1. ConcurrentHashMap
2. 概述

HashMap是线程不安全的，put操作可能会产生死循环，在JDK1.8中更换了数据插入的顺序，解决了这个问题。但是线程安全问题依然未解决，因此Hashtable和Collections.synchronizedMap(hashmap)两种解决方案被提出，这两种方案都是通过对读写加锁，独占式地解决线程安全问题，吞吐量很低，效率低下。

由此，提出了高性能的HashMap：ConcurrentHashMap，JDK1.8之前，它采用分段锁的方式，JDK1.8之后，采用CAS+Synchronized来保证并发的安全问题，底层采用数组+链表+红黑树的存储结构。

1. JDK8前的HashMap
2. 结构

HashMap是不支持多线程的，内部结构是一个数组，数组中的每个元素是一个单向链表，链表中的每一项都是一个嵌套类Entry的实例，Entry包含着四个属性：key, value, hash值和用于单向链表的next。

1. HashMap初始化

public HashMap(int initialCapacity, float loadFactor)

capacity: 数组容量，始终保持为2^n，扩容后数组扩大为两倍；

loadFactor: 负载因子，达到扩容的百分比；

threshold: 扩容的阈值，等于capacity \* loadFactor

1. put过程
2. 数组初始化，在第一个元素插入HashMap时初始化数组，确定数组大小，计算数组扩容的阈值；
3. 通过key计算hash值，根据hash值计算应该放在哪个链表中；
4. 找到链表的表头，判断key是否重复，如果没有重复就将新值放到链表的表尾，在多线程中，这种操作会造成死循环，因为表尾可能直接指向了表头，就会一直循环而找不到表尾，JDK1,8中已经解决；
5. 在插入新值时，如果当前的size已经达到了阈值，并且要插入的数组位置上已经有元素了，那么就会触发扩容，用一个2倍容量的数组替换之前的数组。
6. get过程
7. 根据key计算hash值；
8. 根据hash值找到对应的数组下标；
9. 遍历该数组位置的链表，直到直到相等的key；
10. JDK8前的ConcurrentHashMap
11. 概述

ConcurrentHashMap支持并发操作，由多个Segment组成，Segment通过继承ReentrantLock来进行加锁，每次需要加锁的操作锁住的就是一个Segment，这样其他的Segment不会能够被操作，既保证了线程安全，也保证了性能的高效。每个Segment中类似HashMap的结构，是一个数组，每个数组的元素又是一个单向链表。

Segment数组无法扩容，而每个Segment中的数组是可以扩容的。

1. 初始化

public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrentLevel)

initialCapacity: 初始容量，需要平均分给每个Segment;

loadFactor: 负载因子，给每个Segment中的数组扩容使用；

concurrentcyLevel: 并发数，默认是16，表示有16个Segment，由于Segment数组是不能扩容的，因此有多少个Segment就只能支持多少个线程并发写，这是它的局限性；

1. put过程
2. 根据hash值知道相应的Segment，然后对Segment内部进行put操作；
3. get过程
4. 计算hash值，找到segment数组中的具体位置；
5. 进入Segment后，继续找数组中具体值的位置；
6. 顺着链表查找；
7. JDK8的HashMap
8. 概述

JDK8修改了HashMap，新增了红黑树的结构，因此它是由数组+链表+红黑树组成的。JDK8之前的HashMap中，查找的时候，根据Hash值定位到数组的具体下标，然后顺着链表查找具体的值，时间复杂度取决于链表的长度。JDK8为了降低这部分的开销，当链表中的元素超过了8个之后，将链表转换为红黑树。

JDK7中使用Entry代表每个HashMap中的数据节点，JDK8中使用Node，都是包含key, value, hash和next四个属性。不过Node只能用于链表，红黑树的情况需要使用TreeNode。我们可以根据数组元素中，第一个节点数据类型是Node还是TreeNode来判断该位置是链表还是红黑树。

