一 ConcurrentMap 概述

1. ConcurrentMap 及其子类是 JDK1.5 提供的一套用于对应高并发的映射机制
2. ConcurrentMap 在并发的时候还能比较好的保证线程安全

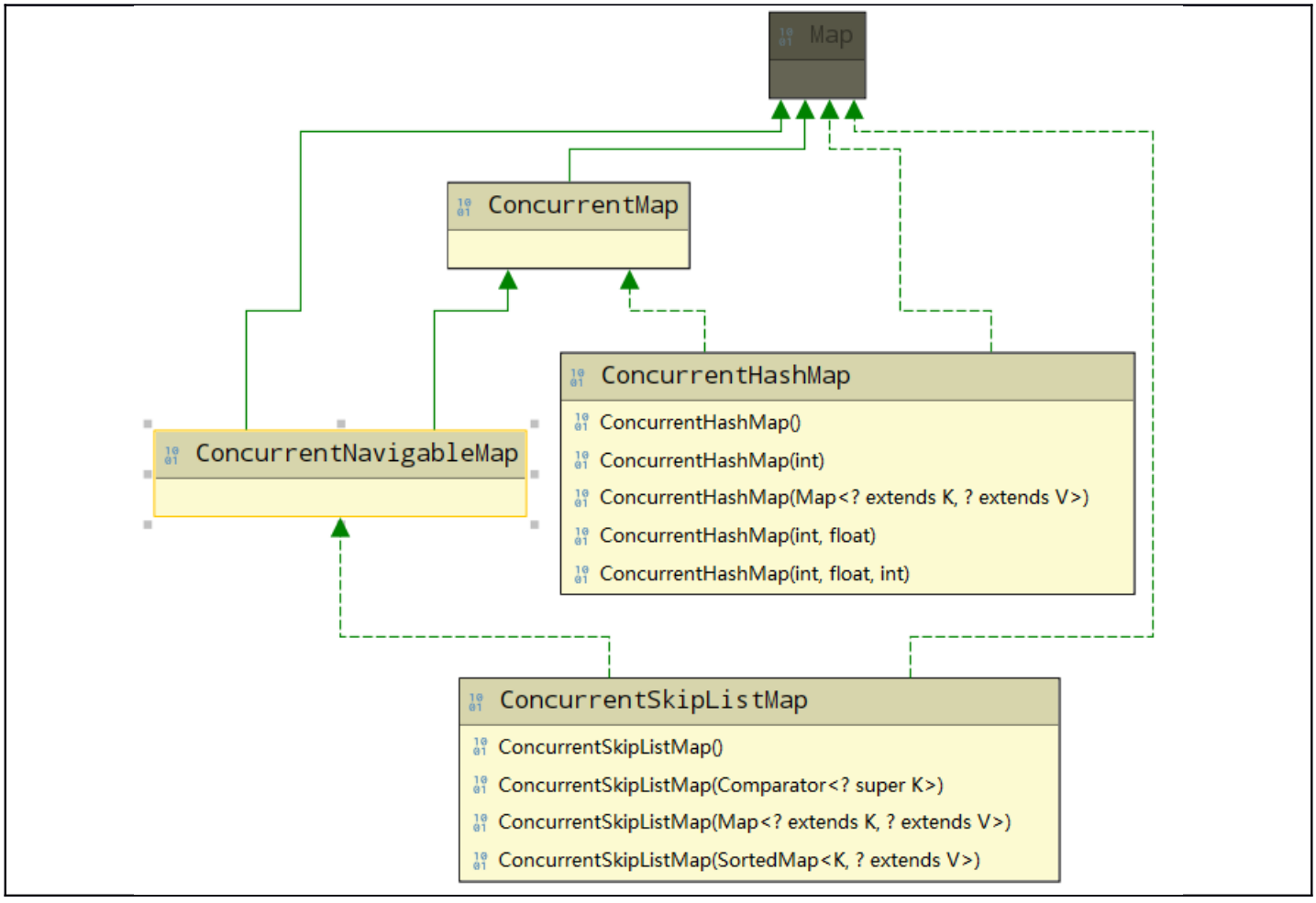
二 ConcurrentMap 继承结构

ConcurrentMap 为接口

ConcurrentHashMap 实现了 ConcurrentMap 接口

ConcurrentNavigableMap 是一个接口，继承了 ConcurrentMap 接口

ConcurrentSkipListMap 实现了 ConcurrentNavigableMap 接口，唯一一个实现类



三 ConcurrentHashMap - 并发哈希映射

1. ConcurrentHashMap 底层是基于数组+链表来存储，数组的默认容量是 16(表示有 16 个桶，桶里为链表)，默认加载因子是 0.75，默认扩容是增加一倍的桶数。
2. ConcurretHashMap 最多允许存在 2^{30} 个桶

```
*/
private static final int MAXIMUM_CAPACITY = 1 << 30;
```

3. 在 ConcurrentHashMap 中，允许指定容量。这个指定的容量经过计算一定是 2^n 的形式

```
// 指定容量 19，实际容量是 32
// 19 -> 16<29<=32 -> 32
// 指定容量 48，实际容量是 128
// 48 -> 64<73<=128 -> 128
// 70 -> 64<106<=128 -> 128
// n -> 2^x<(n + n/2 + 1)<=2^(x+1) -> 2^(x+1)
ConcurrentHashMap<String, String> map =
        new ConcurrentHashMap<>(48);
```

```
*/
public ConcurrentHashMap(int initialCapacity) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException();
    int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
               MAXIMUM_CAPACITY :
               tableSizeFor( c: initialCapacity + (initialCapacity >>> 1) + 1));
    this.sizeCtl = cap;
}
```

```
private static final int tableSizeFor(int c) {
    int n = c - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
```

4. 在 JDK1.8 中，为了提高效率引入了红黑树机制。当桶中的元素个数超过 8 个的时候，这个桶中的链表扭转成一棵红黑树；如果红黑树的节点个数不足 7 个的时候，将红黑树扭转回链表。扭转成红黑树的前提：桶的数量>=64 且元素超过 8 个，那么也就意味着如果桶数<64，那么桶中的元素个数无论是多少都不会变成红黑树

