目录

[ConcurrentHashMap 2](#_Toc514678859)

[预备知识 2](#_Toc514678860)

[Hash 2](#_Toc514678861)

[位运算 2](#_Toc514678862)

[1.7中原理和实现 3](#_Toc514678863)

[ConcurrentHashMap中的数据结构 3](#_Toc514678864)

[初始化做了什么事？ 4](#_Toc514678865)

[在get和put操作中，是如何快速定位元素放在哪个位置的？ 5](#_Toc514678866)

[get（）方法 6](#_Toc514678867)

[put()方法 6](#_Toc514678868)

[扩容操作 7](#_Toc514678869)

[size方法 7](#_Toc514678870)

[弱一致性 8](#_Toc514678871)

[1.8 8](#_Toc514678872)

[与1.7相比的重大变化 8](#_Toc514678873)

[主要数据结构和关键变量 8](#_Toc514678874)

[初始化做了什么事？ 8](#_Toc514678875)

[在get和put操作中，是如何快速定位元素放在哪个位置的？ 9](#_Toc514678876)

[get（）方法 9](#_Toc514678877)

[put()方法 9](#_Toc514678878)

[扩容操作 11](#_Toc514678879)

[size方法 11](#_Toc514678880)

[一致性 11](#_Toc514678881)

[更多的并发容器 11](#_Toc514678882)

[ConcurrentSkipListMap 和 ConcurrentSkipListSet 11](#_Toc514678883)

[跳表 11](#_Toc514678884)

[ConcurrentLinkedQueue 11](#_Toc514678885)

[写时复制容器 12](#_Toc514678886)

[阻塞队列 12](#_Toc514678887)

[概念、生产者消费者模式 12](#_Toc514678888)

[生产者和消费者模式 12](#_Toc514678889)

[常用方法 13](#_Toc514678890)

[常用阻塞队列 13](#_Toc514678891)

[阻塞队列的实现原理 14](#_Toc514678892)

并发容器

# ConcurrentHashMap

Hashmap多线程会导致HashMap的Entry链表形成环形数据结构，一旦形成环形数据结构，Entry的next节点永远不为空，就会产生死循环获取Entry。

HashTable使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法，其他线程也访问HashTable的同步方法时，会进入阻塞或轮询状态。如线程1使用put进行元素添加，线程2不但不能使用put方法添加元素，也不能使用get方法来获取元素，所以竞争越激烈效率越低。

putIfAbsent() ：没有这个值则放入map，有这个值则返回key本来对应的值。

## 预备知识

### Hash

散列，哈希：把任意长度的输入通过一种算法（散列），变换成为固定长度的输出，这个输出值就是散列值。属于压缩映射，容易产生哈希冲突。Hash算法有直接取余法等。

产生哈希冲突时解决办法：开放寻址；2、再散列；3、链地址法（相同hash值的元素用链表串起来）。

ConcurrentHashMap在发生hash冲突时采用了链地址法。

md4,md5,sha-hash算法也属于hash算法，又称摘要算法。

### 位运算

**int类型的位**

**高位 低位**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1 | 0 | 1 | 0 | 0 | 0 |

2的0次方 = 1，2的1次方=2…….，以上表格代表数字 （2的5次方+2的3次方）= 40

由上面的表格可以看出，数字类型在数字渐渐变大时，是由低位慢慢向高位扩展的。

Java实际保存int型时 正数 第31位 =0 负数：第31位=1

常用位运算有：

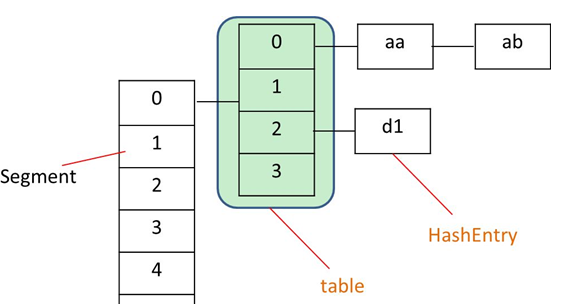
* 位与 & (1&1=1 1&0=0 0&0=0)
* 位或 | (1|1=1 1|0=1 0|0=0)
* 位非 ~ （ ~1=0 ~0=1）
* 位异或 ^ (1^1=0 1^0=1 0^0=0)
* <<有符号左移 >>有符号的右移 >>>无符号右移 例如：8 << 2 = 32 8>>2 = 2
* 取模的操作 a % (Math.pow(2,n)) 等价于 a&( Math.pow(2,n)-1)

