**ArrayList**

1. ArrayList的特性 及 属性



1. 特性
   * 1. RandomAccess : 标记该类的实例 具备下标访问的功能
     2. Serializable : 标记该类的实例对象可以被流化传输
     3. Cloneable : 标记该类的实例对象可以被拷贝，从而创建新的对象(深)
        1. **浅拷贝** ：拷贝后内存中的数据不复制 ， 引用复制一份(数据2份，引用1份)
           1. 引用数据类型 、String 创建同样的数据不改变值的情况下

在堆中不备份数据，只是引用拷贝份

* + - * 1. 实现方式

直接引用 传递

A a = new A ();

A b = a;

* + - 1. **深拷贝** ：拷贝后内存中的数据也复制一份(数据2份，引用2份)
         1. 基本数据类型的 ， String 拷贝后改变值不影响执之前的数据

基本数据类型 ：在栈中直接 备份数据

String ： 拷贝后引用虽然指向的是同一份数据， 但是改变值后 并不会 影响之前的数据 ，而是重新创建一个String对象

* + - * 1. 实现方式

对象反序列后，会成为独立的对象

让引用类型，实现cloneable ，然后调clone 方法

1. 属性
   * 1. defaultCapacity = 10

默认构造器list 初始容量 ，如果size > 10 内部会进行扩容,

为什么list要扩容 ？ 数组长度大小不能改变

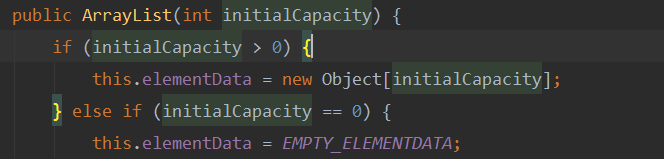
* + 1. empty\_elementData = {}

创建空的new ArrayList(0)时 ， 引用指向的数组。那么无论创建多少个new ArrayList<>实例都指向这个 数组。从而节省内存。 当数组在第一次使用时 会赋初始值 10

* + 1. defaultCapacity\_empty\_elementData

创建带数组内存大小的list默认构造器指向的数据

* + 1. elementData 正在存放数据的地方 长度等于数组容量



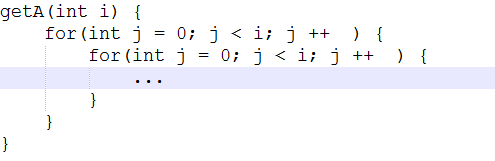
* + 1. 为什么 创建list指定大小跟不指定大小 ，指向的 数组不是同一个

不指定大小是默认的 ， 在没往数组立存数据的情况下是不进行数组初始化的

指定大小 直接进行初始化

1. 什么是时间复杂度 ： 参数 跟 方法执行时间的 关系
   1. 常量时间复杂度 O(1) : 如list.get(1) 直接取一次
   2. 线性时间复杂度 O(n) : 跟循环有关 如迭代器取数据

指数相关时间复杂度 ：O(n^2) 如 循环嵌套



* 1. 对数相关时间复杂度 ： 二分查找

1. Arraylist 添加元素 add(obj)

分两种情况 1、 默认是10大小第一次扩容 2、 指定数据大小

当list不指定初始化大小 ，默认ArrayList中数组的大小时10 ，里add元素时 ， size+1

然后跟数组大小比较，如果 size > 数组长度 ， 就会在旧的数组长度的基础上扩容1.5 倍然后创建一个新的数组 ， 并将就的数组中的数据拷贝到新的数组中(拷贝数组比较 耗性能)

注意 ： 其实在不扩容情况下， list 添加元素的 效率不比linkedList差 因为add(obj) 直接将数据添加到尾部，不牵扯到数据移动。

在存储数据明确的情况相下，可以提前设置ArrayList的大小再往里添加数 据，提高性能

1. Arraylist 添加元素 add(index , obj)

性能比add(obj)低 ， 不扩容时 牵扯到数据的移动 ，如果需要扩容 会先扩容在移动

1. subList() 截取list

List 是可以截取 新的List， 但是截取之后 不能对旧的集合进行 增删影响长度的操作，否则会报错 。 如果截取长度大于 原Size 大小也会报错

1. 根据下标删除元素 remove(int index) 每次删除都会产生新的数组

类似添加元素 ： 将删除的元素 置为null ， 然后后面的元素 下标有 -1

然后将值拷贝至 新数组 ，数组长度 size - 1

1. 跟据值删除元素 remove(Object obj)

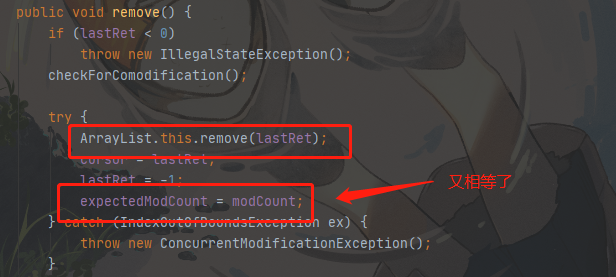
根据值从头遍历list 然后根据值比较删除 再拷贝新的数组（性能比 根据下标删除低）

1. 为什么 不能使用 增强for循环 调 list.remove 方法删除元素 (会报错)

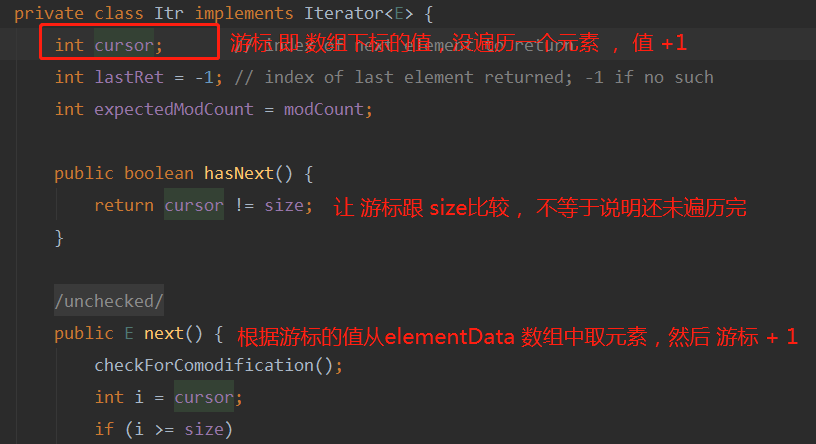
增强for循环 内部的循环实现是使用 迭代器 ， 每当对list做一次操作 他的 modcount 都会 + 1 ， 就会 导致 expectCount ！= modCount 。 触发立即错误机制

1. 为什么使用 迭代器的 remove就可以删除

迭代器的remove方法执行后 ， 会重置 expectCount



1. 迭代器 arrayList().iterator();



1. arrayList().iterator() 和 arrayList().listIterator() 区别
   1. 使用范围
      * 1. Iterator 遍历 list 、set 、 map(获取 key集合转iterator)
        2. listIterator 只能遍历 list
   2. 遍历方式
      * 1. Iterator 只能从 第一个开始遍历然后移动指针。
        2. arrayList().listIterator() 可以从任意位置开始向后遍历List ，提供了listIterator(int i) 方法 ， 还可以逆向遍历
   3. 对List的操作不同
      * 1. 都支持删除元素但是listIterator功能更强大 ，支持修改元素 ，Iterator 不支持修改
2. 线程安全的ArrayList

使用到线程安全的list较少 ， 一般不知用vector 性能较差，通常使用

Collections.*synchronizedList*(list) 转成线程安全的  
 CopyOnWriteArrayList

1. 将数组转成list的 ArrayAsList(数组) 方法

数组的 api 比较少 ， 可将数组转成 list

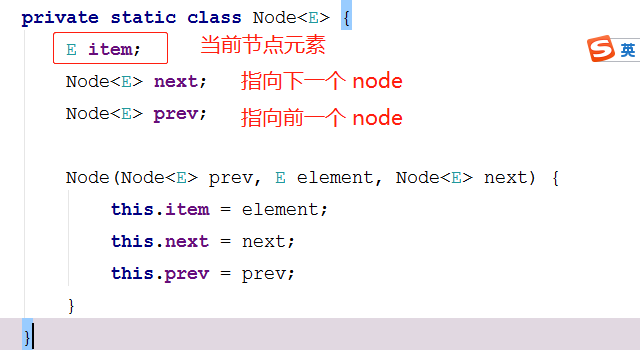
但是ArrayList() 在 存放基本类型的数组时 ，会将数组对象放在 list的一个元素中

也就是说不能将数组转成 List 。引用类型是可以的

**LinkedList**

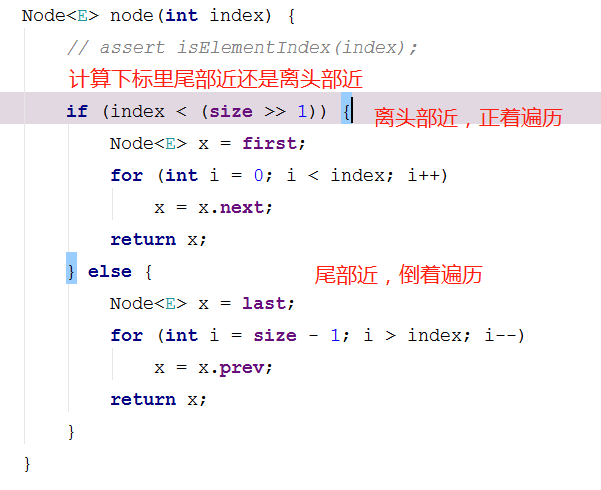
1. linkedList 链表 内部 维护了一个 node 内部类用于 ，关联前后节点。

在插入元素时默认从尾部插入 ， 当指定位置插入时 就是将node链表断开重新将插 入的元素的 next 、prev 指向两边的node 所有性能比较高 （删除也是 一样的原理）



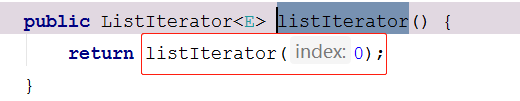
1. linkedList 查询方式
2. Get(int index)方法

由于linkedList是链表不能像arrayList一样通过下标直接取值 ，每次get(i)通过二 分法判断从前、或者从后遍历去查找元素 性能非常低。因此大部分场景下都是查 询比较多 。 所以很少用linkedlist



1. 迭代器 查询 通过游标来记录当前的 node 查询性能高，只有一次遍历

linkedList.iterrator() ; 默认游标从0开始， 然后往后移动查询node节点

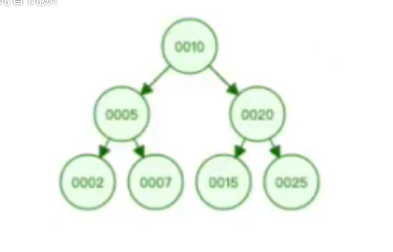


**HashMap**

1. 二叉排序树

特点 ：有序 且 左边的节点 比父节点小

右边的节点 比父节点大



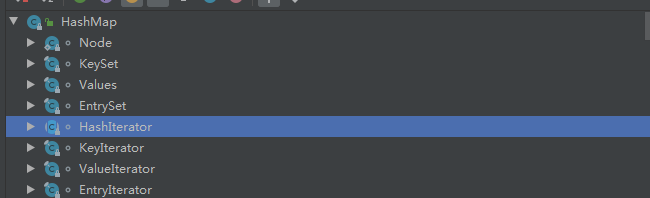
缺点 ： 当数据 一次递增或者递减时 ，二叉树会变成链表，红黑树可以解决

1. 红黑树

特点有一个旋转平衡的过程 ，如果发现递增 会自动将中间的数向上移动 ，让的小左移，大的数据又移动

1. HashMap 中的成员 13个 内部类 ， 6 个属性

**内部类 ：**



1. Node ： 实现了 map 的entry<K,V> 接口存放的数据会被封装成 node对象(单链表)
   * 1. Key
     2. Value
     3. Node<k,v> next

当数据插入单向链表时 ，会插入在头部(链表从头部插入性能高 直接node.next ,如果插入到尾部 还要循环找到最后一个

头部插入后 ，node 数组的下标指向头结点，在扩容时数据拷贝到新的数组时链表顺序会反过来 (jdk1.7)

B、keySet： 保存key的set集合

C、Values： 保存值的集合

D、EntrySet：保存 entry<K,V>的set集合

**属性 ：**



A、size map中元素的总个数

B、capacity hashmap中桶的数量 即 ： 数组长度 默认16

C、loadFactor 装载因子，用来衡量hashmap满的程度 初始值是 0.75

如果map中put值后 ， loadFactor = size/capacity

D、threshold : 阈值 当size大于Threshold 时会进行扩容

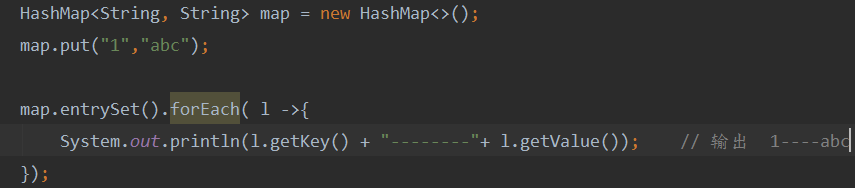
threshold=capacity\*loadFactor

初始值 ： 16 \* 0.75 = 12

Hashmap 首次扩容是在 size = 13 时

当扩容后会重新计算阈值 ： newCapacity \* 加载因子 32\*loadFactor

1. table node 数组
2. entrySet 保存key、value的set集合



1. 为什么要扩容

目的是为了让 链表变短 ，提高查询效率

1. put(K key,V value) 的过程

hash(key) 对key进行hash运算 ，如果 key == null ，hash = 0 ，否则进行

1. 单线程下HashMap扩容

首先创建新数组大小 \*2 ， 然后将链表拷贝到新的数组中

拷贝时通过位运算计算新的数组下标 符计算后有一个规律 要么在原来的位置 ，要么就在 原来位置 \*2 的位置上，然后将这个位置的链表循环拷贝的新的数组中且顺序链表的顺序和原来相反

注意 ：多线程下 HashMap 在扩容时会存在多线死锁的现象

当两个线程同时添加数据且size都刚好大于阈值需要扩容时 ， 这两个线程会创建两个 新的数组，如果其中一个线程执行较慢 在拷贝值时会读取上一个线程修改后的值就有一定几率会出现循环链表的情况

1. hashmap扩容时为什么不直接？扩容 而是通过为运算？

性能高 cpu指令是不认识乘法运算的， 而为运算cpu是可以直接执行的

1. hashMap remove(key)删除元素的问题报错的问题

HashMap.keySet 获取 key集合时， 内部是获取keySet 的迭代器 ，当循环ketSet时 其实就是 调用迭代器的 hasnNext方法 而在hasNext内部迭代器会判断 modcount 是 否等于 expectmodCount ，如果等于迭代器才会去取值 不等于会报错

当 remove(key)值时 ，modcount 的值会 +1 (modcount 表示有一次操作值就会 +1)， 而expectmodCount 值是不变的 ，所以在使用迭代器hasnNext时会报错不能正常删除 元素(ArrayList删除元素也一样)