1 红黑树：

ppt

1.a. 本质上是一棵自平衡二叉查找树

1.b. 二叉查找树(BST)：在二叉树的基础上加入了排序机制，使得左子树一定是小于根，右子树一定是大于根

1.c. 特征：

1.a.i. 所有节点非红即黑

1.a.ii. 根节点为黑

1.a.iii. 红节点的子节点一定是黑的

1.a.iv. 最底下的叶子节点是黑色的空节点

1.a.v. 从根节点到叶子节点经过的路径中的黑色节点个数一致，即黑节点的高度是一致的

1.a.vi. 新添节点的颜色一定是红色的

1.b. 红黑树在构建过程中，每次添加一个新的节点都需要考虑是否需要修正：涂色、左旋、右旋

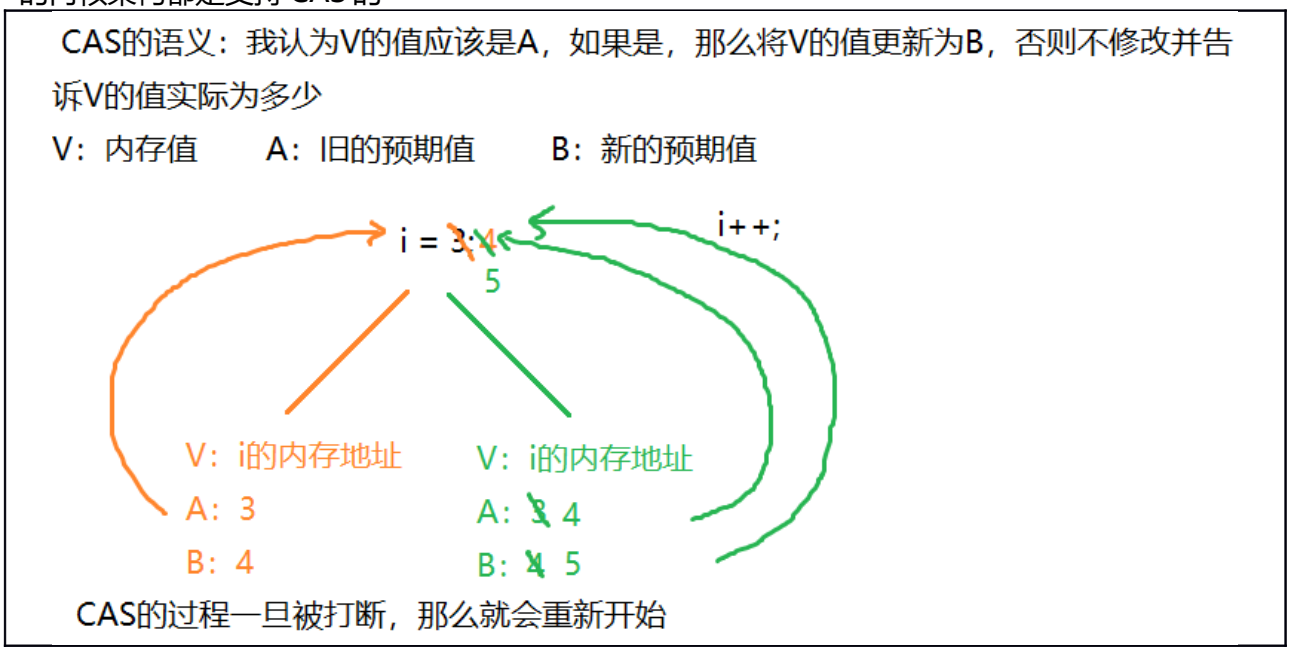
1.c. 红黑树的时间复杂度是 O(logn)

2. ConcurrentHashMap 是一个**异步线程安全**的映射：采用了分段/桶锁机制（对桶加锁，而 Hashtable 是对整个对象加锁）。在后续版本中，ConcurrentHashMap 在分段锁基础上引入了读写锁机制：

1.a. 读锁：允许多个线程同时读，不允许线程写

1.b. 写锁：只允许一个线程写，不允许线程读

2. ConcurrentHashMap 利用锁保证线程安全，但是在使用锁的时候，造成 CPU 的资源浪费(例如线程调度，线程上下文切换等)。在 JDK1.8 中，考虑到锁所带来的开销，引入了无锁算法 CAS(Compare And Swap，比较和交换)。因为 CAS 涉及到线程的重新调度问题，所以需要结合具体的 CPU 内核架构来设计，因此 Java 中的 CAS 底层是依靠 C 语言实现的。目前市面上，服务器中流行的内核架构都是支持 CAS 的



四　ConcurrentNavigableMap - 并发导航映射

1. 提供了用于截取子映射的方法
2. ConcurrentNavigableMap 是一个接口，所以使用的是它的实现类 ConcurrentSkipListMap - 并发跳跃表映射 - 底层是基于跳跃表实现的