**位运算适用**：权限控制，物品的属性非常多时的保存

## 1.7中原理和实现

### ConcurrentHashMap中的数据结构

ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment实际继承自可重入锁（ReentrantLock），在ConcurrentHashMap里扮演锁的角色；HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，每个Segment里包含一个HashEntry数组，我们称之为table，每个HashEntry是一个链表结构的元素。



**面试常问：**

1. **ConcurrentHashMap实现原理是怎么样的或者问ConcurrentHashMap如何在保证高并发下线程安全的同时实现了性能提升？**

答：ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了**锁分离**技术。它使用了多个锁来控制对hash表的不同部分进行的修改。内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的hash table，只要多个修改操作发生在不同的段上，它们就可以并发进行。

### 初始化做了什么事？

初始化有三个参数

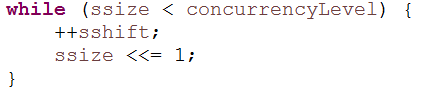
**initialCapacity**：初始容量大小 ，默认16。

**loadFactor,** 扩容因子，默认0.75，当一个Segment存储的元素数量大于initialCapacity\* loadFactor时，该Segment会进行一次扩容。

**concurrencyLevel** 并发度，默认16。并发度可以理解为程序运行时能够同时更新ConccurentHashMap且不产生锁竞争的最大线程数，实际上就是ConcurrentHashMap中的分段锁个数，即Segment[]的数组长度。如果并发度设置的过小，会带来严重的锁竞争问题；如果并发度设置的过大，原本位于同一个Segment内的访问会扩散到不同的Segment中，CPU cache命中率会下降，从而引起程序性能下降。

**构造方法中部分代码解惑：**

1、



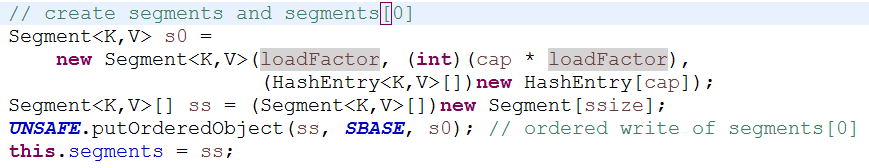
保证Segment数组的大小，一定为2的幂，例如用户设置并发度为17，则实际Segment数组大小则为32

2、



保证每个Segment中tabel数组的大小，一定为2的幂，初始化的三个参数取默认值时，table数组大小为2

3、



初始化Segment数组，并实际只填充Segment数组的第0个元素。

4、

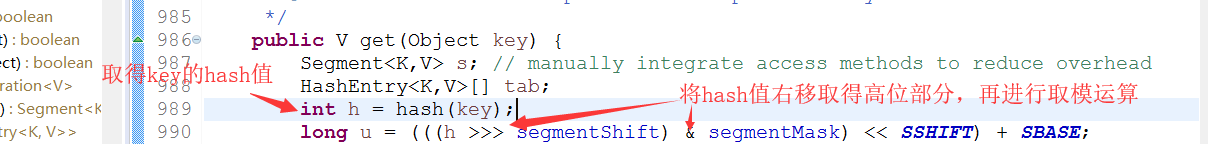


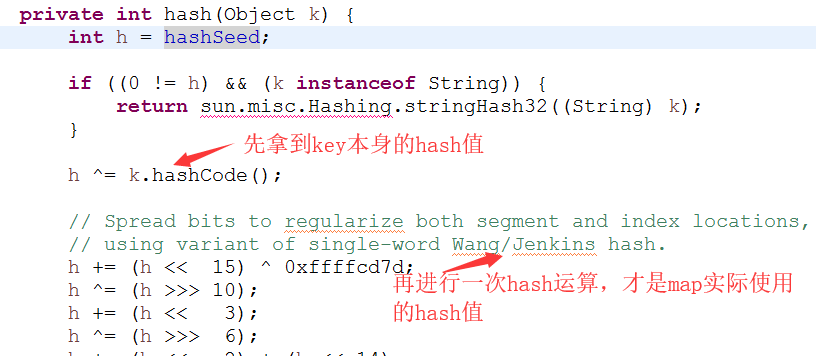
用于定位元素所在segment。segmentShift表示偏移位数，通过前面的int类型的位的描述我们可以得知，int类型的数字在变大的过程中，低位总是比高位先填满的，为保证元素在segment级别分布的尽量均匀，计算元素所在segment时，总是取hash值的高位进行计算。segmentMask作用就是为了利用位运算中取模的操作： a % (Math.pow(2,n)) 等价于 a&( Math.pow(2,n)-1)

