发实现，支持高并发读写操作。

ConcurrentSkipListMap适用于需要高并发性能、支持**有序性**和区间查询的场景，能够有效地提高系统的性能和可扩展性。

3.1 跳表

跳表是一种基于有序链表的数据结构，支持快速插入、删除、查找操作，其时间复杂度为O(log n)，比普通链表的O(n)更高效。

https://cmps-people.ok.ubc.ca/ylucet/DS/SkipList.html

图一

图二

图三

跳表的特性有这么几点：

- 一个跳表结构由很多层数据结构组成。
- 每一层都是一个有序的链表，默认是升序。也可以自定义排序方法。
- 最底层链表（图中所示Level1）包含了所有的元素。
- 如果每一个元素出现在LevelN的链表中（N>1），那么这个元素必定在下层链表出现。
- 每一个节点都包含了两个指针，一个指向同一级链表中的下一个元素，一个指向下一层级别链表中的相同值元素。

跳表的查找

跳表的插入

跳表插入数据的流程如下：

1. 找到元素适合的插入层级K，这里的K采用随机的方式。若K大于跳表的总层级，那么开辟新的一层，否则在对应的层级插入。
2. 申请新的节点。
3. 调整对应的指针。

假设我要插入元素13，原有的层级是3级，假设K=4：

倘若K=2：

3.2 ConcurrentSkipListMap使用

基本用法

```
1 public class ConcurrentSkipListMapDemo {
```

```
    public static void main(String[] args) {
        ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();

        // 添加元素
        map.put(1, "a");
        map.put(3, "c");
        map.put(2, "b");
        map.put(4, "d");

        // 获取元素
        String value1 = map.get(2);
        System.out.println(value1); // 输出: b

        // 遍历元素
        for (Integer key : map.keySet()) {
            String value = map.get(key);
            System.out.println(key + " : " + value);
        }

        // 删除元素
        String value2 = map.remove(3);
        System.out.println(value2); // 输出: c
    }
}
```

4. 电商场景中并发容器的选择

案例一：电商网站中记录一次活动下各个商品售卖的数量。

场景分析：需要频繁按商品id做get和set，但是商品id（key）的数量相对稳定不会频繁增删

初级方案：选用HashMap，key为商品id，value为商品购买的次数。每次下单取出次数，增加后再写入

问题：HashMap线程不安全！在多次商品id写入后，如果发生扩容，在JDK1.7 之前，在并发场景下 HashMap 会出现死循环，从而导致CPU 使用率居高不下。JDK1.8 中修复了HashMap 扩容导致的死循环问题，但在高并发场景下，依然会有数据丢失以及不准确的情况出现。

选型：Hashtable 不推荐，锁太重，选ConcurrentHashMap 确保高并发下多线程的安全性

案例二：在一次活动下，为每个用户记录浏览商品的历史和次数。

场景分析：每个用户各自浏览的商品量级非常大，并且每次访问都要更新次数，频繁读写

初级方案：为确保线程安全，采用上面的思路，ConcurrentHashMap
问题：ConcurrentHashMap 内部机制在数据量大时，会把链表转换为红黑树。而红黑树在高并发情况下，删除和插入过程中有个平衡的过程，会牵涉到大量节点，因此竞争锁资源的代价相对比较高
选型：用跳表，ConcurrentSkipListMap将key值分层，逐个切段，增删效率高于ConcurrentHashMap
结论：如果对数据有强一致要求，则需使用Hashtable；在大部分场景通常都是弱一致性的情况下，使用ConcurrentHashMap 即可；如果数据量级很高，且存在大量增删改操作，则可以考虑使用ConcurrentSkipListMap。

案例三：在活动中，创建一个用户列表，记录冻结的用户。一旦冻结，不允许再下单抢购，但是可以浏览。
场景分析：违规被冻结的用户不会太多，但是绝大多数非冻结用户每次抢单都要去查一下这个列表。低频写，高频读。
初级方案：ArrayList记录要冻结的用户id
问题：ArrayList对冻结用户id的插入和读取操作在高并发时，线程不安全。Vector可以做到线程安全，但并发性能差，锁太重。
选型：综合业务场景，选CopyOnWriteArrayList，会占空间，但是也仅仅发生在添加新冻结用户的时候。绝大多数的访问在非冻结用户的读取和比对上，不会阻塞。