1. put过程

与JDK7类似，区别是在需要扩容时，JDK8先插值再扩容，而JDK7是先扩容后插值的。同时，JDK8还使用了红黑树。

1. get过程分析

在遍历数组时，需要判断链表的头结点是TreeNode还是Node，以此来选择是通过红黑树的方式还是链表的方式查找。

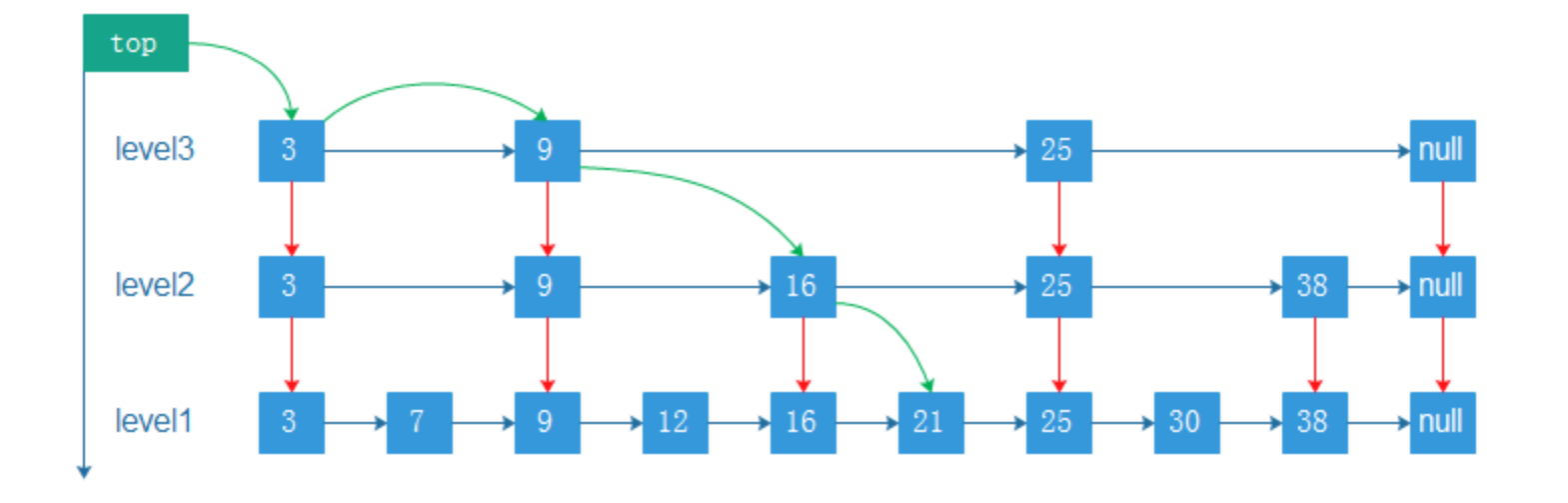
1. JDK8的ConcurrentHashMap

JDK8中采用CAS+Synchronized来保证并发更新的安全，底层采用数组+链表+红黑树的存储结构。不像JDK7采用的分段锁结构，它直接使用一个数组，数组中的每个元素是链表或者红黑树，这解决了并发线程数的限制问题。

1. 与Hashtable的对比
2. Hashtable对get, put, removie都使用同步操作，影响了性能和吞吐量；concurrentHashMap只对put, remove 使用了同步操作，而get操作并不影响；
3. Hashtable在遍历时，如果有remove, put等操作，会抛出ConcurrentModificationException异常。而ConcurrentHashMap没有这方面的问题。
4. ConcurrentSkipListMap
5. 概述

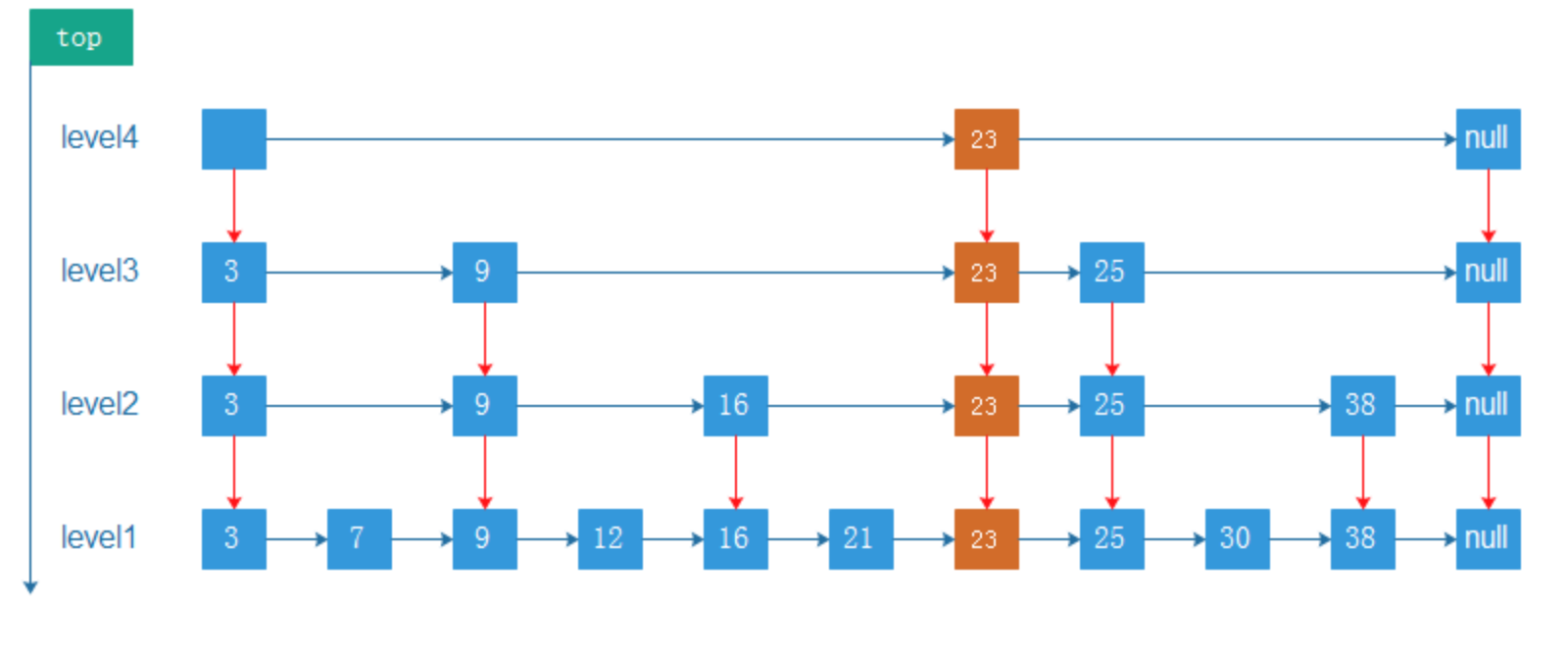
ConcurrentSkipListMap使用的是SkipList数据结构，该结构的效率不低于红黑树。该结构称为跳表，它可以代替平衡树，其中的数据元素按照key值升序。且SkipList中的数据是分布在多层链表中的，以0-1随机数决定一个数据是否需要在不同的层次中攀升，通过空间换取时间的策略，在每个节点中添加了向前的指针，在插入，查找，删除时可以忽略一些不可能涉及到的节点，从而提高了效率。