**如何解决从map中删除元素** : 直接使用迭代器的remove()方法进行删除 在删除时 会有 expectModCount = modeCount的操作

集合中 modeCount 存在的意义 ： 快速失败 ，当expectModCount != modeCoun时直 接抛异常 ，不再执行下面的代码再去报错， 用于解决多线程操作同一个hashMap 时 ，

一个线程修改 modeCount +1 ，一个线程查询 一旦校验不一致就抛异常

13、hashMap 1.7底层实现

(1) Jdk 1.7 之前 : 数组 + 链表 (链表用来防止 hash 冲突)

(2) 为什么使用链表 :防止hash碰撞

(3) 允许存null 值 :Null 默认存放第一个数组位置 ，且只能有一个

(4)hashmap 初始化数组容量的问题

Hashmap的初始化数组容量的值是2 幂次方 ， 如果写入的值不是 2

的幂次方， 那么hashmap 内部会设置一个 2的幂次方 且刚好比填入的值大的数

(5)当put值时根据key获取 hashCode() 时为什么会对计算出的 hash值进行右移、左移

目的为了在计算 数组下标时 ，尽可能得到不同的值 让数据尽量分布在数组不同的位 置， 不至于让有些链表过长， 从而在 get数据时效率较低

(3)Hashmap put值计算数组下标时 ，为什么不取 余 ， 而是用 &

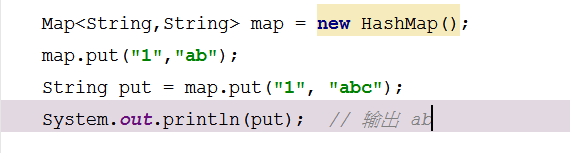
为了性能高 ，之前是取余

1. 当存的字符串想相同时，hashmap中只存放一份数据

相同的字符串计算Hashcode() 值是一样的 。

map.put(“k1”,”v1”) 返回oldValue值

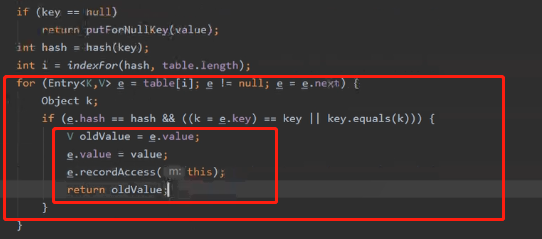
如果 put的key已存在 就覆盖原有的value值， 返回原有的 value值 。如果是添加 新 值就是返回null



(7)Hashmap 是如何对旧的值进行覆盖的?

通过for循环，先比较hash值 因为key相等计算出的 hash值相等然后就能找到 共

同的数组下标，然后再链表上找到该key 将旧的值替换成新的值，并返回旧的值



1. 扩容

jdk 1.7 是先扩容再把 值put到hashmap 里面的。 1.8是相反的，先添加再扩容

(8)Hashmap为什么要扩容resize

防止在数据量比较大时 ，链表长度过长 put、get的性能较低

(9)hashmap中 size 、capacity、loadFactor、threshold 的含义

A、size map中元素的总个数

B、Capacity hashmap中桶的数量 即 ： 数组长度

C、loadFactor 装载因子，用来衡量hashmap满的程度 初始值是 0.75

如果map中put值后 ， loadFactor = size/capacity

D、Threshold : 预值 当size大于Threshold 时会进行扩容

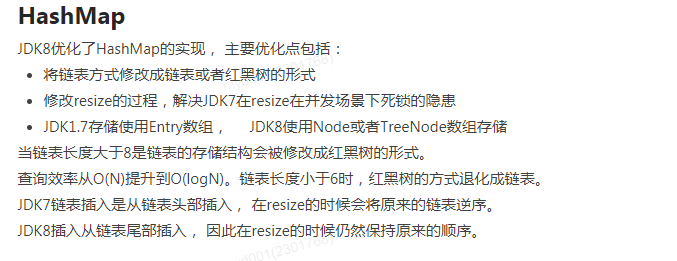
threshold=capacity\*loadFactor

初始值 ： 16 \* 0.75 = 12

Hashmap 首次扩容是在 size = 13 时

(10)Jdk 1.7是怎么扩容的

首先当 size > =预值时 ， Capacity 会 乘以 2 new一个32大小的entry 数组。 然后 循环数组， 在循环链表得到每一个entry 然后根据之前算好的hash值 &(与)计算出数组下标 ，在从新将元素 一个个放进新的 数组中



1. hashMap 、LinkendHashMap 、treeMap、currentHashMap、CurrentSkipHashMap
   1. hashMap 、LinkendHashMap、treeMap

A、hashmap 一个无序 线程不安全

B、LinkendHashMap 使用链表实现的有序map

C、treeMap使用红黑树实现的有序map

* 1. currentHashMap、CurrentSkipHashMap、hashtable

A、Hashtable不在current包下，使用 sync实现线程安全 后面两个使用lock实现 线程安全

B、都是线程安全 currentHashMap无序， 当线程较多时CurrentSkipHashMap有序且比currentHashMap支持更高的并发。

## **HashMap底层数据结构**

JDK7：数组+链表

JDK8:  数组+链表+红黑树（看过源码的同学应该知道JDK8中即使用了单向链表，也使用了双向链表，双向链表主要是为了链表操作方便，应该在插入，扩容，链表转红黑树，红黑树转链表的过程中都要操作链表）

## **JDK8中的HashMap为什么要使用红黑树？**

当元素个数小于一个阈值时，链表整体的插入查询效率要高于红黑树，当元素个数大于此阈值时，链表整体的插入查询效率要低于红黑树。此阈值在HashMap中为8

## **JDK8中的HashMap什么时候将链表转化为红黑树？**

这个题很容易答错，大部分答案就是：当链表中的元素个数大于8时就会把链表转化为红黑树。但是其实还有另外一个限制：当发现链表中的元素个数大于8之后，还会判断一下当前数组的长度，如果数组长度小于64时，此时并不会转化为红黑树，而是进行扩容。只有**当链表中的元素个数大于8，并且数组的长度大于等于64时才会将链表转为红黑树**。

上面扩容的原因是，如果数组长度还比较小，就先利用扩容来缩小链表的长度。

## **JDK8中HashMap的put方法的实现过程？**

1. 根据key生成hashcode
2. 判断当前HashMap对象中的数组是否为空，如果为空则初始化该数组
3. 根据逻辑与运算，算出hashcode基于当前数组对应的数组下标i
4. 判断数组的第i个位置的元素（tab[i]）是否为空

A.如果为空，则将key，value封装为Node对象赋值给tab[i]

B.如果不为空：

a.如果put方法传入进来的key等于tab[i].key，那么证明存在相同的key

b.如果不等于tab[i].key，则：

a).如果tab[i]的类型是TreeNode，则表示数组的第i位置上是一颗红 黑树，那么将key和value插入到红黑树中，并且在插入之前会判断 在红黑树中是否存在相同的key

b)如果tab[i]的类型不是TreeNode，则表示数组的第i位置上是一个 链表，那么遍历链表寻找是否存在相同的key，并且在遍历的过程中 会对链表中的结点数进行计数，当遍历到最后一个结点时，会将 key,value封装为Node插入到链表的尾部，同时判断在插入新结点之 前的链表结点个数是不是大于等于8，如果是，则将链表改为红黑树。

c.如果上述步骤中发现存在相同的key，则根据onlyIfAbsent标记来判断是否需要更新 value值，然后返回oldValue

**5.**modCount++

**6**.HashMap的元素个数size加1

**7**.如果size大于扩容的阈值，则进行扩容

## **JDK8中HashMap的get方法的实现过程**

1. 根据key生成hashcode
2. 如果数组为空，则直接返回空
3. 如果数组不为空，则利用hashcode和数组长度通过逻辑与操作算出key所对应的数组下标i
4. 如果数组的第i个位置上没有元素，则直接返回空
5. 如果数组的第1个位上的元素的key等于get方法所传进来的key，则返回该元素，并获取该元素的value
6. 如果不等于则判断该元素还有没有下一个元素，如果没有，返回空
7. 如果有则判断该元素的类型是链表结点还是红黑树结点
8. 如果是链表则遍历链表
9. B.如果是红黑树则遍历红黑树

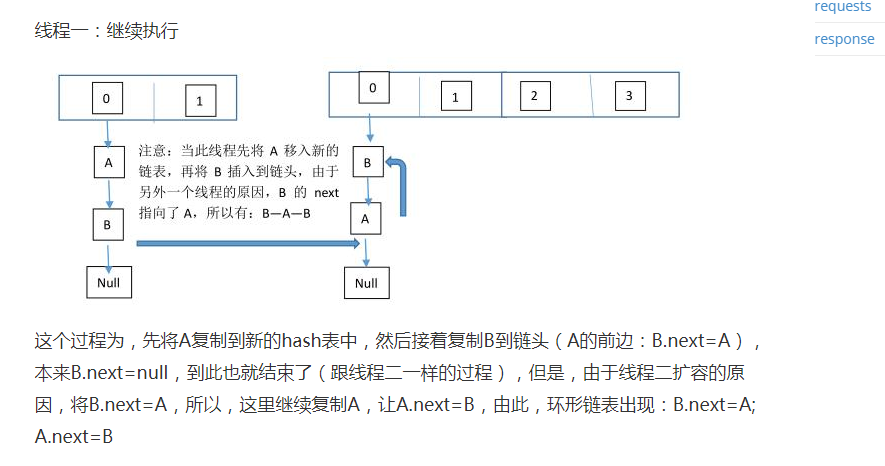
**8.**找到即返回元素，没找到的则返回空

## **JDK7与JDK8中HashMap的不同点**

1. JDK8中使用了红黑树
2. JDK7中链表的插入使用的**头插法**（扩容转移元素的时候也是使用的头插法，头插法速度更快，无需遍历链表，但是在多线程扩容的情况下使用头插法会出现循环链表的问题，导致CPU飙升），JDK8中链表使用的**尾插法**（JDK8中反正要去计算链表当前结点的个数，反正要遍历的链表的，所以直接使用尾插法）
3. JDK7的Hash算法比JDK8中的更复杂，Hash算法越复杂，生成的hashcode则更散列，那么hashmap中的元素则更散列，更散列则hashmap的查询性能更好，JDK7中没有红黑树，所以只能优化Hash算法使得元素更散列，而JDK8中增加了红黑树，查询性能得到了保障，所以可以简化一下Hash算法，毕竟Hash算法越复杂就越消耗CPU
4. 扩容的过程中JDK7中有可能会重新对key进行哈希（重新Hash跟哈希种子有关系），而JDK8中没有这部分逻辑
5. JDK8中扩容的条件和JDK7中不一样，除开判断size是否大于阈值之外，JDK7中还判断了tab[i]是否为空，不为空的时候才会进行扩容，而JDK8中则没有该条件了
6. JDK8中还多了一个API：putIfAbsent(key,value)
7. JDK7和JDK8扩容过程中转移元素的逻辑不一样，JDK7是每次转移一个元素，JDK8是先算出来当前位置上哪些元素在新数组的低位上，哪些在新数组的高位上，然后在一次性转移

Jdk7 环形链表的产生原因 ：

假如 在添加第3个元素时， 需要扩容



**线程1 扩容：**

使用头插法 先根部遍历链表 ， 依次移动A 、B 进行数据拷贝

扩容的 A、B 位置互换 步骤如下 ：

next = null --> B.next = A ---> A.next = null

**接着线程2 也进行扩容 ：**

有该位置有值 直接A.next = B --> B.next = A

就出现了 两个 节点互相引用的问题

# **ConcurrentHashMap JDK7中的ConcurrentHashMap是怎么保证并发安全的？**

主要利用Unsafe操作+ReentrantLock+分段思想。

主要使用了Unsafe操作中的：

1. compareAndSwapObject：通过cas的方式修改对象的属性
2. putOrderedObject：并发安全的给数组的某个位置赋值
3. getObjectVolatile：并发安全的获取数组某个位置的元素

分段思想是为了提高ConcurrentHashMap的并发量，分段数越高则支持的最大并发量越高，程序员 可以通过concurrencyLevel参数来指定并发量。ConcurrentHashMap的内部类Segment就是用来表示 某一个段的。

每个Segment就是一个小型的HashMap的，当调用ConcurrentHashMap的put方法是，最终会调用 到Segment的put方法，而Segment类继承了ReentrantLock，所以Segment自带可重入锁，当调用 到Segment的put方法时，会先利用可重入锁加锁，加锁成功后再将待插入的key,value插入到小型 HashMap中，插入完成后解锁。

## **JDK7中的ConcurrentHashMap的底层原理**

ConcurrentHashMap底层是由两层嵌套数组来实现的：

1. ConcurrentHashMap对象中有一个属性segments，类型为Segment[];
2. Segment对象中有一个属性table，类型为HashEntry[];

当调用ConcurrentHashMap的put方法时，先根据key计算出对应的Segment[]的数组下标j，确定好 当前key,value应该插入到哪个Segment对象中，如果segments[j]为空，则利用自旋锁的方式在j位 置生成一个Segment对象。

然后调用Segment对象的put方法。

Segment对象的put方法会先加锁，然后也根据key计算出对应的HashEntry[]的数组下标i，然后将 key,value封装为HashEntry对象放入该位置，此过程和JDK7的HashMap的put方法一样，然后解锁。

在加锁的过程中逻辑比较复杂，先通过自旋加锁，如果超过一定次数就会直接阻塞等等加锁。（具 体流程请求看vip视频.）

## **JDK8中的ConcurrentHashMap是怎么保证并发安全的？**

主要利用Unsafe操作+synchronized关键字。

Unsafe操作的使用仍然和JDK7中的类似，主要负责并发安全的修改对象的属性或数组某个位置的值。

synchronized主要负责在需要操作某个位置时进行加锁（该位置不为空），比如向某个位置的链表进 行插入结点，向某个位置的红黑树插入结点。

JDK8中其实仍然有分段锁的思想，只不过JDK7中段数是可以控制的，而JDK8中是数组的每一个位 置都有一把锁。

当向ConcurrentHashMap中put一个key,value时，

1. 首先根据key计算对应的数组下标i，如果该位置没有元素，则通过自旋的方法去向该位置赋值。
2. 如果该位置有元素，则synchronized会加锁
3. 加锁成功之后，在判断该元素的类型
4. 如果是链表节点则进行添加节点到链表中
5. 如果是红黑树则添加节点到红黑树