### 在get和put操作中，是如何快速定位元素放在哪个位置的？

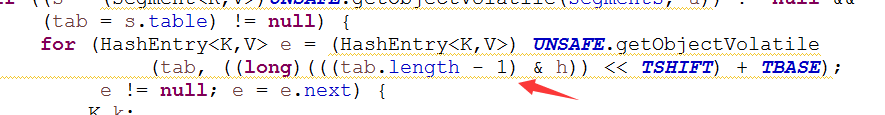
对于某个元素而言，一定是放在某个segment元素的某个table元素中的，所以在定位上，

**定位segment：**取得key的hashcode值进行一次再散列（通过Wang/Jenkins算法），拿到再散列值后，以再散列值的高位进行取模得到当前元素在哪个segment上。





定位table：同样是取得key的再散列值以后，用再散列值的全部和table的长度进行取模，得到当前元素在table的哪个元素上。

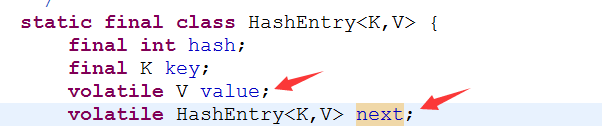


### get（）方法

定位segment和定位table后，依次扫描这个table元素下的的链表，要么找到元素，要么返回null。

**在高并发下的情况下如何保证取得的元素是最新的？**

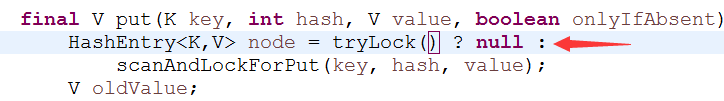
答：用于存储键值对数据的HashEntry，在设计上它的成员变量value等都是volatile类型的，这样就保证别的线程对value值的修改，get方法可以马上看到。



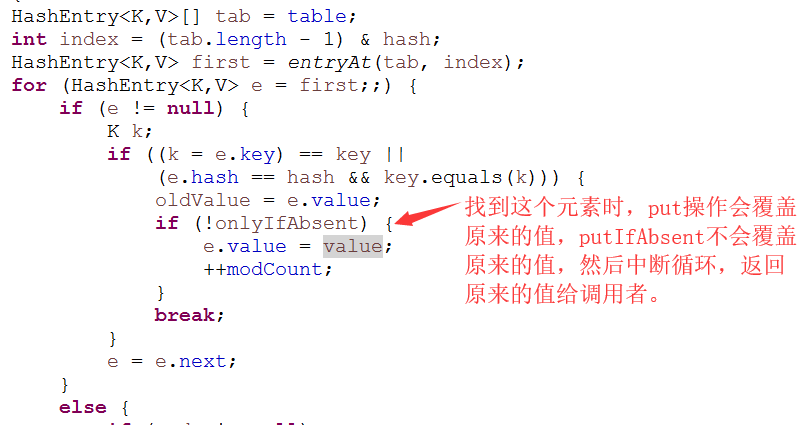
### put()方法

1、首先定位segment，当这个segment在map初始化后，还为null，由ensureSegment方法负责填充这个segment。

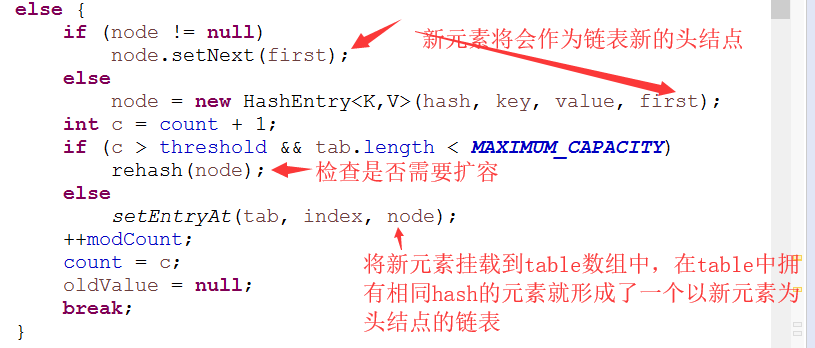
1. 对Segment 加锁



3、定位所在的table元素，并扫描table下的链表，**找到时：**

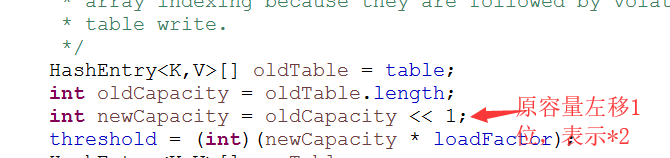


**没有找到时：**



### 扩容操作

Segment 不扩容，扩容下面的table数组，每次都是将数组翻倍



**带来的好处**

假设原来table长度为4，那么元素在table中的分布是这样的：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hash值 | 15 | 23 | 34 | 56 | 77 |
| 在table中下标 | 3 = 15%4 | 3 = 23 % 4 | 2 = 34%4 | 0 = 56%4 | 1 = 77 % 4 |