1　跳跃表：

1.a. 针对有序列表进行操作

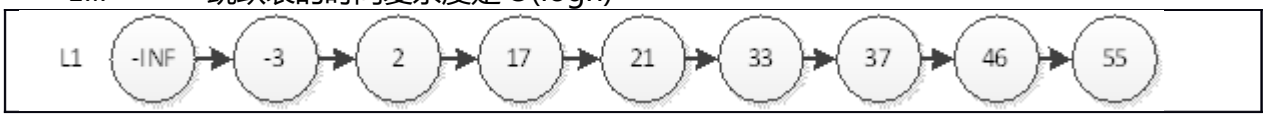
1.b. 适用于查询多而增删少的场景

1.c. 典型的"以空间换时间"的产物

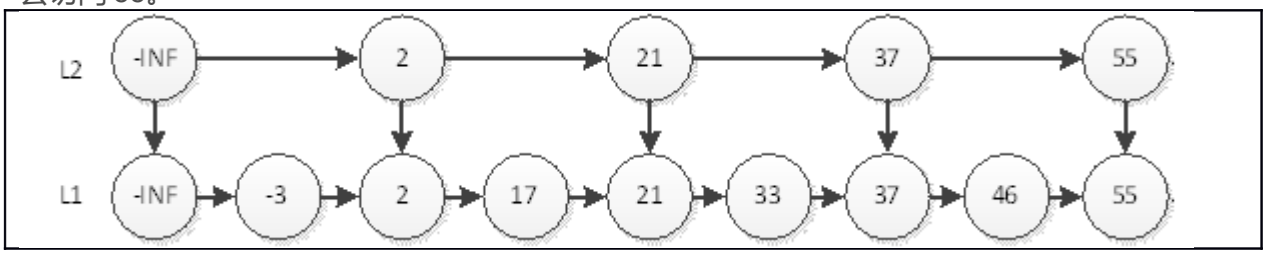
1.d. 跳跃表允许进行多层提取，最上层跳跃表的元素个数不能少于 2 个

1.e. 如果在跳跃表中新添了元素，那么新添的元素是否要提取到上层的跳跃表中遵循"抛硬币"原则

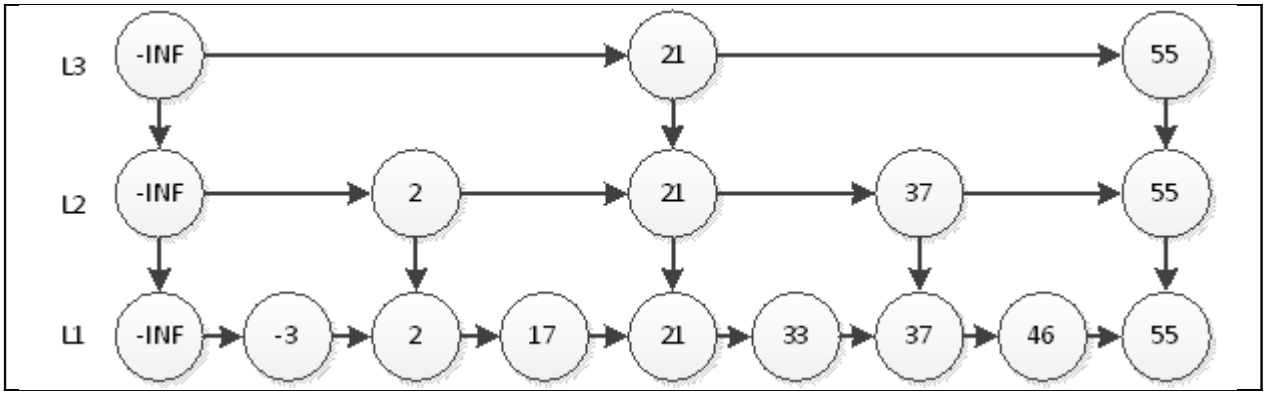
1.f. 跳跃表的时间复杂度是 O(logn)



如上图，我们要查询元素为 55 的结点，必须从头结点，循环遍历到最后一个节点，不算-INF(负无穷)一共查询 8 次。那么用什么办法能够用更少的次数访问 55 呢？最直观的，当然是新开辟一条捷径去访问 55。



如上图，我们要查询元素为 55 的结点，只需要在 L2 层查找 4 次即可。在这个结构中，查询结点为 46 的元素将耗费最多的查询次数 5 次。即先在 L2 查询 46，查询 4 次后找到元素 55，因为链表是有序的，46 一定在 55 的左边，所以 L2 层没有元素 46。然后我们退回到元素 37，到它的下一层即 L1 层继续搜索 46。非常幸运，我们只需要再查询 1 次就能找到 46。这样一共耗费 5 次查询。
那么，如何才能更快的搜寻 55 呢？有了上面的经验，我们就很容易想到，再开辟一条捷径。



2 测试

```
@Test
public void test03(){
    // 提供了用于截取子映射的方法
    ConcurrentNavigableMap<String, Integer> map =
            new ConcurrentSkipListMap<>();
    map.put("d", 5);
    map.put("e", 6);
    map.put("b", 3);
    map.put("a", 4);
    map.put("c", 6);
    //已排序
    System.out.println(map);//{a=4, b=3, c=6, d=5, e=6}
    // 从头开始截取到指定的位置
    System.out.println(map.headMap("e"));//{a=4, b=3, c=6, d=5}
    // 从指定位置开始截取到末尾
    System.out.println(map.tailMap("e"));//{e=6}
    // 截取指定范围内的元素 包头不包尾
    System.out.println(map.subMap("e", "u"));//{e=6}
}
```