**4.**添加成功后，判断是否需要进行树化

**5**.addCount，这个方法的意思是ConcurrentHashMap的元素个数加1，但是这个操作也是需要并发安全的，并且元素个数加1成功后，会继续判断是否要进行扩容，如果需要，则会进行扩容，所以这个方法很重要。

**6.**同时一个线程在put时如果发现当前ConcurrentHashMap正在进行扩容则会去帮助扩容。

扩容流程源码解析请看vip视频讲解。

## **JDK7和JDK8中的ConcurrentHashMap的不同点**

这两个的不同点太多了...，既包括了HashMap中的不同点，也有其他不同点，比如：

1. JDK8中没有分段锁了，而是使用synchronized来进行控制
2. JDK8中的扩容性能更高，支持多线程同时扩容，实际上JDK7中也支持多线程扩容，因为JDK7中的扩容是针对每个Segment的，所以也可能多线程扩容，但是性能没有JDK8高，因为JDK8中对于任意一个线程都可以去帮助扩容
3. JDK8中的元素个数统计的实现也不一样了，JDK8中增加了CounterCell来帮助计数，而JDK7中没有，JDK7中是put的时候每个Segment内部计数，统计的时候是遍历每个Segment对象加锁统计（当然有一点点小小的优化措施，看视频吧..）。

4、synchronizeMap 和 currentMap的区别

synchronizeMap 一次性加锁 ，锁住synchronizeMap 对象只能让单线程访问资源

currentMap 分段加锁来保证资源的安全性 ，同时性能比 synchronizeMap 高

5、Servlet 不是线程安全的 ， 在springMVC中 threadLocal保证了 线程安全了问题

6、在线程运行时如果中断了会怎样 ？

开始是否进行 异常捕获 ， 如果不捕获 就终止执行

如果捕获将会发线程异常信息 抛给 Thrand.UncaughExceptionHnadler

7、Int 和 Integer区别

Int 基本数据类型 默认值是 0 ， 占 4 字节

Integer引用数据类型 默认值是 null 是对象 ，大小根据 类型指针 + 实例数据 中的属性 + 对齐填充 来进行计算大小

Int之间的变量比较值大小是通过 ==

Integer 之间比较大小最好使用equals

如果使用 == ， 只能比较 -128 - 127 之间的数值 。 因为Integer会将这 个区间的数据进行缓存 ， 比较大小时直接取

如果超过了这个大小 ， 将会创建新的对象 使用 == 则会比较内存地址

使用 equals ，Integer 对equals方法进行了重写 也 是比较值

8、Timer 定时器类

Timer : java util 提供的形式任务类 ，用于实现周期性的 调度任务 。

但是不能定义 某个时间执行任务 ，如 每天早上6点执行。

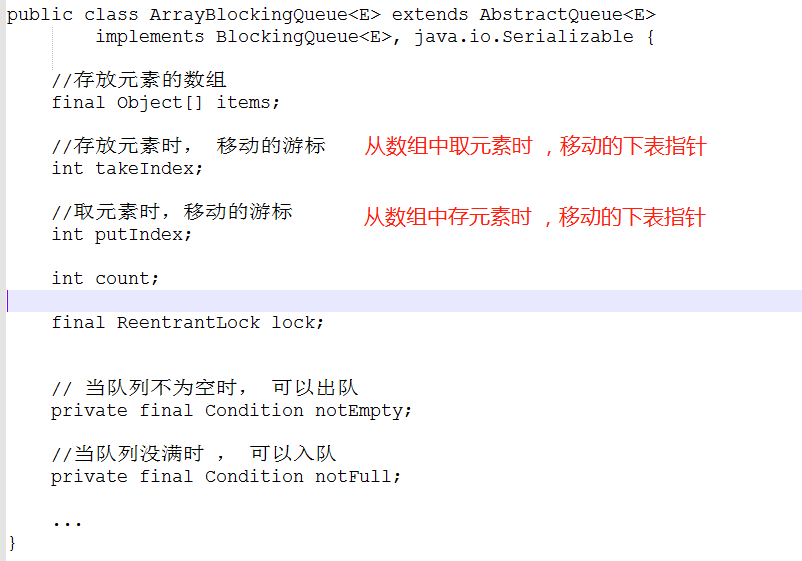
Quartz ： 功能强大的定时任务框架

触发器 ： 用于触发任务

作业 ： 处理任务

**ArrayBlockingQueue**

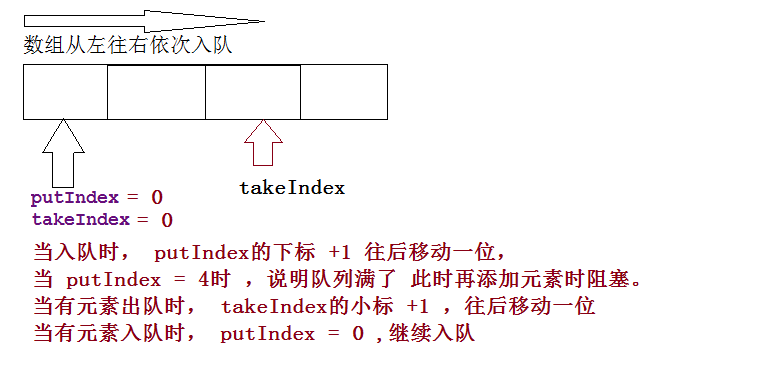
1. 队列操作相关
   1. 存
      * 1. add 添加 满了抛异常
        2. offer 添加 满了返回false
        3. put 添加 满了阻塞 ， 且可以设置阻塞时间
   2. 取
      * 1. remove 取元素，没了抛异常
        2. poll 取元素，没了返回null
        3. tack 取元素，没了阻塞等待 ，可以设置等待时间
   3. 查看队首元素
      * 1. Element 队列为空时， 抛异常
        2. peek 队列为空时， 返回false
2. ArraylistBlockingQueue 有界阻塞队列 在初始化时指定大小, 内部成员



注意 ：在使用 Condition唤醒 和 阻塞 线程时 ，要使用同一condition对象。

如 notEmpty： 在队列空时 阻塞其他线程notEmpty.await();，当唤醒线程时也要 使用noetEmpty.signal();

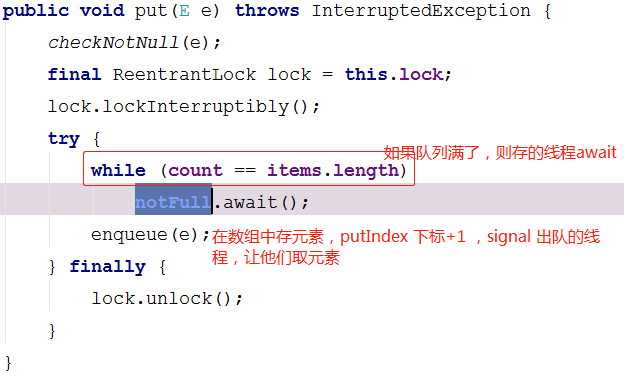
1. ArrarListBlockingQueue的原理



多线程操作同一个队列时 ，会对当前队列上锁同时只能有一个线程往队列中存取元素。

当存的时候 先wiat 取的线程，存完后再signal取的线程，让他们去取元素

1. 入队



1. 防止指令重排的方式
   1. 锁 (Lock 、 sync )
   2. Voiltle
   3. Final 修饰
   4. 使用 UnSafe类提供的内存屏障

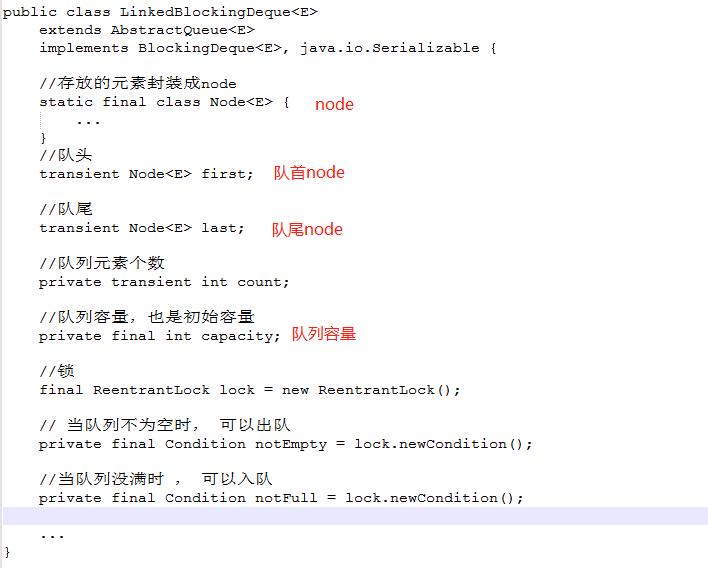
**LinkedBlockingQueue**

1. 线程池的默认阻塞队列

不建议使用 jdk 自带的线程池的原因是其用的阻塞队列是无界队列 和 cache缓冲线程池无限的创建线程 ,我们在使用线程池时 使用

newThreadPoolExcutor() ，指定自己的阻塞队列 ，如果队列满了指定自己的拒绝策略

1. 无界阻塞队列 LinkedBlockingQueue成员



**新特性**

1. Lamada表达式
   1. 实现接口 适用接口中只有一个不带参数的方法的情况

Runnable r = () -> {};

* 1. 实现接口 适用接口中只有一个带参数的方法的情况

1. 方法调用 ::

类名:: 方法名 适用调用普通方法，静态方法，构造器

使用方式 ： A、创建接口 并使用@FunctionInterface注解 并创建方法

1. 创建 类

使用： Interface.方法名(类名::方法名)

注意： 接口参数类型应该是 T

使用场景 ： 方法传值

1. @FunctionInterface 注解

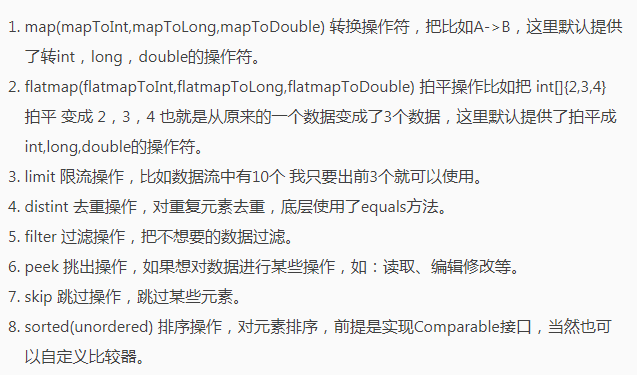
如果一个接口中此注解注释 ， 就表明此接口中不能再添加方法包括 default方法

1. 其实适用lamada表达式就是适用匿名内部类。 匿名内部类只能够引用 final 类型的变 量 ,适用 idea 写代码时 可以不给变量加 final修饰 ，在编译代码时会自动上final 但是这个变量不能在匿名内部类中 赋值 ，因为是final 。否则编译报错。 另外匿名 内部类在编译时，会重新编一个新的class文件, 文件命名 (类名$方法名.class)
2. Stream 管道流 ， 串行、 并行(多线程) 操作 List

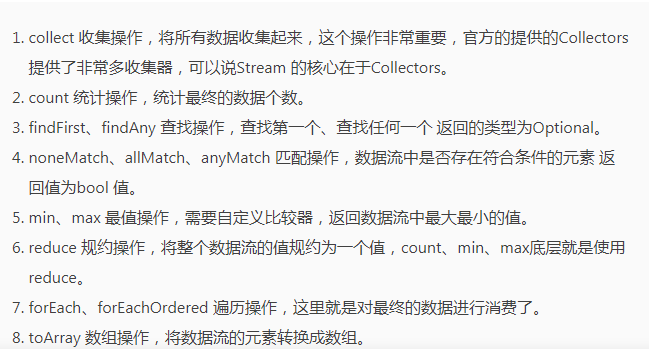


延迟执行 ： 做终止操作的时候 ，操作做真正的操作处理数据

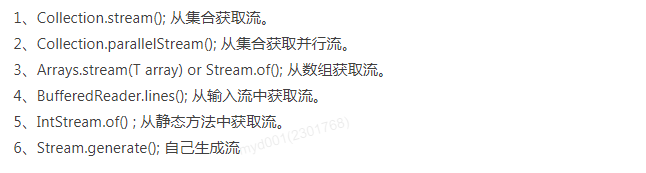
* 1. 操作分为两种 中间操作符 和 终止操作符
     + 1. 中间操作（特点：处理结果可以继续交给下一个中间操作继续加工数据）



* + - 1. 终止操作（特点：数据加工到此处，不会向下传递数据并且只能使用一次）

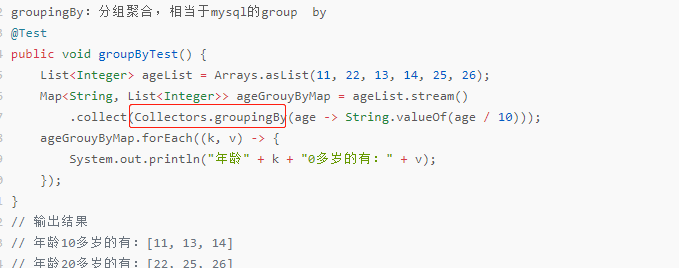


* 1. 创建数据源的方式有哪些 （如果将 数组、集合转成stream对象）





* 1. 分组统计 类似 mysql 中的 group by



**线程安全得单例模式**

1. 内存泄漏问题

内存泄漏的现象 ： 对象是可达的 ，但是是无用的对象 导致不能回收导致

如果这些对象长时间不能被回收，且不断的增多就会导full gc 甚至oom

1. 如何避免 内存泄漏
   1. 连接即时关闭
   2. 变量的不合理定义： 作用的范围 大于 使用范围
   3. 定义了静态 集合应注意
2. 又如何定位内存泄漏的问题
   1. 增加 堆内存的 ，但是治标不治本
   2. 通过 jstack -gc pid 要观察 gc 的情况 。 重点关注 full gc 的次数 以及发生的频率

如果full gc 比较频繁应进行优化

1. 较优的单例实现 ： 高效、安全、懒加载
   1. **Holder模式 ( DCL模式的单例 中的 volaitor 是防止指令重排)**

使用较多：利用静态内部类 在出次使用只实例化一次的特点 且没有使用锁，实现懒 汉模式 ，并且确保线程安全

声明类的时候，成员变量中不声明实例变量，而放到静态内部类中

使用场景（项目中的线程池 ， 不能用到多线程时就创建一个新的线程池）

public class HolderDemo {

private HolderDemo(){

}

private static class Holder{

private static HolderDemo instance=new HolderDemo();

}

public static HolderDemo getInstance(){

return Holder.instance;

}

}

当调用 HolderDemo .getInstance() 方法时 记性HolderDemod的初始化，而静态内部类的加载不需要依附外部类，在使用时才会进行类的初始化，（首次用） 所以实现了懒汉懒汉 ，由于类只初始化一次 也是线程安全的

* 1. **枚举**

public class EnumSingletonDemo {

private EnumSingletonDemo(){

}

//延迟加载

private enum EnumHolder{

INSTANCE;

private static EnumSingletonDemo instance=null;

private EnumSingletonDemo getInstance(){

instance=new EnumSingletonDemo();

return instance;

}

}//懒加载

public static EnumSingletonDemo getInstance(){

return EnumHolder.INSTANCE.instance;

}

}

**JMM**

1. 并发的几个特性
   1. 可见性

A、Synchronize、lock 、Volalilte 具有可见性 。 当线程操作共享数据时， 可以将变 化的数据刷新到主存 。让其他线程感知到共享数据的变化 即 可见性

Synchronize 、Volalilte、lock 还可以防止 编译、运行时的指令重排(即有序性)