将不同的节点向上层提取为索引，每次查找时从上层开始的索引开始查找，如果没有找到，则向下一层查找，这样能够减少比较的次数。



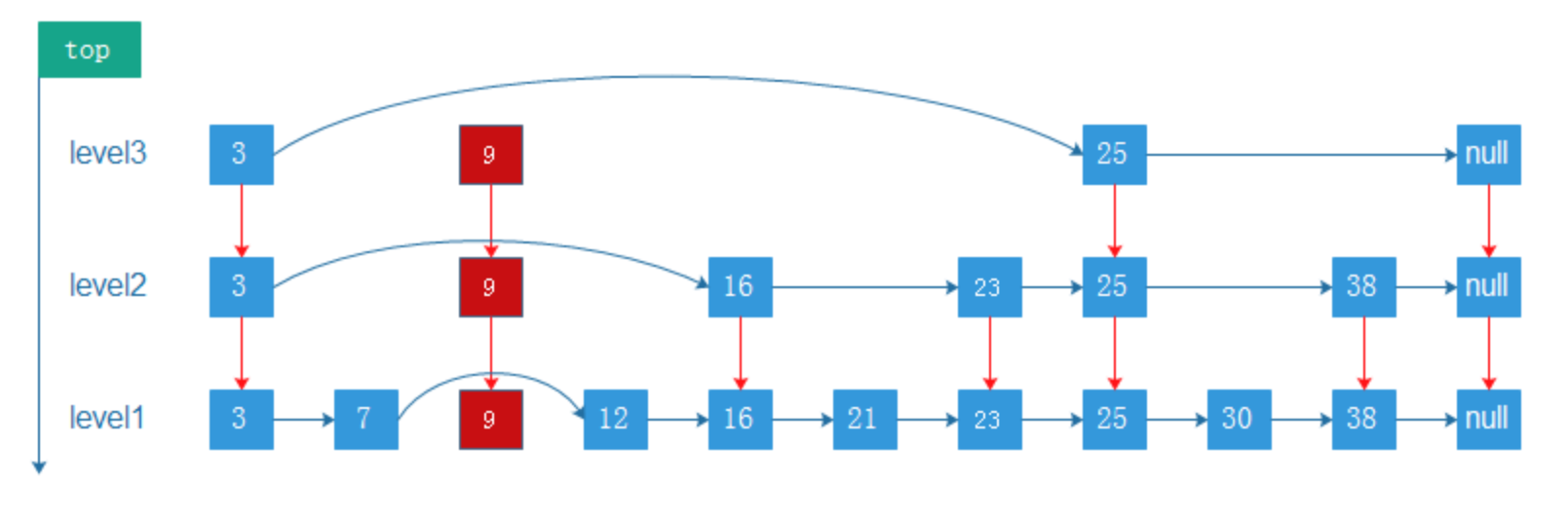
1. 插入操作

查找合适的位置，在确定新节点要占据的层次时，是通过随机的方式决定的，如果产生的层次大于SkipList的层次，则重新申请新的层，否则插入指定的层；申请新的节点，调整其他的节点的指针。注意，在插入的层次以下的那些层次中都需要插入，而不仅仅是插入一层。



1. 删除操作

找到节点，删除节点，调整指针，同样地，将该节点的所有层次都删除。



1. 总结
2. ConcurrentHashMap只是并发效率高的Map，用来替换其他线程安全你的容器，如Hashtable和Collections.synchronizedMap。但是，它的线程安全只能保证自身的数据不被破坏，以及在多个线程之中是可见的，对于多个线程的并发操作，无法保证若多个线程都对该容器有操作时的线程安全问题。即如果多个线程都对ConcurrentHashMap有操作，则需要使用锁来锁住某个共享的资源，保证原子性。
3. ConcurrentHashMap在获取对象时是没有使用锁的；
4. ConcurrentHashMap允许边更新，边遍历，遍历的数据会随着remove，put操作产生变化。