# 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

# 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，唯一个实现类

|  |
| --- |
|  |

# 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230个桶

|  |
| --- |
|  |

* 1. 在ConcurrentHashMap中，允许指定容量。这个指定的容量经过计算一定是2n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); |
|  |
|  |

* 1. 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

### 红黑树：

ppt



* + 1. 本质上是一棵自平衡二叉查找树
    2. 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
    3. 特征：
       1. 所有节点非红即黑
       2. 根节点为黑
       3. 红节点的子节点一定是黑的
       4. 最底下的叶子节点是黑色的空节点
       5. 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
       6. 新添节点的颜色一定是红色的
    4. 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
    5. 红黑树的时间复杂度是O(logn)
  1. 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. 读锁：允许多个线程同时读，不允许线程写
     2. 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

|  |
| --- |
| 计算机生成了可选文字: CAS 的 语文： 我 认 为 V 的 值 应 该 是 A 如 果 是 ， 那 么 将 V 的 值 更 新 为 B  诉 V 的 值 实 际 为 多 少  否 则 不 修 改 并 告  B ． 新 的 预 期 值  内 存 值  A ： 旧 的 预 期 值  V ： i 的 内 存 地 址  5  V ： i 的 内 存 地 址  A ： \ 4  CAS 的 过 程 一 旦 被 打 断 ， 月 仫 就 会 重 新 开 始 |

# 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

### 跳跃表：

* + 1. 针对有序列表进行操作
    2. 适用于查询多而增删少的场景
    3. 典型的"以空间换时间"的产物
    4. 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
    5. 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
    6. 跳跃表的时间复杂度是O(logn)

|  |
| --- |
| 这里写图片描述 |

 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。

|  |
| --- |
| 这里写图片描述 |

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。

|  |
| --- |
| 这里写图片描述 |

### 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} } |