B、当多线程操作同一资源 （切换线程也具有可见性） ，由于要到主存中同步数据 在切换线程时也 具有可见性 但是这种是随机的， 并不能使用代码真正保障数据的可见性

假如 Thread.sleep() 睡眠线程， 意思就是 当前线程让出cpu ，当时间结束时,线程会重新来执行 ，这个就会发生cpu线程切换重新同步数据，能从新读主存 中的数据 具有可见性

* 1. 有序性

Synchronize 、lock、Volalilte

即代码按照编写的步骤运行，不进行指令重排

final关键字的可见性是指：被final修饰的字段在构造器中一旦初始化完成，并且构造器没有把“this”的引用传递出去（this引用逃逸是一件很危险的事情，其他线程有可能通过这个引用访问到“初始化了一半”的对象），那在其他线程中就能看见final字段的值。

* 1. 原子性

一个操作或几个操作 视为一个完整的操作， 只能被当前线程执行完，其他线程才能争抢锁执行

1. sync lock 锁能保证代码块原子性
2. AutomicInteger 能保证操作 变量时的原子性 ，用于对变量进行计算操作

总结 ： Synchronize 、lock、Volalilte 既满足可见性，又满足有序性 ，锁能保证原子性

多线程下如何判断是否线程安全 ， 就是看是否满足这3个特性

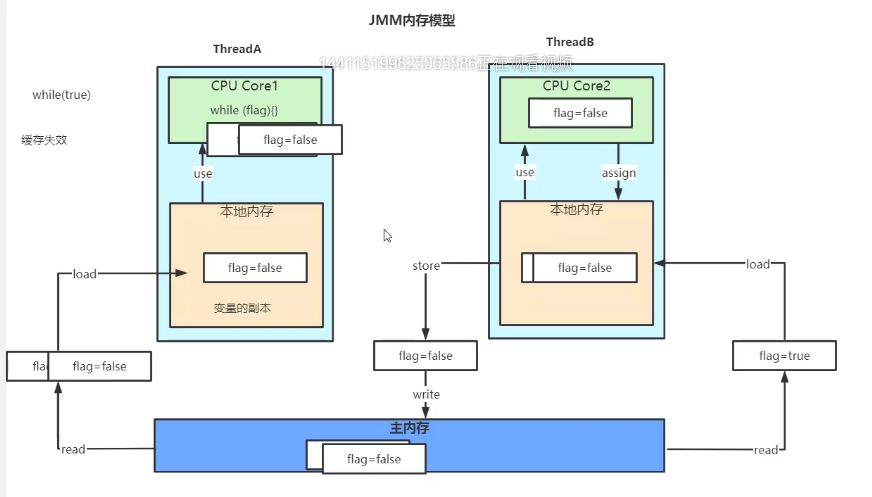
1. 调用线程为什么要 调用statr()方法 ，而不是直接调用run 方法

start() 方法是创建线程 ， 等线程创建完毕 调用run执行任务

如果直接调用 run方法 只是简单的放大调用，不是开启一个新的线程去处理

1. 并发的风险
   1. Cpu频繁线程上下文切换 ，导致性能低
   2. 锁的问题 ： 出现了 死锁、活锁、饥饿问题 导致业务代码不正常执行
   3. 线程是否安全（多线程操作临界资源是否会导致数据的不一致）
2. 并发和并行的区别
   1. 并发 ： cpu 多核状态下 交替执行一个任务 ，多线程执行一任务
   2. 并行 ： 同一时间 有多个执行或程序 同时运行
3. 什么是 jmm 抽象模型 （主存 和 各自线程的内存）

作用：让jvm在各个操作系统上运行无差异化



1. JMM 问题（多线程下不可见问题）
   1. 多线程操作同一变量时 会将主存中的变量备份到各自的工作空间，如果这个变量不具有可见性(没被Volalilte 修饰或者没有Synchronize ) 那么如果线程A 改变了变量， 线程B 感知不到 B线程会一直使用缓存中之前同步的这个变量的值。当B使用一段时间这个变量后,如果再次需要用到这个变量时 会到自己的缓存中重新加载一次这个变量。假如这个缓存过期了 才会到 主存中读取，此时才发现数据发生了变化

注： 缓存失效的时间 大概是 20 毫秒 ，如果方法执行的时间大于这个值，再次从缓存中去读值时 ，缓存就会失效 然后从主存中读取

如果方法执行的时间小于20毫秒， 会直接从缓存取 ，然后刷新缓存时间



如果在 while { } 中加上 system.out.print() 由于 print()方法是被sync修饰

的具有可见性并且会影响整个方法体

1. 什么是伪共享

cpu是有一级缓存、二级缓存、3级缓存的 。当多线层频繁的操作公共的数据 ，这些数据回频繁的发生变化 从而导致各个线程中缓存的数据失效只能从主内存中读取数据，从而导致性能下降 这种现象就是伪共享

(1)如何 解决伪共享 ，提高程序的运行效率

* + - 1. 缓存行填充
      2. 使用@sun.misc.Contended 注解（java8） 在juc包下使用较多

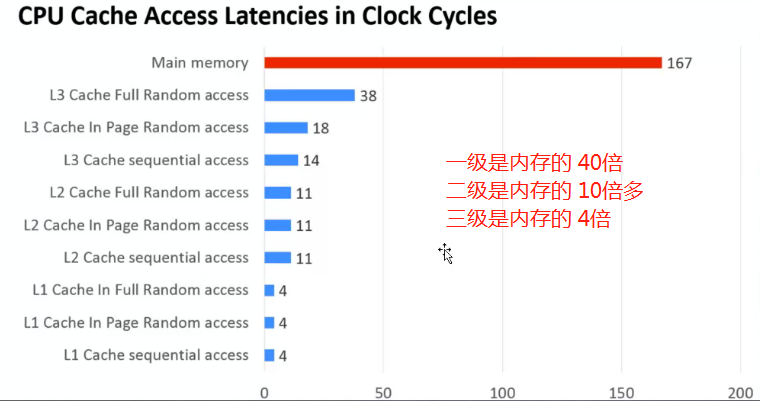
注意： 添加 jvm参数-XX:-RestrictContended 才起作用

1. 指令的重排序问题 （单例 DCL模式在创建对象时就会出现重排序问题）
   1. 什么是重排序

Jvm 在JIT即时编译阶段为了提高性能会将我们写的代码重排序，但是不影响单线程下执行的结果 即 ：as-if-serial 语音原则

|  |
| --- |
| Int a = 1;  Int b = 2 ;  a = a + 1;  如果代码的执行顺序是上面的方式 ：先从内存中加载a，再加载先从内存中加载b。然后再从内存加载a ，+1 将2复制给 a 。 如果进行指令重排少加载一次a优化  Int a = 1;  a = a + 1;  Int b = 2 ;  先从内存加载a ，然后从cpu缓存中 直接拿a进行 +1，再从内存加载b |

* 1. Cpu 从缓存中加载数据 和 从内存中加载数据比较



关于性能的问题总结：

1. MESI 的性能高但是存在缓存伪共享 ：通过@sun.misc.Contended 注解不让 cpu缓存失效让性能更高
2. 指令重排：Jvm 在jit阶段重排序提高程序的性能，但是在多线程下导致数据 的不一致 ：通过 voiletor、final、lock、sync、UnSafe中的内存屏障解决重排
3. Happens-before 原则(即 可见性，Voiltale 是可见性的实现)
4. 由于线程启动的顺序并不能决定数据读写的顺序多线程操作同一份数据时，一个 写，一个读 。可能会存在后面的线程读完了 前面的线程还没写。这样就会出现数据的 不一致。因此为这份数据在多线程之间必须是可见的才能保证数据的安全 ，这就是 happens-before 原则 因此出现了 Voiltale 关键字
5. 也可以根据 happens-before 原则来判定我们写的多程序是否是安全的(共8种)
6. 程序执行：多线程下读一定要在写之后执行
7. 锁定规则：unlock 一定要在 lock之后执行；
8. volatile变量规则：变量读操作 一定 在写操作之后
9. 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操 作A先行发生于操作C；

5.线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作；

6.线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生；

7.线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行；

8.对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始；

1. 内存屏障 ： 是底层防止指令重排的实现方式
2. **为什么需要 内存屏障 ？**

为了防止指令重排 在程序jit即时编译阶段为了优化性能，在不影响单线程 执行结果的前提下， 会对编译后的代码进行指令重拍

1. **指令重拍的问题**

在单线程下进行指令重拍虽然不影响程序执行的结果，但是在多线程下 由于线程执行的先后顺序是随机 ， 指令重拍可能会引起数据的不一致 。 因此在开发多线程的程序时 ，要禁止指令重拍

1. **实现的方式3种**
   * 1. 读屏障 和 写屏障
2. Storestore 读读
3. soreLoad 读写(在 x86 处理器下，只使用这一种 其他3种无效)
4. loadStore 读写
5. loadLoad 写写

手动添加 ：通过 Unsafa 类的中的 sotreFence() 方法 ，就能实现和volitale 一样的效果

* + 1. Volitale 中的Lock前缀（open jdk中的方法） ，实现有序性不是底层不是通过内存屏障，而是通过对cpu总线加锁和对cpu中的缓存加锁。 （volitale 内部防止指令重拍就是使用 lock锁，起到了内存屏障的作用）
    2. Synchronize 实现的内存屏障

1. **内存屏障的作用范围**

sotreFence() 方法 前后的的指令可以防止重排序

1. CPU MESI 缓存一致性协议

总线加锁 ： 早之前是通过对cpu总线加锁 来确保数据的一致性但是性能低

mesi作用 ： 通过对缓存加锁保证多核cpu 各自缓存具有一致性，不让系统数据发生混乱。从而既兼顾了性能、有保证的 数据的一致性，但是mesi 也有性能问题就是会存在伪共享，但是比总线加锁的性能高，总线加锁是让多核cpu串行执行 ，不能让cpu并行

Lock锁 ： 通过对总线加锁 和 缓存加锁

1. Volalilte

保证可见性、有序性 （都是由内部的 lock实现）

1. 可见性 : lock刷新数据到主存
2. 有序性 ： 内部通过lock前缀防止指令重拍

**Unsafe**

1. Unsafe类 (lock、synchronize 底层依赖类)

java不能直接访问操作系统底层，而是通过本地方法来访问。Unsafe类提供了硬件级别 的原子操作（基本都是 native方法） 包括 数组操作、对象操作、CAS操作、线程park 操作、unPark、另外也是 juc 并发包的底层基础实现。

(1)Info相关

主要返回某些低级别的内存信息：addressSize(), pageSize()

(2)Objects相关

主要提供Object和它的域操纵方法：allocateInstance(),objectFieldOffset()

(3)Class相关

主要提供Class和它的静态域操纵方法： staticFieldOffset(),defineClass(),defineAnonymousClass(),ensureClassInitialized()

(4)Arrays相关。数组操纵方法：arrayBaseOffset(),arrayIndexScale()

(5)Synchronization相关。

主要提供低级别同步原语：

monitorEnter(),tryMonitorEnter(),monitorExit(),compareAndSwapInt(),putOrderedInt()

(6)Memory相关。直接内存访问方法（绕过JVM堆直接操纵本地内存）： allocateMemory(),copyMemory(),freeMemory(),getAddress(),getInt(),putInt()

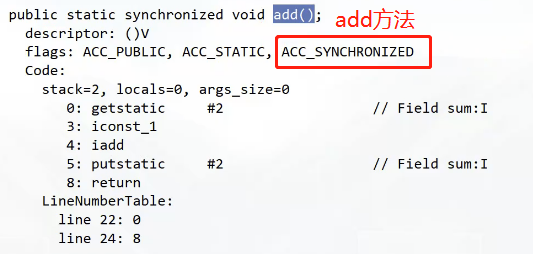
**synchronize**

1. 互斥性

锁具有互斥性 表示被加锁的资源同时只能被一个线程持有

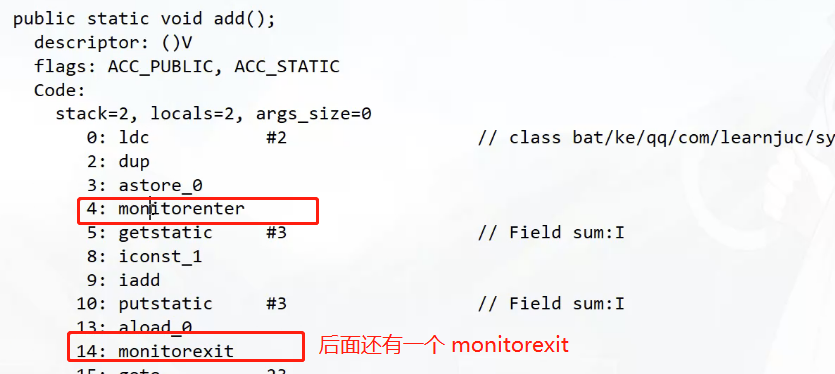
1. 使用
   1. 静态方法上加锁
   2. 方法上加锁

字节码指令阶段 ：方法的flags 属性上加 acc\_synchronize 属性 进而对方法加锁，静态方法也是一样的



* 1. Synchronize 代码块被编译后的信息

编译的字节码 在加上 monitorEnter 和 monitorExit 指令 ，并且monitorExit 指 令有2个 ，后面一个用于异常跳出

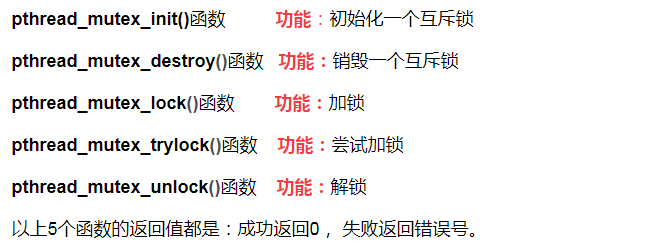


总结 ： synchronize被编译成acc\_synchronize、monitorEnter、monitorExit指令， 在每一次上锁指令 有时由monitorEnter触发的 ，**轻量级 和 重量级锁 锁释放锁**

**是由monitorExit触发的**

* 1. 重量级锁的底层实现

重量级锁的acc\_synchronize、monitorEnter、monitorExit指 令后底层加锁是通过 linux 操作系统函数mutex 进行加锁的



1. 重量级锁上锁的底层实现

重量级锁的实ObjectMinitor现 和 aqs的实现有些类似 ， 都遵循管程模型

就是通过队列的方式来实现对多个线程竞争锁的问题

什么是管程 ？

管理共享变量 以及共享变量的操作过程，让这些变量支持并发 ，重量级锁的管程对象是 ObjectMinitor 对象

当多线程争抢同步资源时，synchronize锁会升级成重量级锁生成的字节码指令是 acc\_synchronize、monitorEnter、monitorExit。 在open jdk源码中维护了线程争抢锁的相关信息类 ObjectMinitor类 包括了

当前持有锁的线程owner

锁获取的次数

重入次数

阻塞线程对 EntrySet(队列存放未争抢到锁被阻塞的线程)

