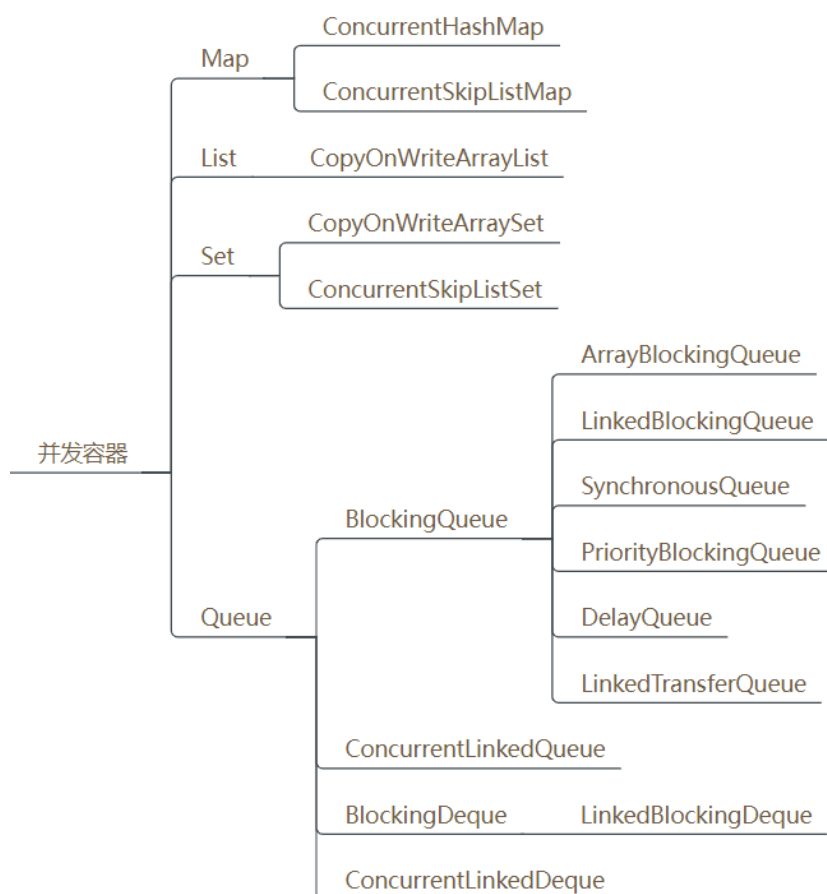


1. JUC包下的并发容器

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

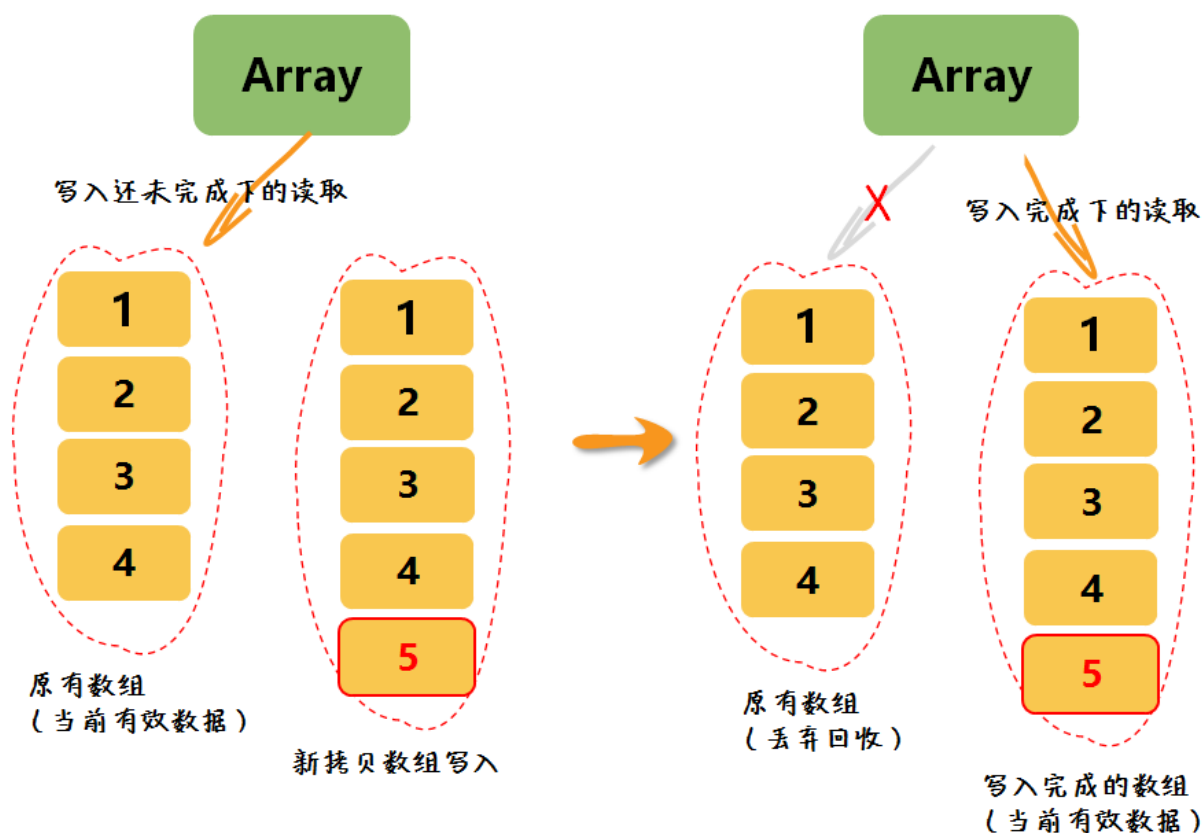
对应的非并发容器：TreeMap

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。

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。它通过在修改操作时创建底层数组的副本来实现线程安全，从而保证了并发访问的一致性。



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。

```
1 // 创建一个 CopyOnWriteArrayList 对象
2 CopyOnWriteArrayList phaser = new CopyOnWriteArrayList();
3 // 新增
4 copyOnWriteArrayList.add(1);
5 // 设置（指定下标）
6 copyOnWriteArrayList.set(0, 2);
7 // 获取（查询）
8 copyOnWriteArrayList.get(0);
9 // 删除
10 copyOnWriteArrayList.remove(0);
11 // 清空
12 copyOnWriteArrayList.clear();
13 // 是否为空
14 copyOnWriteArrayList.isEmpty();
15 // 是否包含
16 copyOnWriteArrayList.contains(1);
17 // 获取元素个数
18 copyOnWriteArrayList.size();
19
```

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

```
1 public class CopyOnWriteArrayListDemo {
2
3     private static CopyOnWriteArrayList<String> copyOnWriteArrayList = new
        CopyOnWriteArrayList<>();
4     // 模拟初始化的黑名单数据
5     static {
6         copyOnWriteArrayList.add("ipAddr0");
7         copyOnWriteArrayList.add("ipAddr1");
8         copyOnWriteArrayList.add("ipAddr2");
9     }
10
11     public static void main(String[] args) throws InterruptedException {
12         Runnable task = new Runnable() {
13             public void run() {
