# ConcurrentHashMap核心源码

1. **ConcurrentHashMap的前置知识扫盲**

什么时候使用？

请求数据的时候直接从数据库拿，mysql的性能无法达到我的预期，不管去redis还是mysql都是一次io。所以要把缓存直接存在内存中

HashMap，线程不安全，单线程内部使用

ConcurrentHashMap 多个线程访问，多个线程去读写，全局

如果在多个service下，nginx配置负载均衡，会有多个ConcurrentHashMap，Zookeeper监听通知，通知znode 去同步数据，延迟小

或者可以使用rabbitMq，广播机制，延迟大

CAP 保证AP，在某一个时刻数据会有不一致，

我redis性能不够

ConcurrentHashMap 设置生存时间，0.5s ，用减轻redis压力

何时使用：springbean，nacos存服务地址、名称

链表查询的时间复杂度是O(N), 红黑树的查询的时间复杂度是O(logN)

存储结构JDK1.8 数组加链表加红黑树，引入的目的是解决链表过长导致查询效率降低的问题

Node： 存储前会把key 和value封装成一个Node，int hash; key; value;next 4个属性

根据key的值来决定存放数组哪个位置

1. **ConcurrentHashMap的红黑树转换时机**

链表转红黑树，必须满足 数组长度大于等于64，切链表的长度大于等于8

这么设计的原因是因为红黑树的出现本身就是为了提高查询效率，如果数组长度太短，同时出现了大量数据存储到一个位置的情况，此时优先扩容，让数据尽可能的分散到数组上

源码中的数组长度64是用final修饰的，不让修改

为什么链表长度是到8才转红黑树？

无非就是红黑树出现的概率不同，7的话出现的概率会高一些，9的话概率低一些，8的话概率是0.00000006

当红黑树的节点变少时，元素个数达到6个时，会将红黑树转回链表

1. **ConcurrentHashMap的扩容时机**

扩容就是构建一个原来二倍的数组，将老数组中的数据迁移到新数组。

1. 如果是链表长度到达8，且数组小于64，此时会优先扩容。
2. 当元素个数超过的ConcurrentHashMap的阈值时，就会出发扩容。 阈值 = 数组长度 \* 0.75 这个0.75是负载因子，HashMap是可以更改。
3. **ConcurrentHashMap的DCL操作**

什么是DCL操作？

一般情况下，第一次接触DCL应当是在写单例模式的懒汉机制时碰到的。

DCL操作就是Double Check Lock（双重检查锁）

在每个ConcurrentHashMap对象中，数组是全局唯一的。在new ConcurrentHashMap之后，数组并没有构建出来，而是在put添加数据时，才会将数组初始化

If (xxx == null) { check

If (U.compareAndSwapInt(this, SIZECTL, sc， -1)) { // 这里的锁是CAS Lock

If(xxx ==null) { check

}

}

}

掌握sizeCtl属性

sizeCtl == -1 代表当前有线程正在初始化数组

private transient volatile int sizeCtl;

初始化之前会先把sizeCtl设置成-1

1. **ConcurrentHashMap的散列算法**

如何基于key确定元素存储的数组索引位置

i = (n - 1) & hash 其中n是数组的长度，hash是key的hashCode返回的的int值

如何尽可能的打散

所谓打算就是减少hash冲突的出现。

源码：(h ^ (h >>> 16)) & HASH\_BITS;

将高低位做了一个亦或运算，然后再来计算存储的索引位置

亦或操作就是相同为0，不同为1

1. **ConcurrentHashMap的并发安全**

针对的是写写的线程安全

ConcurrentHashMap的线程安全有一个简单的发展过程。

从HashTable --> JDK1.7的ConcurrentHashMap --> JDK1.8的ConcurrentHashMap

JDK1,7是用的ReentrantLock来实现的分段锁，也是互斥锁，包装了一层，用的是Segment（默认16个）

JDK1.8是每个数组上都上锁，有两种情况，CAS和synchronized

往数组上放数据的时候，如果数组的位置上的null，基于cas的方式，将数组上的null，替换为存放的Node

将数据挂到链表上或者挂到红黑树的时候，用的是synchronized,锁住数组上的Node对象

1. **ConcurrentHashMap的计数器实现机制**

计数器就是记录当前ConcurrentHashMap中的元素个数

添加元素，计数器+1

删除元素，计数器-1

Java中的JUC包下，提供了一套原子类，实现了++，--时的线程安全，不需要再搞新的，可以直接用AtomicInteger,AtomicLong之类的原子类。

他们的底层都是基于CAS来实现++，--的线程安全。

But，这种Atomic原子类都存在一个问题。

|  |
| --- |
| public final int getAndAddInt(Object o, long offset, int delta) {  int v;  do {  v = getIntVolatile(o, offset);  } while (!weakCompareAndSetInt(o, offset, v, v + delta));  return v;  } |

Automic原子类都是基于do-while循环去对元素进行++或者—操作，如果CAS失败，会一直重复执行。导致CPU一直在执行CAS，但是一直不成功，有点浪费 CPU资源。

所以，ConcurrentHashMap没用使用Atomic原子类，而是采用了LongAdder的实现机制。但是它没有直接引用LongAdder，而是将LongAdder中核心的++，--功能的代码复制了一份，拿来使用

这里的LongAdder有点分段锁的效果

ConcurrentHashMap除了成员变量baseCount之外（它是通过CAS来实现的），还提供了一个CounterCell数组，在CounterCell对象中有volatile修饰的long类型的value存储元素个数。每个线程会根据随机数，随机找数组中的某一个CounterCell，对其value做+1操作。

1. **ConcurrentHashMap的size实现策略**

Size方法返回当前元素的个数

baseCount + 所有的CounterCell里面value的累加和

# HashMap底层原理

1.7的时候数组+链表

1.8数组+链表+红黑树

八大基础数据结构

数组 栈 链表 图 队列 堆 树 散列表

Hashmap是散列表的结构 存储KV的键值对

存储数据 java方式存储一个整体 key value

Class Node {

int hash;

private K key;

private V value;

Node next;

}

红黑树的5个特点

1. 每个节点要么是黑色要么是红色。
2. 根节点是黑色
3. 每个叶子节点是黑色。注意：这里叶子节点，是指为空的叶子节点
4. 如果一个节点是红色的，则它的子节点必须是黑色的。
5. 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点

它是一颗黑节点平衡的平衡二叉树

它是个静态内部类

Transient 修饰node ，它是不会被序列化的属性

Node[] table 初始大小1 << 4 10000 16

Public native int hashCode(); 取值范围 就是int的取值范围

Hashmap 1.7是 key.hashcode % 16 模 余数 就是 0~15之间

取模运算不快，计算机中最快的几种运算 位运算 逻辑运算 与或非移位 是最快的

位运算 没有办法扩容，一旦数组扩容，就会发生问题，效率就会降低