竞争锁的队列(**存放执行notifyall/notify后被唤醒的线程 。notifyall/notify后阻塞队列的线程和 waite队列的线程都会进入此队列** )

等待唤醒对列waitSet(存放被wait的线程)等

**重量级锁的上锁过程 ：**

当是重量级锁时 ，markword保存的是ObjectMinitor对象的内存地址，以及锁状态时10。 当线程争抢锁时 先通过cas 判断 当前的 ObjectMinitor类中 thread属性是否是null ，如果是null 表示没有线程持有锁，于是让

ObjectMinitor类中 thread 等于**争抢锁的线程或从队列出队的线程**，表示上锁成功。

当然真正持有锁的操作是执行pthread\_mutex\_lock函数 ，如果返回 0 表示当前线程锁持有成功 。 接着进行可重入判断 就是判断ObjectMinitor类中 thread 是否是当前线程 ，如果是就进行重入 可重入属性 + 1

如果 线程对象不等null ，表示被其他线程持有着呢 ，**然后就将当前线程 封装成 waiter节点 ，入队至阻塞队列，入队后线程就把自己挂起等待被其他线程唤醒** 。

**重量级锁的释放锁过程 ：**

将ObjectMinitor的 thrand属性置为null ，接着调用底层的pthread\_mutex\_unlock释放 锁。 然后检查 竞争队列 和 阻塞队列中是否有等待锁的线程 。如果有 就根据不同 的**唤醒策略** 选择一个线程来持有锁

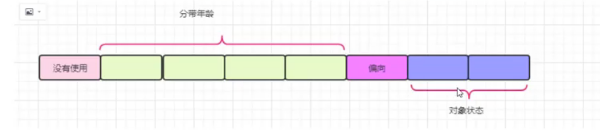
**重量级：**不是公平锁 ， 在释放锁后 ，和检查队列中是否有等待线程之前 ， 可能会 有新的线程进来(跟aqs 非公平锁的实现一样)

1. 当多线程争抢对象锁时 ，如何判断对象锁是否被线程持有

**对象头中 markWord ， 64位机 占64位** ，**通过后3位来判断锁的状态**

前25位不使用 被 01 填充 **+** 后31位 保存 hashCode **+** 1位不使用**+**

4位存对象的age(垃圾回收) **+** 1位 偏向锁标志 **+** 2位 对象状态



1. **64位机下对象头markword 信息详情**

前25位不使用

无锁状态下往后31位 ： 存放对象的 hashcode

偏向锁状态下往后的31位 ： 存当前线程id

轻量级锁状态下 往后31位存 指向线程栈中锁记录的内存地址

重量级锁状态下 往后31位存 ObjectMonitor对象的内存地址

倒数第3位 ： 1 偏向 0不偏向 **偏向锁 101**

倒数后两位 ：  **001 无锁、 010 重量级锁 、 轻量级锁 000**

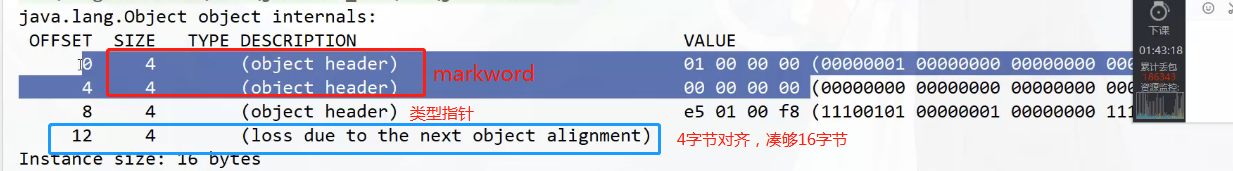


1. .无锁存 对象的hashcode ， 分代年龄 **01**
2. .偏向锁 存线程id 和 分代年龄 然后偏向为 **101**  
   (2) .轻量级锁存 线程栈的lock record内存地址 （轻量级锁的） ，无分代年龄 **00**
3. .重量级锁 存指向monitor地址 **10**

轻量级锁重量级锁的hashCode存在与什么地方？

答案：线程栈中，轻量级锁的LookRecord中，或是代表重量级锁的ObjectMonitor的成员中

**B、如何看对象头信息 ： java object layout 包**



1. **偏向锁自己是不能重偏向的自己无法撤销只有能被其他线程撤销然后重偏向**

偏向锁升级成轻量级锁， 需要把偏向撤销 ， 而当前线程不能执行完任务后 不会撤销偏向锁，需要其他线程把偏向锁撤销

1. **锁升级过程**

当有一个线程执行或者交替执行synchronize时是只有偏向锁的 ，如果存在线程交替执行且**存在短时间竞争**才会升级成轻量级锁，不会升级成重量级锁

偏向 ： 单线程执行 或 多线程交替执行

轻量 ： 多线程存在短时间竞争锁 （自适应自旋成功）

重量 ： 多线程激烈的竞争锁

1. **无锁状态 ：** 当不是多线程竞争的时候只有一个线程访问synchronize同步块时 。 首先执行monitorEnter指令， 执行这个指令前会先当前线程信息保 存至当前线程栈中 。接着检查对当前对象是否是偏向状态 和 象 头markword中的 thread id 是否存在，如果不存在则通过cas将自 己的线程id添加至markword中升级未偏向锁 ，偏向状态置为 1， 锁状态01
2. **偏向锁 :** 升级为偏向锁后，接着执行 synchronize同步块中的业务代码 ，如 果在执行业务代码块的过程中，存在线程2也访问同步块争抢锁 时 ，线程2会检查markword存放的线程id是否存在线程id----> 存在 ，说明锁已经被其他线程持有， 则线程2 开始**撤销偏向锁**，

撤销时不是立即撤销，而是提交一个撤销偏向锁的一个任务存放至队列中 当这个任务执行时，先判断 线程1是否到达安全点 (endpoint)，如果到达安全点才能撤销成功

撤销后 判断当前锁是否是偏向自己 ，此时偏向锁肯定是偏向线程 1的 ，然后cas设置成自己的thread id。 如果cas成功

就可以争抢到锁(此时还是偏向锁)

(如果1 执行完释放锁 ， 2也要进项撤销偏向锁的操作)

如果线如果cas不成功则线程1并没有释放锁，锁将升级成轻量级锁。 遍历线程栈中的加锁记录 ，拿到初次加锁的记录 ，然后通过cas修改这条记录并对象的markword中的线程锁记录存至线程栈 ，然后mardword存放存线程栈中lock record指针 ，并将对象置为不偏向 0 ， 对象锁状态置为 00

什么是安全点(safe point) ： 到达安全点 是指除了vm线程 jvm中其他 的线程都处于阻塞状态，不能执行任务。 只有VM线程可以执行任务, 如 gc线程 和撤销偏向锁线程

为什么到达安全点才执行偏向锁撤销 ？

因为在撤销过程中会往线程栈中读写数据 ，到达安全点停止其他线程往 线程栈中读写数据 ，来确保数据的一致性 避免并发问题

如果是一个线程 多次重入同步块 ，当线程第二次尝试获取锁时 先判断当 前锁是否是偏向自己 ， 肯定是偏向自己的， 直接执行代码块 。 这样大大提高 了效率

锁重入数据的保存 ： 锁每次在cas尝试获取锁前 ，都会将当前锁记录插入当前 线程栈内

判断锁状态--->发现是偏向锁

撤销偏向锁---> 撤销之前先判断之前线程是否到达安全点--> 到达安全点才能撤销

撤销后--->判断是否偏向自己-->偏向自己直接重入

不偏向自己---> cas修改线程id -->

cas修改成功当前线程持有锁 偏向自己

cas修改不成功 升级轻量级锁

1. Sync关键字的 批量重偏向

100个对象 再被第一个线程访问时 ，那么都是偏向第一个线程的 。当第二个线程访问这100个对象时 ， 会依次将之前的偏向锁撤销，重新升级轻量锁 当撤销到20个时 ，就不再升级轻量锁 剩下的80个对象就会发生批量重偏向第二个线程 。 （目的提高性能）

A、刚开线程交替执行 无锁---->偏向--->轻量---->(长时间竞争的前提下)重量级

1. 刚开始就多线程同时竞争synchronize代码块时，不会升级成偏向锁 ，而是

无锁---->轻量级锁 -----> 重量级

1. **轻量级锁：** 升级成轻量级锁后， 如果存在多个线程同时竞争代码块的情况下， 且存在竞争时间较长。 假如又来了多个线程，势必有线程争抢不到 锁， 争抢不到锁的线程会自适应自旋， 当自旋到一定次数能争抢到 线程，则新的线程持有锁。如果自旋后扔不能获取锁则会升级成重量 级锁 ，升级成重量级锁之前先获取轻量级锁的状态 和 空的 ObjectMinitor对象 ，然后通过cas修改锁状态为 膨胀中 。如果cas 修改锁状态为膨胀中失败，则说明其他线程也在争抢锁导致膨胀 ， 或者膨胀结束。膨胀结束后 ，将 ObjectMinitor的线程置为当前线程。

然后调用 mutex\_lock持有锁 ，然后

cas修改markword中重量级锁的 monitor指针地址 ，将做 状态置 为 10

**轻量级锁** 决定要不要升级成 重量级锁 ，是根据自适应自旋的结果 来 判断的

自适应自旋 ： 早期无自适应 ，就是10次 ，后来优化成自适应自旋 ，线程争抢锁时 会根据上次自旋成功后的次数来判断，当前自旋的次数 ，假如上次自 旋成功了 ，这次可能成功的几率会很大 将增加自旋次数，来确保自旋 成功 （也可以设置jvm参数自定义）

在升级成重量级锁之前 ：在多线程挣抢锁的的整个过程中，都没有去阻塞线程，而是通过自旋然线程争抢锁，这样就避免了 阻塞线程，唤醒线程 从而优化了性能

轻量级锁释放锁逻辑 ： 通过 monitorExeit 指令。

**膨胀成重量级锁的3种情况 ：**

1. . 在 偏向锁 或者 轻量级锁有线程调用 对象的 hashCode()方法 ，膨胀成重量 级锁。

**原因：**由于原来保存hashCode的位置保存偏向锁的线程id的值或者保存了轻 量级锁的 lock record 的内存地址 ，现在不能保存hashcode

1. . 一旦调用 wait方法， 会立即膨胀成重量级锁

原因 ： 因为锁在偏向或轻量状态时 ，是不存在 ObjectMinitor对象的，没有 ObjectMinitor对象 就没办法保存 被wait的线程。因此会膨胀成重量级锁

1. 轻量级锁的并发竞争 且 自适应自旋不成功
2. **重量级锁：** 升级成轻量级锁后。当有新线程争抢且获取不到锁 ，就直接阻塞线 程，当持有锁的线程执行完代码块，释放锁后 则唤醒其他阻塞线程 重新争抢锁
3. 重量级锁撤销

重量级锁的撤销是 gc来触发的 ，当gc时 会判断当前对象锁状态是否是中重量级锁，如果是则判断 当前对象处理gc线程操作 ，是否有其他线程操作 。如果有不撤销，如果没有则撤销 置为 无锁

对象如果被锁持有 就是 gc root

1. 为什么 synchronize 是非公平竞争的方式获取锁，并且在唤醒时不能精确的唤醒某一个 ?

synchronize 是基于对象的 ，不是基于线程的 lock 锁是基于线程的

1. 偏向锁 并没有实现线程的互斥 ， 而是轻量锁实现的 ，为什么还引入偏向锁 ？

因为在真实的项目中， 大多数的并发场景 线程都是交替执行的 存在线程激烈竞争的

场景并不多 ， 所以使用偏向锁可以 提高获取锁的性能

1. **偏向锁执行完同步块 不会释放 锁 ， 轻量级会释放锁**
2. Wait() 方法
   1. 让当前线程释放锁，notify()方法必须在wait之后使用
   2. 如果有使用 wait 方法 ，则直接无锁升级重量级锁
   3. 另外wait只能和 synchronize配 合使用否则会报错
   4. 释放、唤醒线程的方法
      * 1. Wait notify

直接基于 pthread\_mutex\_lock、 pthread\_mutex\_unlock

* + - 1. Support类的 Support.park() Support.unpark()

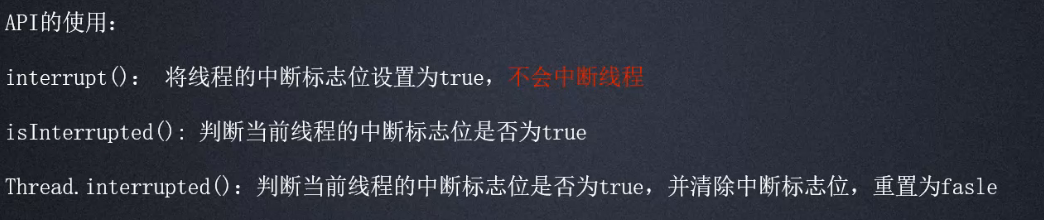
特点 ： 无先后顺序， unpark 也可以在 park之前执行

* + - 1. Condition类的 wait() / signal() (基于park unpark)

Juc包下所有的 阻塞唤醒都是基于 park() 、unpark()

1. 如何优雅的关闭线程
2. Thread.Stop () 不能使用，会立即关闭线程 可能当前线程刚持有锁，就被stop 会出现死锁

(2) 线程中断 Thread.interruppter() 可以使用 (使用时分为 清除中断信号 和 不清楚中断信号2种)



当执行完t.interruppter() 后，当前线程不会停止，只会设置中断信号 == true。设置后，手动使用代码跳出任务(如 return，break循环)， 让其处于空闲之后线程才会被停止 或者

1. 通过Thread.interruppted() 方法校验是否 == true ，并且清除t.interruppter() 设置的中断信号。
2. 通过Thread.isInterruppted()方法校验是否== true不会清除设置的中断信号

|  |
| --- |
| **使用 ：Thread.interruppted()情况**  t.interruppter(); //将线程的中断 信号置为 true  While(true) {  If(Thread.interruppted()) { //执行任务的线程识别到 为true时,  break;  }  } |

当被sleep期间 ，也能接收到中断信号 因此可以通过中断打断sleep ，被中断后并抛一个异常

wait(5000)、Support.park() 也可以感知中断信号，被中断的并抛中断异常

**lock**

1. CAS 比较与交换 保证线程安全的无锁算法 （乐观锁）

用于保证变量具有线程安全的场景 如AutomicIntnger 、ReentraLock 的state属性， 即 拿期望值 与 待修改的变量值进行比较，如果等于则将变量值修改成新的值，如果不等于则一直自旋

注意：

1. 变量值必须具有可见性通过底层的lock实现 ，由于使用了lock 所以具有可见 性、有序性、原子性 因此 CAS 就可以实现锁 来确保线程安全
2. CAS 只能保证一个变量是线程安全的