14                 // 模拟接入用时
15                 try {
16                     Thread.sleep(new Random().nextInt(5000));
17                 } catch (Exception e) {}
18
19                 String currentIP = "ipAddr" + new Random().nextInt(6);
20                 if (copyOnWriteArrayList.contains(currentIP)) {
21                     System.out.println(Thread.currentThread().getName() + " IP " +
currentIP + "命中黑名单, 拒绝接入处理");
22                     return;
23                 }
24                 System.out.println(Thread.currentThread().getName() + " IP " +
currentIP + "接入处理...");
25             }
26         };
27         new Thread(task, "请求1").start();
28         new Thread(task, "请求2").start();
29         new Thread(task, "请求3").start();
30
31         new Thread(new Runnable() {
32             public void run() {
33                 // 模拟用时
34                 try {
35                     Thread.sleep(new Random().nextInt(2000));
36                 } catch (Exception e) {}
```

```

37
38         String newBlackIP = "ipAddr3";
39         copyOnWriteArrayList.add(newBlackIP);
40         System.out.println(Thread.currentThread().getName() + " 添加了新的非法
    IP " + newBlackIP);
41     }
42     }, "IP黑名单更新").start();
43
44     Thread.sleep(1000000);
45 }
46 }

```

2.3 原理

很多时候，我们的系统应对的都是读多写少的并发场景。CopyOnWriteArrayList容器允许并发读，读操作是无锁的，性能较高。至于写操作，比如向容器中添加一个元素，则首先将当前容器复制一份，然后在新副本上执行写操作，结束之后再将原容器的引用指向新容器。