扩容后table长度变为8，那么元素在table中的分布变成：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hash值 | 56 |  | 34 |  |  | 77 |  | 15,23 |
| 下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

可以看见 hash值为34和56的下标保持不变，而15,23,77的下标都是在原来下标的基础上+4即可，可以快速定位和减少重排次数。

### size方法

size的时候进行两次不加锁的统计，两次一致直接返回结果，不一致，重新加锁再次统计

### 弱一致性

get方法和containsKey方法都是通过对链表遍历判断是否存在key相同的节点以及获得该节点的value。但由于遍历过程中其他线程可能对链表结构做了调整，因此get和containsKey返回的可能是过时的数据，这一点是ConcurrentHashMap在弱一致性上的体现。

## 1.8

### 与1.7相比的重大变化

1. 取消了segment数组，直接用table保存数据，锁的粒度更小，减少并发冲突的概率。
2. 存储数据时采用了链表+红黑树的形式，纯链表的形式时间复杂度为O(n)，红黑树则为O（logn），性能提升很大。什么时候链表转红黑树？当key值相等的元素形成的链表中元素个数超过8个的时候。

### 主要数据结构和关键变量

Node类存放实际的key和value值。

sizeCtl：

负数：表示进行初始化或者扩容,-1表示正在初始化，-N，表示有N-1个线程正在进行扩容

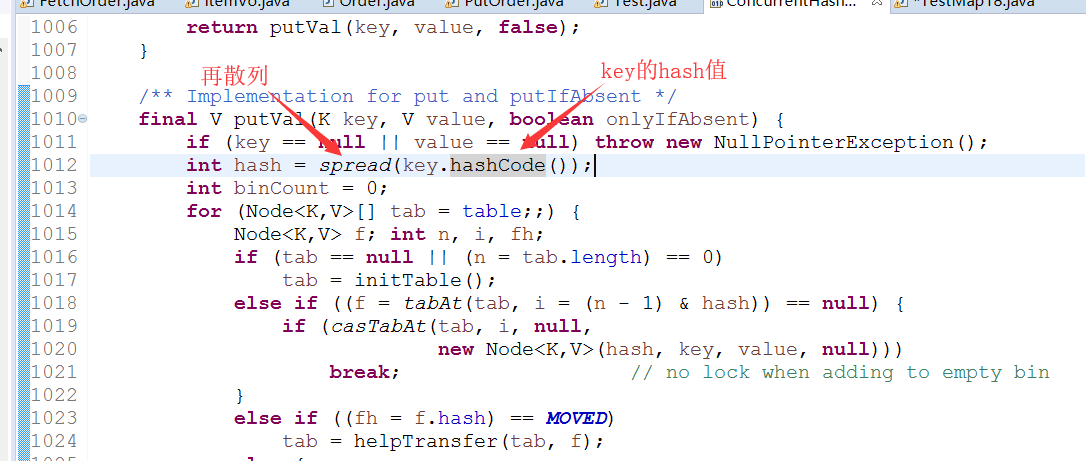
正数：0 表示还没有被初始化，>0的数，初始化或者是下一次进行扩容的阈值

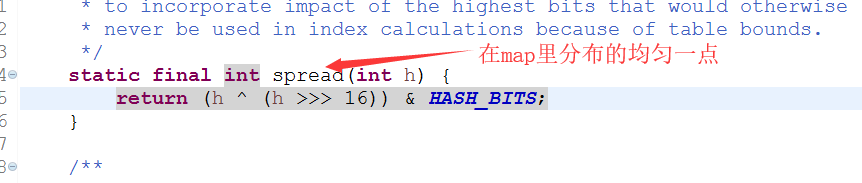
TreeNode 用在红黑树，表示树的节点, TreeBin是实际放在table数组中的，代表了这个红黑树的根。