Aba问题 ：通过版本号解决

1. Lock的作用
   1. 原子性：加锁
   2. 有序性：防止指令重拍(lock 防止指令重拍的实现不是通过内存屏障)
   3. 可见性：将更改的缓冲区数据刷新到主存
2. Lock接口方法

void lock(); 拿到对象的锁 ，如果该锁被其他线程持有 则等待

使用lock() 获取锁后 ，在释放锁时如果发生异常则不会释放 因此需要放到 Finally代码 避免死锁情况发生

void lockInterruptibly() ; 获取锁 ：当两个线程A/B使用lockInterruptibly 获取锁时 ，假如A获取到了锁 ，线程B就处于等待状态 调用自己的 lockInterruptibly方法能终止等待

而synchronized修饰的话，在等待获取锁时 只能一直等下去 直到获取到锁

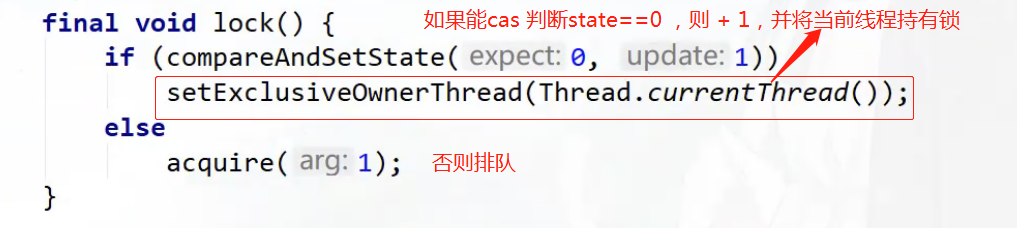
boolean tryLock(); 尝试获取锁如果成功返回true，如果是失败返回false

boolean tryLock(long time, TimeUnit unit) 尝试获取锁如果陈功返回true ，如 果获取不到 则等待一段时间，成功返回true 失败返回false

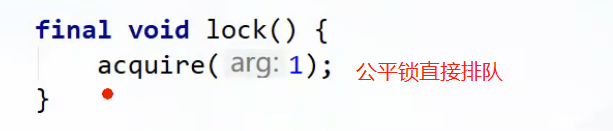
void unlock(); 释放锁

Condition newCondition();

1. AQS
2. 非公平锁



1. 公平锁

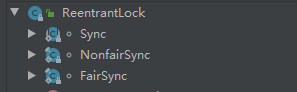


1. ReentrantLock 的1个属性 ， 3个内部类
2. 1个属性

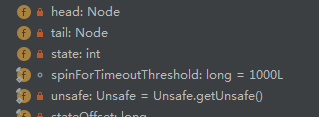


只有一个 sync 属性，即 AQS 类

1. 3个内部类 ，sync继承AQS /公平AQS/非公平AQS



1. . AQS 类 ：AbstractQueuedSynchronizer 重要属性及内部类
   * 1. .属性



Node ： 双向的链表

Head ： aqs的头结点

tail ： aqs链表头节点node

state ： 线程持有锁的状态

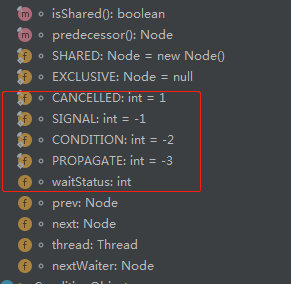
Unsafe ：用于底层对线程操作 唤醒阻塞等

Thread : 当前持有锁的线程对象

* + 1. .内部类



a.Node : 队列（双向链表实现）



Thread 当前未持有锁的线程被封装到node中

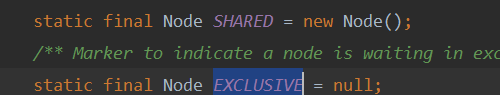
waitStatus保存线程执行任务的各个状态

线程的种类(共享和独占)

独占 本质： node 对象中的 node属性 = null 。

属性等于null ，目的就是让 nextWaiter属性 =null

共享 本质： node 对象中的 node属性 new Node()



prev 前一个node

Next 后一个node

nextWaiter 第一次排队的线程是null

b.ConditionObject ： 实现用于Condition实现线程协作

1. Node 中 waitStatus 表示node 的状态, 有5 种

-3 ：下一次共享状态获取，将无条件传播下去

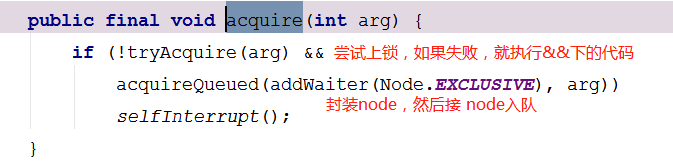
-2 ： 当Condition调用signal唤醒线程时后，该节点将会从等待队列移到同步队列

-1 ： head 节点的状态表示当前线程 的下一个节点可以被唤醒。如果head释放锁 出队，就将下一个 node 置为head 然后 waitStatus = -1

0 ： 入队的节点 waiteStaus的状态 = 0 等待被置为-1 然后持有锁

1 ： 当线程超时 被中断了 ，表示无效的线程节点 。 会在park线程时被删除

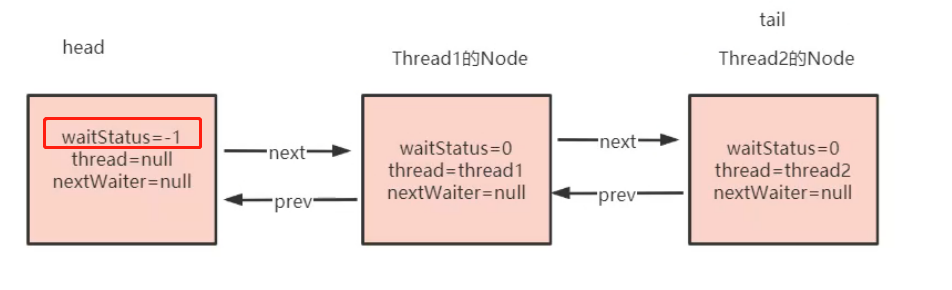
1. **AQS上锁过程**



acquire(1) ---> tryAcquire(1) 尝试上锁和重入判断 不成功 --->

addWaiter(独占node) 封装node，节点进行入队（但是waitStatus = 0 ） --->

acquireQueue(独占node,1) node的waitStatus = -1 表示可以被唤醒入队然后park

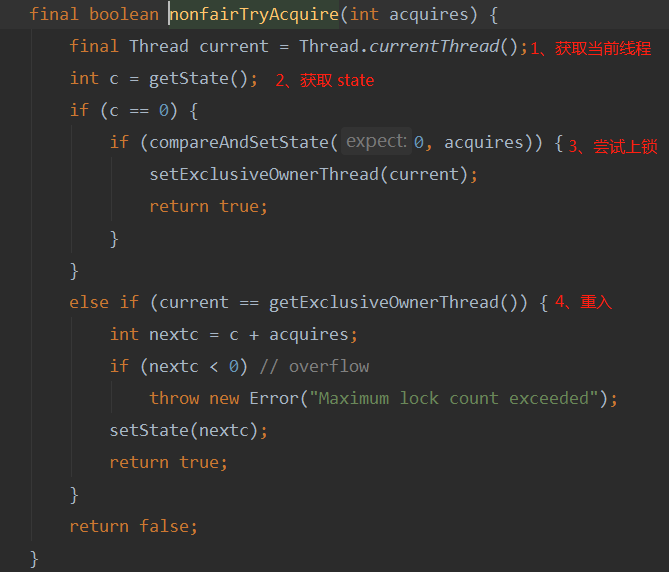


Exclusive [ɪkˈskluːsɪv] ： 独占，专有

**tryAcquire(1) :** 获取state 是否 == 0 ， 如果等于0 通过CAS state = 1，然后让当 前线程持有有锁setExclusiveOwnThread(currentThread); retuern true;

如果 state != 0 ,校验当前持有锁的线程 是否是当前线程 ，如果是旧 进行重入 state +1 ; return true;

如果返回 false ，则进行入队



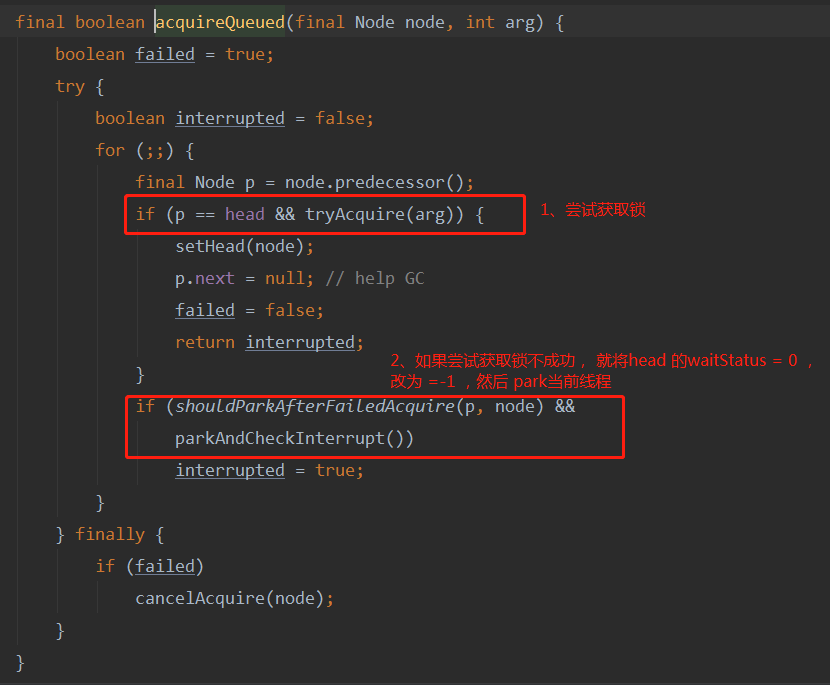
**addWaiter(独占node =null) ：** 创建 node将当前thread存放至node中 ，然后判断 node的pre节点是否存在 ，如果不存在 说明没有 node队列 ，通过for(;;) 先创建 一个 队首 = 队尾的 空node队列 (head中的 thread 永远=null )，然后让当 前node 指向 队首head。进行入队 retuen node;

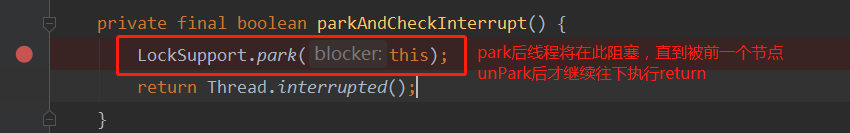
**acquireQueue(独占node=null, 1) ：** for(;;)自旋先判断当前的 node的pre是否是 head 如 果是则说明当前 node是第二个节点 ，应该 再一次尝 试加锁 (因为 很可能前一个线程已经释放了锁) 。 如果能尝试上锁成功 ，则将当前node置为head node 中的thread 置为null ， 之前的head =null**进行出队**触 发gc回收

也有一种情况，由于是for(;;) 下一个线程被唤醒后继续 在执行for(;;)代码然后尝试获取锁 ，成功后 对之前的 head进行出队

如果不成功 ，获取前一个节点的 waitStatus的值 ， 如果等于0 ，就将head的waitStatus = -1 。 然后返回 true 接着对当前线程进行 park 然 后 线程阻塞 ，

（head节点waitStatus 从0 置为 -1 是通过自旋来完 成的 ， 将 head 的waitStatus 从0置为-1 。目的就 是为了park 线程未竞争到锁的线程 ）





1. **AQS解锁过程**

release(1) ---> tryRelease(1) 尝试释放锁 --->

tryRelease(1) ： 先获取 state 然后 -1 ， 然后判断 state是否 = 0 , 如果 != 0 ,表示重入 那么当前线程就继续持有锁 return false;

如果等于 0 表示线程释放锁

接着获取对列的head节点 ， 如果当前head节点wiatStatus != 0

（此时是 等于-1 ） 就将将当前 aqs中的线程置为null 。 接着获 取head节点的下一个节点然后将waitStatus置为 -1 ，head的 waitStatus = 0 然后获取下一个节点进行unpark 。 当将下一个node unpark后 ，然后阻塞的线程就会继续执行**acquireQueue(node) for(;;)**入队的代码 接续判断是否是第二个节点，是就**tryAcquire(1)** 尝试获取锁 ，然后加锁成 功。当前head 置为 null 。进行出队

1. 公平锁 与 非公平锁的区别
   1. 公平锁 ：有线程持有资源， 后面的线程直接 调用acquire(1) 进行入队
   2. 非公平锁: 当前竞争资源的线程 回先通过 cas判断 state是否等于 0 ，如果等于0 可以直接插队持有锁 ，！=0 则入队
2. 为什么 head 节点中 thread 永远为 null

== null 表示当前node 是持有锁的线程 ， 线程被设置成 AQS 中的thread属性

1. 线程在每次竞争锁时 ， 都会tryAcquire () 方法 ，如果当前节点在入队时是第二个node就进行两次tryAcquire () 上锁

第一次尝试上锁 刚开始 通过cas判断 state 不等于 0 时 ，会进行尝试上锁 。 如果尝试上锁成功就持有锁 ，如果不成功 就将当前线程封装成 node节点进行入队操作 。 在入队时 也判断当前节点是否是 第二个节点 ，如果是第二个节点也会有个尝试上锁的操作 ，如果不是第二个节点就park

1. 每次线程竞争不到锁入队后 ，都会将前一个node节点的waitStatus 状态置为-1，然后 park阻塞 一直阻塞在执行park()方法的地方 。直到被前一个线程唤醒时才开始尝试加 锁
2. Head节点 一直都是获取锁的那个线程节点，并且 head 节点的thread 永远为null ，

当前获取锁的线程是被放置在 aqs的exclusiveOwnerThread中

1. 每次线程unlock时，都会unpark后一个线程去争抢锁 ，后一个线程尝试加锁，

接着head的waitState==0 head节点，一直都是head节点出队置为null ，然后将下一 个节点置为head

1. 独占锁和共享锁的区别

独占锁 head节点只能唤醒后面的一个node节点 ，**共享锁可以唤醒head后面的多个node节点**

独占锁 使用的waitStatus 使用到的状态 0 和 -1

1. 两种锁对别



当线程较多且竞争资源比较激烈的场景 合适用lock 锁。如果使用synchronize会使用 重量级锁

1. CountDownLatch 应用场景: 主线程阻塞 其他线程准备好主线程再往下执行的场景

定义成全局：CountDownLatch countDownLatch = **new** CountDownLatch(5);

方法使用 ：

1. await() 阻塞当前线程
2. countDown() 对state -1 ,直到等于 0 。被阻塞主线程才会往下执行

特点 ： 一次性执行完 ，不能复用

1. CyclicBarrier ( 栅栏) 应用场景 ：线程一组一组的执行

定义成全局：Cyclicbarrier cyclicbarrier = new Cyclicbarrier (5);

调用cyclicbarrier .await() 5个线程相互等待， 知道都准备好才一起往下执行

如果在来线程，也要凑个5个线程 ，不满5个将阻塞 。 当执行后 state会重置 然 后来的线程会继续等待，等凑够5个

方法使用 ：cyclicbarrier .await() 阻塞当前线程 ，当被阻塞线程个数 = 5 ，则一起执行

特点 ： 循环复用

1. . Semaphore [ˈseməfɔːr] 信号量 应用场景 ：控制一个资源 最多可以被少个线程访问

Semaphore semaphore = new Semaphore (5);