- 线程安全的，多线程环境下可以直接使用，无需加锁；
- 通过锁 + 数组拷贝 + volatile 关键字保证了线程安全；
- 每次数组操作，都会把数组拷贝一份出来，在新数组上进行操作，操作成功之后再赋值回去。

从整体架构上来说，CopyOnWriteArrayList 数据结构和 ArrayList 是一致的，底层是个数组，只不过 CopyOnWriteArrayList 在对数组进行操作的时候，基本会分四步走：

- 加锁；
- 从原数组中拷贝出新数组；
- 在新数组上进行操作，并把新数组赋值给数组容器；
- 解锁

除了加锁之外，CopyOnWriteArrayList 的底层数组还被 volatile 关键字修饰，意思是一旦数组被修改，其它线程立马能够感知到，代码如下：

```

1 private transient volatile Object[] array;

```

整体上来说，CopyOnWriteArrayList 就是利用锁 + 数组拷贝 + volatile 关键字保证了 List 的线程安全。

优点

读操作（不加锁）性能很高，因为无需任何同步措施，比较适用于读多写少的并发场景。Java的list在遍历时，若中途有别的线程对list容器进行修改，则会抛ConcurrentModificationException异常。而CopyOnWriteArrayList由于其"读写分离"的思想，遍历和修改操作分别作用在不同的list容器，所以在使用迭代器进行遍历时候，也就不会抛出ConcurrentModificationException异常了。

缺点

- 内存占用问题，毕竟每次执行写操作都要将原容器拷贝一份。数据量大时，对内存压力较大，可能会引起频繁GC；
- 无法保证实时性，因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意：在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）

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


```

1 // 创建一个 ConcurrentHashMap 对象
2 ConcurrentHashMap<Object, Object> concurrentHashMap = new ConcurrentHashMap<>();
3 // 添加键值对
4 concurrentHashMap.put("key", "value");
5 // 添加一批键值对
6 concurrentHashMap.putAll(new HashMap());
7 // 使用指定的键获取值
8 concurrentHashMap.get("key");
9 // 判定是否为空
10 concurrentHashMap.isEmpty();
11 // 获取已经添加的键值对个数
12 concurrentHashMap.size();
13 // 获取已经添加的所有键的集合
14 concurrentHashMap.keys();
15 // 获取已经添加的所有值的集合
16 concurrentHashMap.values();
17 // 清空
18 concurrentHashMap.clear();
19

```

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。

4. `computeIfAbsent(key, Function)`

如果存在则返回key的值。如果不存在，则Function返回值作为key的值

5. `merge(key, value, BiFunction)`

不存在指定的key时，将value设置为key的值。当key存在值时，执行BiFunction接收oldKey和value，返回结果设置为key的值。

统计文件中英文字母出现的总次数

- 将26个英文字母分别循环200次，每个字母作为一个单词，一共有5200个单词。
- 每个单词中间用"\n"分隔，乱序存入26个文件中

- 生成26个线程对26个文件中的单词进行计数，存入map中

1) 生成测试文件

```
1  /**
2   * 生成测试文件
3   * @throws IOException
4   */
5  public void produceData() throws IOException {
6      //定义26个字母的字符串
7      String data="abcdefghijklmnopqrstuvwxyz";
8      List<String> list=new ArrayList<>();
9      //循环遍历26个字母，每个字母循环200次，最后将5200个字母放入集合
10     for (int i = 0; i < data.length(); i++) {
11         for (int j = 0; j < 200; j++) {
12             list.add(String.valueOf(data.charAt(i)));
13         }
14     }
15     //将集合打乱
16     Collections.shuffle(list);
17     //遍历26次。每次取出集合中的200个元素加上“换行符”放入文件中
18     for (int i = 0; i < 26; i++) {
19         try(FileWriter fw=new FileWriter((i+1)+".txt")){
20             fw.write(list.subList(i*200,
21 (i+1)*200).stream().collect(Collectors.joining("\n")));
22         }
23     }
```

2) 读取文件

```
1  /**
2   * 定义读文件的方法
3   */
4  private static void read(List list, int i) {
5      //创建输入缓冲字符流
6      try (BufferedReader bf = new BufferedReader(new FileReader((i + 1) + ".txt"))) {
7          String data;
8          //读取每行数据，判断是否为空
9          while ((data = bf.readLine()) != null) {
10              //将字母加入到集合中
11              list.add(data);
12          }
13      } catch (FileNotFoundException e) {
14          e.printStackTrace();
15      } catch (IOException e) {
16          e.printStackTrace();
17      }
18  }
19
```