### 初始化做了什么事？

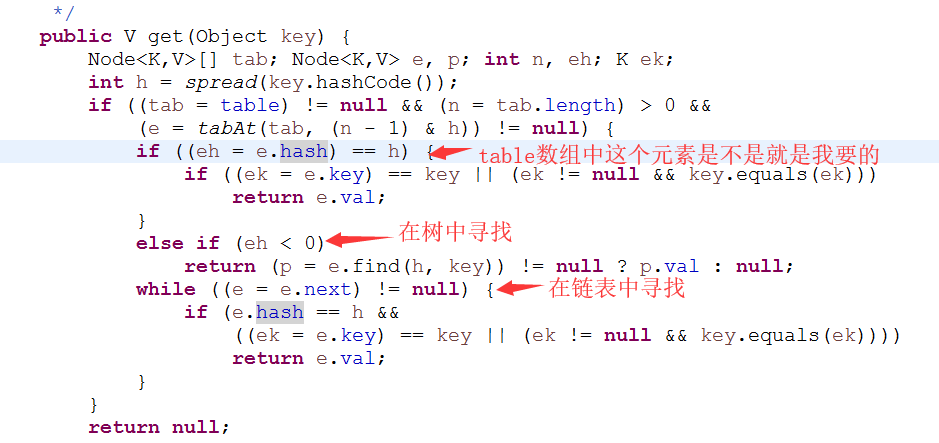
只是给成员变量赋值，put时进行实际数组的填充

### 在get和put操作中，是如何快速定位元素放在哪个位置的？



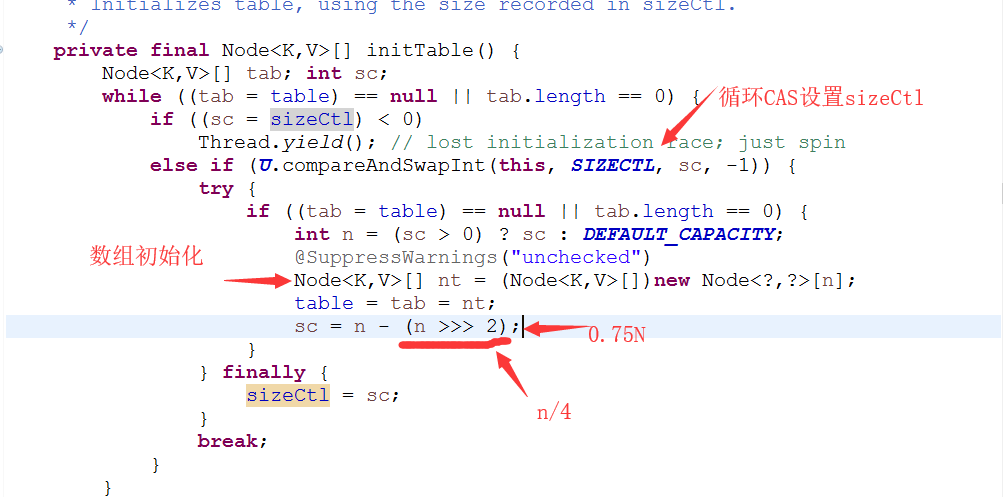


### get（）方法

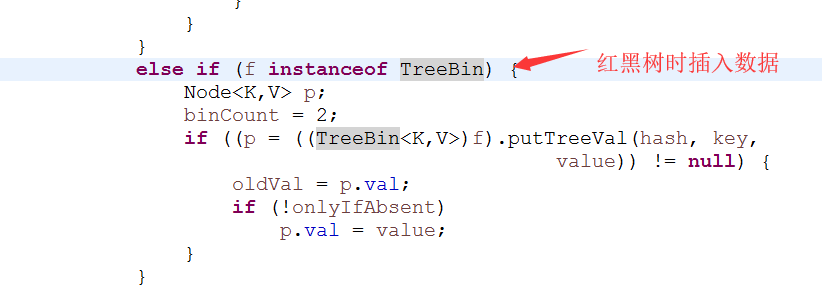


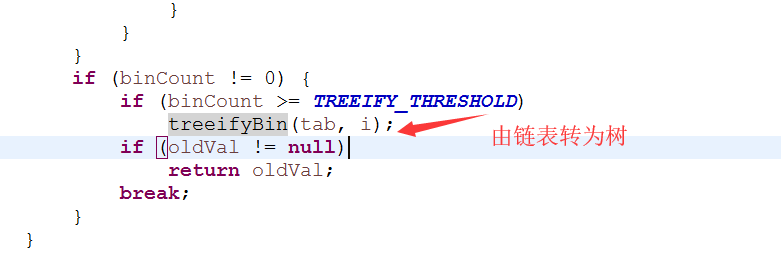
### put()方法

数组的实际初始化









### 扩容操作

transfer()方法进行实际的扩容操作，table大小也是翻倍的形式，有一个并发扩容的机制。

### size方法

估计的大概数量，不是精确数量

### 一致性

弱一致

# 更多的并发容器

## ConcurrentSkipListMap 和 ConcurrentSkipListSet

TreeMap和TreeSet有序的容器，这两种容器的并发版本

### 跳表

SkipList，以空间换时间，在原链表的基础上形成多层索引，但是某个节点在插入时，是否成为索引，随机决定，所以跳表又称为概率数据结构。

## ConcurrentLinkedQueue

无界非阻塞队列，底层是个链表，遵循先进先出原则。

add,offer将元素插入到尾部，peek（拿头部的数据，但是不移除）和poll（拿头部的数据，但是移除）

## 写时复制容器

写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以写时复制容器也是一种读写分离的思想，读和写不同的容器。如果读的时候有多个线程正在向容器添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的，只能保证最终一致性。

适用读多写少的并发场景，常见应用：白名单/黑名单， 商品类目的访问和更新场景。

存在内存占用问题。

# 阻塞队列

## 概念、生产者消费者模式

1)当队列满的时候，插入元素的线程被阻塞，直达队列不满。

2)队列为空的时候，获取元素的线程被阻塞，直到队列不空。

### 生产者和消费者模式

生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这种生产消费能力不均衡的问题，便有了生产者和消费者模式。生产者和消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通信，而是通过阻塞队列来进行通信，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

## 常用方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 方法 | 抛出异常 | 返回值 | 一直阻塞 | 超时退出 |
| 插入方法 | add | offer | put | Offer(time) |