方法使用 ：

1. semaphore.acquire() 允许当前线程能够访问
2. Semaphore.release() 让当前线程释放资源

new Semaphore (5); 表示最多有5个线程能同时访问资源 ，如果先占有资源的线程没有 release释放资源 ，其他线程则一直等待 。 如果有一个线程释放了资源， 那么也只能有1个线程去 acquire() 去占有资源

**线程**

1. 用户线程 和 守护线程的区别

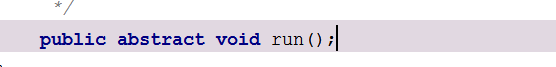
用户线程 ： 执行业务代码运行的线程

守护线程 ： 在程序运行时 守护线程 在 用户线程之后 退出 ， 可以理解为用户线 程的守护者 。 当用户线程不存在 被销毁 了 ， 守护线程也会销毁退 出

1. Java 创建线程的本质

无论是是 Thread 类 、线程池 还是 callAble获取线程返回值得线程本质都是Runnable接口来创建线程

1. Runnanle 、CallAble 、Future 接口
   1. Runnable 创建一个没有返回值的线程

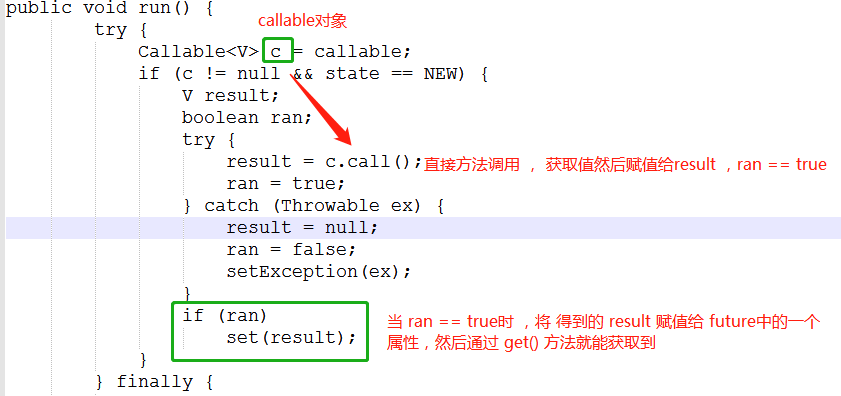


* 1. Callable 创建一个带返回值的线程，内部还是使用 Runnable接口



* 1. Future 依赖 Callable对象 ,获取线程执行的返回值

在创建Future时依赖自己创建的Callable 对象， Future 内部也实现了 Runnable 的run方法。当线 程启动时只会调用run方法， 而我们在实现Callable接口时并没有重写 run 因此内部回到用 Future实现类的run方法。由于Future又依赖Callable对象 ，就能获取Callable对象 ，然后通过方法调用callable.call() 方法 获取到值 。 然后将获取到的值 赋值给 Future 的属性 ，然后通过 get方法就能返回。



* 1. call 和 run 的区别 ： （一个新线程调用，一个是普通的方法调用）

1. . run 方法 ： 是创建线程后 ，新的线程调用 run方法
2. . call 方法 ： call 方法是在 run 方法内部调用的方法



public interface Future<V> {

//返回当前任务的执行结果 ，如果任务还未执行完 则此方法阻塞等待结果 然后才往下执行

V get() ;

//在设置时间内 返回执行结果

V get(long timeout, TimeUnit unit);

//停止正在运行的任务（通过mayInterruptIfRunning判断），如果任务停止成功 则返回true ，如果不能停止、任务已经执行结束、任务已停成功 返回false

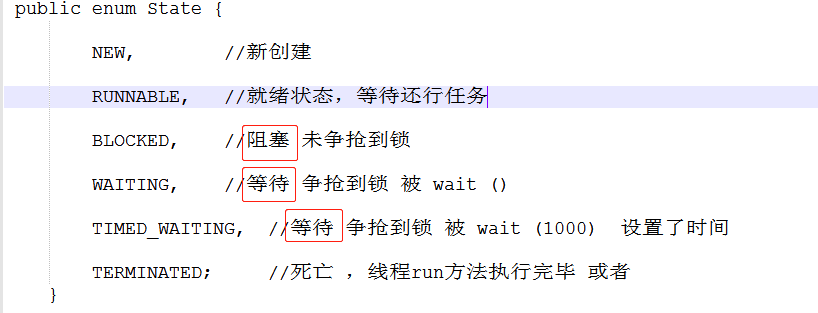
boolean cancel(boolean mayInterruptIfRunning);

boolean isCancelled(); //判断当前任务是否已经取消

boolean isDone(); //判断当前任务是否执行完毕

}

1. 线程的运行状态 ： Thread 类的 state属性



**继续状态**： notify 、notifyAll、unPark 方法 ，线程被唤醒 就绪继续开始挣抢说

**等待状态**: sleep 、wait、park、join 都是让线程处于 waiting/timed\_waiting,

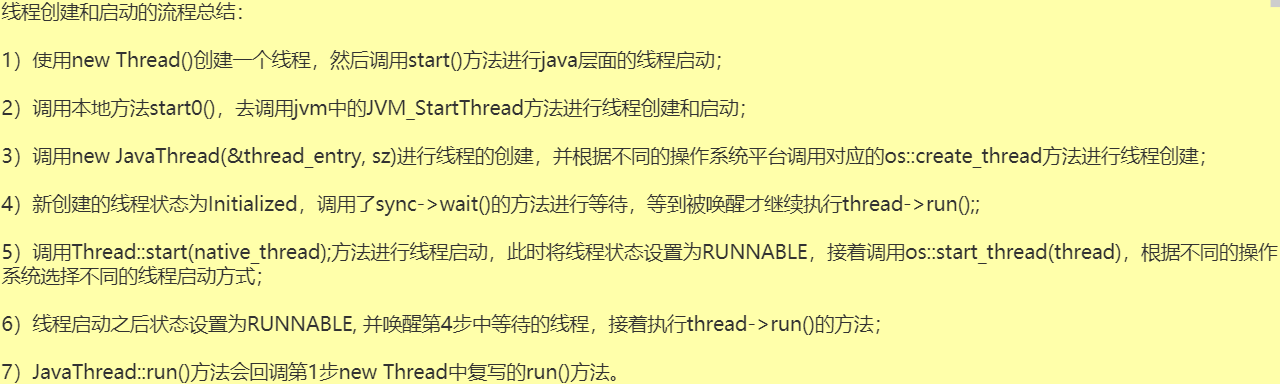
timed\_waiting 是超时等待 ，是设置了时间的

**线程阻塞** ： 未获取到synchronize 同步块 monitor锁

注意：这些被定义的线程是站在程序角度被定义的状态

另外还有 jvm层面 和操作系统层面被定义的状态

1. Thread 类 start() 启动的线程的过程



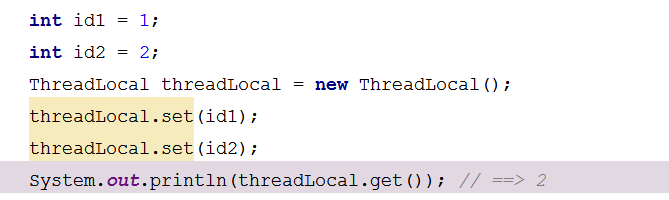
Thread本身就是个java对象 ，当start()时 是调本地方法start0() , 然后调用jvm 的JVM\_StartThread方法创建线程并将创建的thread对象与 StartThread绑定，由于线程是操作系统级别的 进而调用系统的p\_thread方法创建新的线程 。线程被创建为 new 状态 然后后被启动 Runnable状态 ，接着就开始通过 jni调用 run方法

1. ThreadLocal 实现原理 : 定义局部线程变量 只对当前线程可见

特点 ： 一个threadLocal只能存一个对象 ， 如果想存储多条数据 可以封装成一个

如果存多个 会覆盖前一个

因为 threadLocal 只提供了 一个get() 无参方法获取值。



ThreadLocal.ThreadLocalMap threadLocals = null;

A、Spring中dataSource为什么使用threadLocal保存 ？

为了保证连接在线程内部可以， threadLocal能提高性能连接性能不用每次连接数据库 都创建一个新的线程 。 threadLocal一个线程可以为一个用户建立多次连接 来保证线 程间的数据不互相影响

B、spring 中 bean的创建也是用到了 threadLocal

使用 ： 1、set(T value)

2、 get()

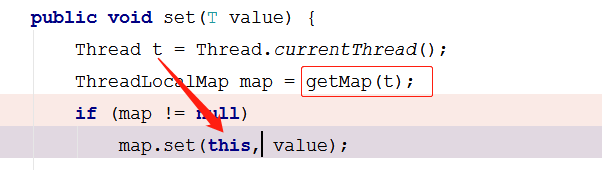
set 是如何 存放值的？

A、在set值时会Thread.currentThread() ; 获取线程

B、然后通过当前线程对象创建一个 ThreadLoacalMap 对象

数组初始化 16 的table ，table中保存 Entry 对象 （Entry 存放 key value）

1. 存放数据时 类似map 计算key的hash值然后通过位运算确定 table的下标



注意 ：使用完后 调用 remove方法 ，防止内存溢出

get取值时 获取当前线程对象 作为key 获取map中对应的value

1. . ThreadLocalMap 的enry静态内部类为什么 继承 **WeakReference** 实现弱引用

为了让gc 回收entry对象。 防止内存溢出

因为 value对象是存在 enry中的 ，当使用 remove方法删除值时 会将

Value对象 = null 。 存放的value对象可以被回收 但是enry还在

因此集成若引用 ，当没有被对象引用时再下次gc时会回收

1. 可能存在 内存溢出的问题

一定 调用remove()方法 ， 不remove可能会内存溢出 Remove方法就是从map 中删除 key 和 value 。

**线程池**

1. Executors Executor ExecutorService ThreadPoolExcutor
   1. Executors 创建各种线程池
2. newSingleThreadExecutor 创建一个单线程化的线程池，使用 LinkedBlockingQueue

缺点 ： 无界队列不可控

1. newFixedThreadPool 创建固定大小的线程池 使用LinkedBlockingQueue

核心线程数 = 最大线程数

执行任务的特点 ： 假如固定线程数是 5 ， 当任务来时 使用这个5个线程执行5个任务。当这 个5个任务全部都完成后 再从队列中出队5个 再一次执行 。

缺点 ： 无界队列不可控

1. newCachedThreadPool创建一个可缓存线程池 使用synchronousQueue

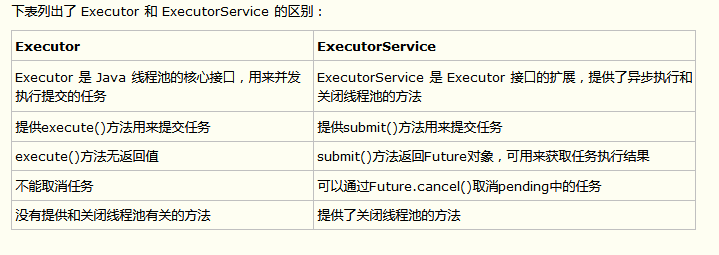
特点 ： 核心线程数是 0 ， 最大线程数是无限大 。 根据任务个数来创建线程去执行 。 执行完毕后时间小于60s线程可自动回收可以复用 大于60秒销毁。

synchronousQueue队列一直都是存放一个任务 入队一个立马出队 ，没有线程就创建线程去执行。

缺点 ： 创建的 线程数不可控

1. newScheduledThreadPool 创建一个定时线程池，支持定时及周期性任务执行。当执行任务时间 大 于 定时间隔时间时 ， 任务会继续执行

特点： 也是来一个任务创建一个线程类似 缓存线程池

1. newWorkStealingPool 偷取任务的线程池
   1. Executor execute接口用于提交任务
   2. ExecutorService 扩展了Executor 除了启动线程池，还扩展了其他操作

ExecutorService 可以批量提交任务 ， submit的各种重载方法

shutdownNow、isShutdown、isTerminated

* 1. ThreadPoolExcutor 创建自定义线程池

new ThreadPoolExecutor(1, 1,0L, TimeUnit.MILLISECONDS,  
 new LinkedBlockingQueue<Runnable>(),  
 threadFactory)

参数1 ： corePooSize 线程池在初始化的后 线程数 （与pool的生命周期相同）

参数2： maxPoolSize 当队列满了以后，线程池开始创建新的线程执行任务

参数3： long keepAliveTime 线程活跃的时间 （规定maxpPool 新创建线程活跃 的时间）

参数4： TimeUnit unit 活跃时间的单位（秒、分、小时）

参数5： BlokingQueue<Runnable> workQueue 存放任务的队列 ，当任务比较多 maxPoolSize 线程执都在执行时， 任务存进队列

参数6： 线程工厂对象 ，可以使用guva线程工厂（使用不传工厂的构造器，那 就使用ThradFactory 内部new thread对象）

参数7：处理器 ： 当线程池达到 饱和状态下 设置拒绝策略

4种拒绝策略（触发时机：队列饱和）

1. CallerRunsPolicy

当触发拒绝策略，只要线程池没有关闭的情况下，max线程也都处于活跃状态 来了一个任务也直接 提交让主线程处理

1. AbortPolicy (默认)

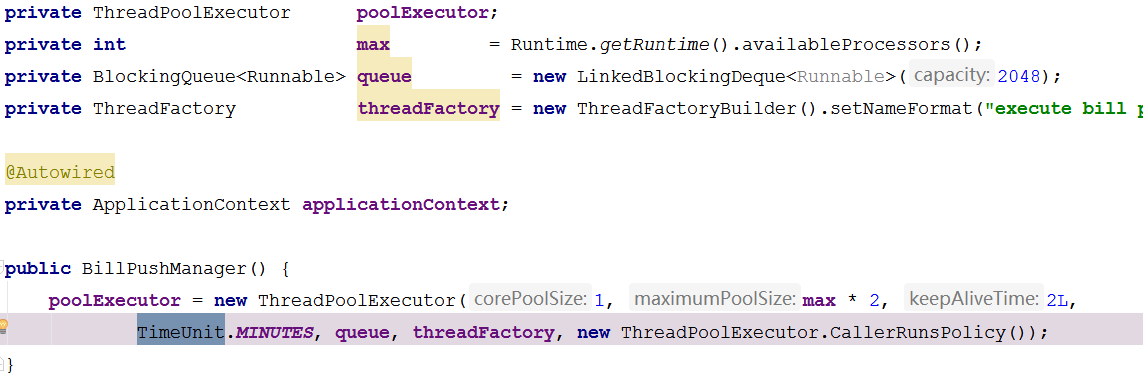
丢弃任务，并抛出拒绝执行 RejectedExecutionException 异常信息。线程池默认的拒绝策略。必须处理好抛出的异常，否则会打断当前的执行流程，影响后续 的任务执行。