3) 生成线程：操作每个文件对应的list，存放线程共享的map

```

1  /**
2   * 定义26个线程读26个文件并将结果放入map。map由函数式接口作为参数提供，放入map由Consumer函数
   式接口处理。
3   *
4   * @param supplier 提供者：提供map集合存放单词计数
5   * @param consumer 消费者：对list（第二个参数）进行计数并存入map（第一个参数）中
6   */
7  private static <T> void deal(Supplier<Map<String, T>> supplier, BiConsumer<Map<String,
   T>, List<String>> consumer) {
8      //获得map集合，用于存放单词计数
9      Map<String, T> map = supplier.get();
10     //利用闭锁保证26个线程都执行完任务
11     CountdownLatch count = new CountdownLatch(26);
12     //循环创建26个线程，读取26个文件的内容，并进行计数操作
13     for (int i = 0; i < 26; i++) {
14         int j = i;
15         new Thread(() -> {
16             List<String> list = new ArrayList();
17             //读取文件
18             read(list, j);
19             consumer.accept(map, list);
20
21             count.countDown();
22         }).start();
23     }
24     try {
25         count.await();
26     } catch (InterruptedException e) {
27         e.printStackTrace();
28     }
29     System.out.println(map);
30 }

```

你要做的是实现两个参数

- 一是提供一个 map 集合，用来存放每个单词的计数结果，key 为单词，value 为计数
- 二是提供一组操作，保证计数的安全性，会传递 map 集合以及 单词 List

正确结果输出应该是每个单词出现 200 次

```
1 {a=200, b=200, c=200, d=200, e=200, f=200, g=200, h=200, i=200, j=200, k=200, l=200,
  m=200, n=200, o=200, p=200, q=200, r=200, s=200, t=200, u=200, v=200, w=200, x=200,
  y=200, z=200}
```

4) 测试代码

```
1
2 // 换成ConcurrentHashMap可以吗?
3 deal(() -> new HashMap<String, Integer>(), (map, words) -> {
4     for (String word : words) {
5         Integer counter = map.get(word);
6         int newValue = counter == null ? 1 : counter + 1;
7         map.put(word, newValue);
8     }
9 });
10
11 //正确的实现1
12 deal(() -> new ConcurrentHashMap<String, LongAdder>(), (map, list) -> {
13     //遍历集合内容
14     list.forEach(str -> {
15         //单词数累加: map中没有str的key则new LongAdder, 有则进行加1
16         map.computeIfAbsent(str, (key) -> new LongAdder()).increment();
17     });
18
19 });
20 //正确的实现2
21 deal(() -> new ConcurrentHashMap<String, Integer>(), (map, list) -> {
22     //遍历集合内容
23     list.forEach(str -> {
24         //单词数累加: map中没有str的key则set(str,1), 有则set(str,Integer.sum(oldvalue,1))
25         map.merge(str, 1, Integer::sum);
26     });
27
28 });
```

2.3 数据结构

HashTable的数据结构

JDK1.7 中的ConcurrentHashMap

在jdk1.7中，结构是用Segments数组 + HashEntry数组 + 链表实现的（写分散的思想）。

ConcurrentHashMap内部维护了一个Segment数组。每个Segment继承自ReentrantLock并且它内部本质上是一个Hash表。这样做的好处是能够减小锁的粒度，提高并发访问的效率。默认Segment 数量为16，可以通过构造函数来修改默认值。当需要put或get一个元素时，线程首先通过hash定位到具体的Segment，然后在对应的Segment上进行锁定操作。

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

```
1 #树化阈值为8
2 static final int TREEIFY_THRESHOLD = 8;
3 #最小树化容量值为64
4 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 {
2     public static void main(String[] args) {
3         ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();
4
5         // 添加元素
6         map.put(1, "a");
7         map.put(3, "c");
8         map.put(2, "b");
9         map.put(4, "d");
10
11        // 获取元素
12        String value1 = map.get(2);
13        System.out.println(value1); // 输出: b
14
15        // 遍历元素
16        for (Integer key : map.keySet()) {
17            String value = map.get(key);
18            System.out.println(key + " : " + value);
19        }
20
21        // 删除元素
22        String value2 = map.remove(3);
23        System.out.println(value2); // 输出: c
24    }
25 }

```

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。可以使用CopyOnWriteArrayList。

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。