| 移除方法 | remove | poll | take | Poll(time) |
| 检查方法 | element | peek | N/A | N/A |

* **抛出异常**：当队列满时，如果再往队列里插入元素，会抛出IllegalStateException（"Queuefull"）异常。当队列空时，从队列里获取元素会抛出NoSuchElementException异常。
* **返回特殊值**：当往队列插入元素时，会返回元素是否插入成功，成功返回true。如果是移除方法，则是从队列里取出一个元素，如果没有则返回null。
* **一直阻塞**：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到队列可用或者响应中断退出。当队列空时，如果消费者线程从队列里take元素，队列会阻塞住消费者线程，直到队列不为空。
* **超时退出**：当阻塞队列满时，如果生产者线程往队列里插入元素，队列会阻塞生产者线程一段时间，如果超过了指定的时间，生产者线程就会退出。

## 常用阻塞队列

·**ArrayBlockingQueue**：一个由数组结构组成的有界阻塞队列。

按照先进先出原则，要求设定初始大小

·**LinkedBlockingQueue**：一个由链表结构组成的有界阻塞队列。

按照先进先出原则，可以不设定初始大小，Integer.Max\_Value

**ArrayBlockingQueue和LinkedBlockingQueue不同：**

* 锁上面：ArrayBlockingQueue只有一个锁，LinkedBlockingQueue用了两个锁，
* 实现上：ArrayBlockingQueue直接插入元素，LinkedBlockingQueue需要转换。

·**PriorityBlockingQueue**：一个支持优先级排序的无界阻塞队列。

默认情况下，按照自然顺序，要么实现compareTo()方法，指定构造参数Comparator

·**DelayQueue**：一个使用优先级队列实现的无界阻塞队列。

支持延时获取的元素的阻塞队列，元素必须要实现Delayed接口。适用场景：实现自己的缓存系统，订单到期，限时支付等等。

·**SynchronousQueue**：一个不存储元素的阻塞队列。

每一个put操作都要等待一个take操作

·**LinkedTransferQueue**：一个由链表结构组成的无界阻塞队列。

transfer()，必须要消费者消费了以后方法才会返回，tryTransfer()无论消费者是否接收，方法都立即返回。

·**LinkedBlockingDeque**：一个由链表结构组成的双向阻塞队列。

可以从队列的头和尾都可以插入和移除元素，实现工作密取，方法名带了First对头部操作，带了last从尾部操作，另外：add=addLast; remove=removeFirst; take=takeFirst

## 阻塞队列的实现原理

比如，ArrayBlockingQueue就是基于Lock和Condition实现的。