1. DiscardPolicy

直接丢弃，其他啥都不做

1. DiscardOldestPolicy

当触发拒绝策略，只要线程池没有关闭的话，丢弃阻塞队列 workQueue 中最老的一个任务，并将新任务加入

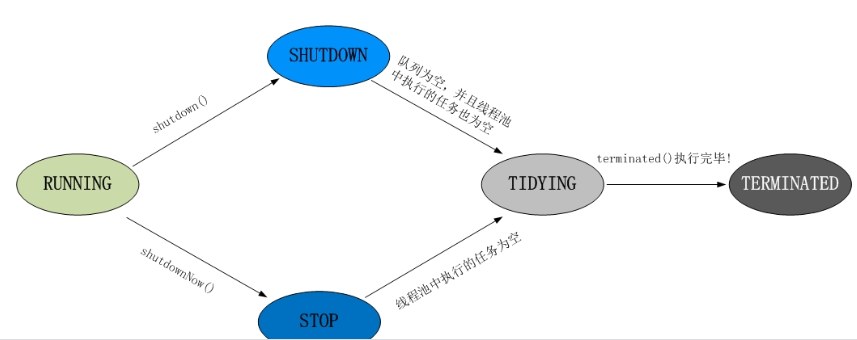


1. ExecutorService 中的 execute 和 submit 的区别

都是提交执行任务 ， execute 类似run 方法 无返回值

Submit 可以获取返回值。 submit内部调用了 execute方法

1. 线程池生命周期



1、RUNNING

(1) 状态说明：线程池处在RUNNING状态时，能够接收新任务，以及对已添加的任务进行处理。  
 (2) 状态切换：线程池的初始化状态是RUNNING。换句话说，线程池被一旦被创建，就处于RUNNING状态，并 且线程池中的任务数为0！

private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));

2、 SHUTDOWN

(1) 状态说明：线程池处在SHUTDOWN状态时，不接收新任务，但能处理已添加的任务。  
 (2) 状态切换：调用线程池的shutdown()接口时，线程池由RUNNING -> SHUTDOWN。

3、STOP

(1) 状态说明：线程池处在STOP状态时，不接收新任务，不处理已添加的任务，并且会中断正在处 理的任务。  
 (2) 状态切换：调用线程池的shutdownNow()接口时，线程池由(RUNNING or SHUTDOWN ) -> STOP。

4、TIDYING（[ˈtaɪdiɪŋ]）

(1) 状态说明：当所有的任务已终止，ctl记录的”任务数量”为0，线程池会变为TIDYING状态。当线 程池变为 TIDYING状态时，会执行钩子函数terminated()。terminated()在ThreadPoolExecutor类中 是空的，若用户 想在线程池变为TIDYING时，进行相应的处理；可以通过重载terminated()函数来实 现。  
 (2) 状态切换：当线程池在SHUTDOWN状态下，阻塞队列为空并且线程池中执行的任务也为空时， 就会由 SHUTDOWN -> TIDYING。当线程池在STOP状态下，线程池中执行的任务为空时，就会由 STOP -> TIDYING。

5、 TERMINATED

(1) 状态说明：线程池彻底终止，就变成TERMINATED状态。  
 (2) 状态切换：线程池处在TIDYING状态时，执行完terminated()之后，就会由 TIDYING -> TERMINATED。

1. ThreadPoolExecutor 线程池提交任务的顺序 和 执行任务的顺序

ThreadPoolExecutor 实现了 ExecutorService 接口。

**为什么线程池分提交 和 执行两个阶段 。**

因为线程池 提交完任务后 ，立即执行任务 然后立即return ； 继续提交下一个任务

类似生产消费者 ， execute方法只专注提交任务 提交后然后return

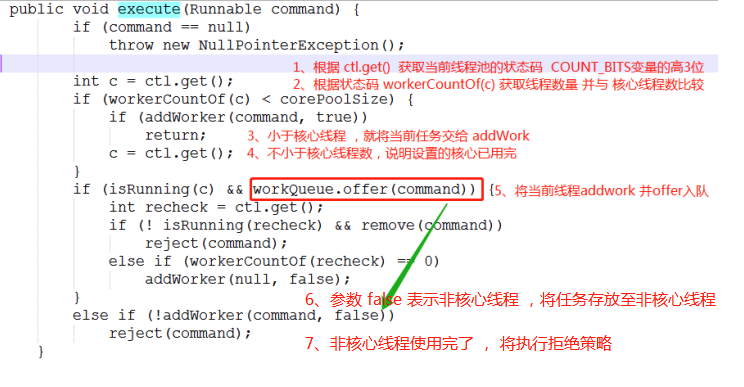
execute 方法中的 addWork消费是处理任务

* 1. **提交任务的顺序 ）**

当任务来临时首先提交给核心线程去执行 当核心线程不够用时就将线程提交到阻塞队列 。 阻塞队如果列满了 就将新的线程交给非核心线程 如果再满了 就触发拒绝策略

核心线程--->阻塞队列 --->非核心线程: 即ThreadPoolExecutor 的execute具体实现

**addWork(task,boolean) 方法是核心，用于内部调用 线程start方法启动线程**



**WorkQueue对**象：阻塞队列

execute提交线程后返回 ， 然后addwork() 添加任务 ，并将任务封装成worker对象添加， 然后执行自己run方法调用runWork 内部有死循环去执行任务处理任务

**addWork()方法** 将任务封装成 work对象，然后将work对象（work类实现了Runnable接口重写了 run方法）添加至HashSet<Worker> **workers 中 ，接着从work中获取线程然后调用自己的start方法启动这个work对象的run方法**

**---->runWork(work) 方法**

**----->拿到 work中的thread 然后 直接 thread.run()进行方法级别的调用 去处理业务**

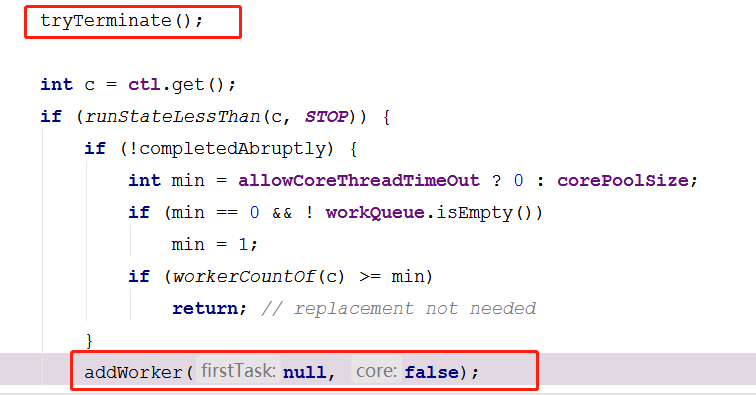
**处理业务前 会有对 当前work 对象中的 task 进行是否为 null的判断操作 ， 由于是创建的新线程去处理任务，任务肯定是不为空的，因此会直接处理任务**

**执行结束后**

**----> 调用 继续将之前的work对象传入 processWorkEixt(work,fase)**

**然后从 workers 删除当前 work对象(set 集合直接remove)**

**任务执行完后， 还会检查线程池状态， 查看线程是否需要关闭**



**----> 再调用 addWork(null,false) 然后再---> 启动这个work对象的run方法**

**---->runWork(work) 方法**

**----->拿到 work中的任务是null ，然后执行getTask() 方法从队列汇中task获取任务**

**------>然后执行 任务的.run() 方法**

**总结： 创建线程数 < 核心线程数 来一个任务就会创建一个新的线程**

**线程的复用是 取 阻塞队列中任务进行复用的**

**不是来一个任务 有空的线程就直接复用处理任务**

**addWork(task,boolean) 方法是核心，用于内部调用 线程start方法启动线程**

**参数1： 任务 实现runable 接口的run方法**

**参数2： boolean类型 ，true 表示使用核心线程接受任务，创建线程然后start()**

**添加任务 ：**

**A、当线程数小于核心线程数时 调用addWork(task,true)**

**B、当核心线程使用完，入队时 调用addWork(null,false)**

**但是当往对列中添加线程时 ， task为null 且是非核心线程**

**C、当队列满了以后 ，workQueue.offer 返回 false后 。 使用最大线程时**

**调用 addWork(task,false) 因此无界队列中的任务 ， 永远不会 被最大线程执行，一直都是被 核心线程数执行**

**另外 核心线程被创建满了 ， 此时来一个任务不是被核心 线程直接执行 而是先入队 然后才被核心线程执行**

**注意 ：任务执行的顺序即 执行addWork() 方法的顺心 ，因为addWork方 法内部会start() 线程 。 虽然 A、B、C 步骤 都使用了 addWork方法， 但是 第二步往队列中添加任务时 ， 执行 addWork task==null 因此内部并 不会start处理任务，因此 处理任务的顺序是 A-->C-->B**

**执行任务 ：**

**addWork()方法** 将任务封装成 work对象，然后将work对象添加至

HashSet<Worker> **workers 中 ，接着从work中获取线程然后start启动这个work对象的run方法**

**---->runWork(work) 方法**

**----->拿到 work中的thread 然后 直接 thread.run()进行方法级别的调用 去处理业务 执行结束后**

**----> 调用 processWorkEixt(work,fase)**

**----> 再调用 addWork(null,false) 然后再---> 启动这个work对象的run方法**

**---->runWork(work) 方法**

**----->拿到 work中的任务是null ，然后执行getTask() 方法从队列汇中task获取任务**

**------>然后执行 任务的.run() 方法**

* 1. **执行任务的顺序**

核心线程---> 非核心线程--->阻塞队列

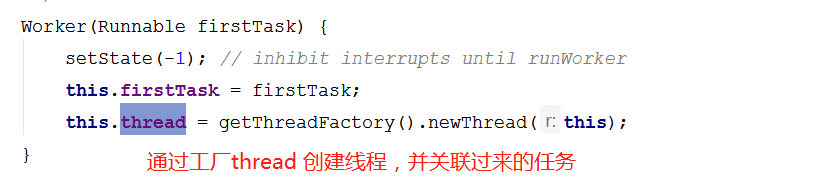
**addwork**(参数1: ，参数2)方法 添加任务封装Worker 对象 ： 实现Runnable 和 AQS

参数1：提交的第一个任务

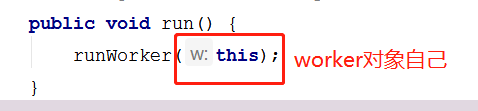
参数2：是否提交到核心线程 true ：核心 false：最大线程 和 提交到队列

**Work 对象**就是将添加的任务关联一个线程封装成work对象

工厂内部也是直接new Thread() 对象

addwork(参数1: ，参数2)中做的事情 ：

1. 自旋判断 线程的状态
2. 将任务封装成 worker对象
3. 然后获取线程 Thread t = work.thread;
4. t.start() 方法执行 work对象的 run方法 ，run方法调用的是runWork()



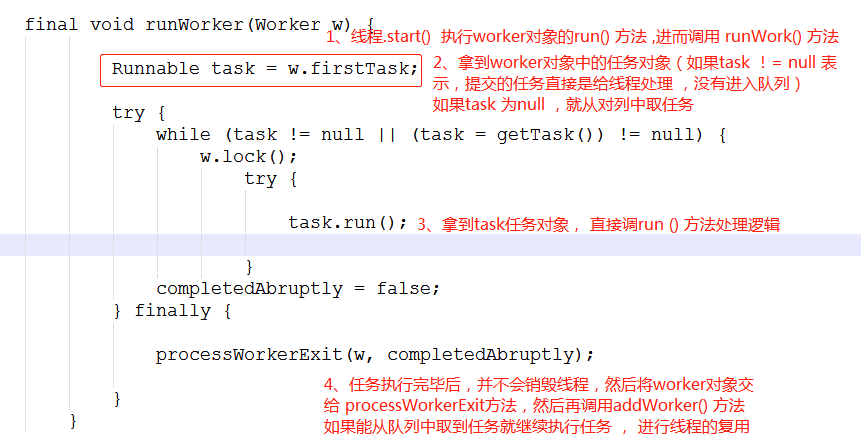
1. runWork(worker) 方法
   1. **死循环 处理任务**

每次先获取 work中的firstTask 任务(**不是第一个任务，应该理解为 关联的这个线程第一次接受的任务，当任务执行完毕 还继续会调用 当前的线程，然后从队列中取任务去执行** ) 不为空 就 上锁 (worker 对象实现了 qas) 处理任务然后释放锁 ，当task为空时说明处理完 就从阻塞队列中获取任务不为空就上锁处理

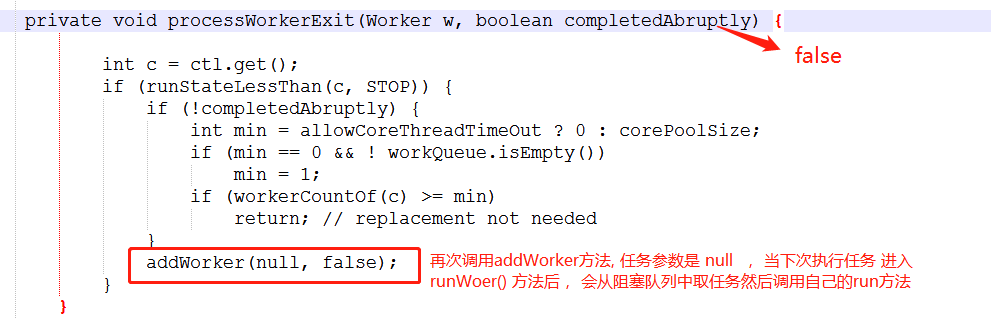
**注意 ： 在处理任务时 由于能获取到我们提交的 task任务对象**

**就可以直接 使用任务对象 调用.run() 方法 ，实现方法级别的调用**

**而没有使用 .start() 方法**



* 1. processWorkerExit() 方法



**线程复用总结** ： 当前线程池还未达到最大线程数处理任务时， 第 一次创建线程会将线程 和 任务一起封装成worker对象，然后 执行完任务后，将worker对象 从线程池中的 set集合中移除。

(虽然 work被移除 ， 里面包含了线程 但是只是线程对象 ，线程被start后是还在运行且没有被销毁) ， 执行完任务的线程

会 再次执行 ddWorker(null，false)方法 ， 但是任务对象worker

为null，然后执行run() 方法。 进而执行runWorker() 方法。然 后从阻塞队列中取任务再执行任务的run() 方法

1. 线程池中 任务的排队策略（使用哪种阻塞队列 处理任务）

a、直接提交（如SynchronousQueue） ：默认排队策略

直接提交策略表示线程池不对任务进行缓存。新进任务直接提交给线程池，当线程池 中没有空闲线程时，创建一个新的线程处理此任务。这种策略需要线程池具有无限增 长的特性 适用

Executors.newCachedThreadPool()使用SynchronousQueue创建线程池。

b、无界队列（如不具有预定义容量的LinkedBlockingQueue）

LinkedBlockingQueue将导致当所有 corePoolSize 线程都忙时新任务在队列中等待。这 样，创建的线程就不会超过 corePoolSize。（因此，maximumPoolSize 的值也就无效了。） 当每个任务完全独立于其他任务，即任务执行互不影响时，适合于使用无界队列。

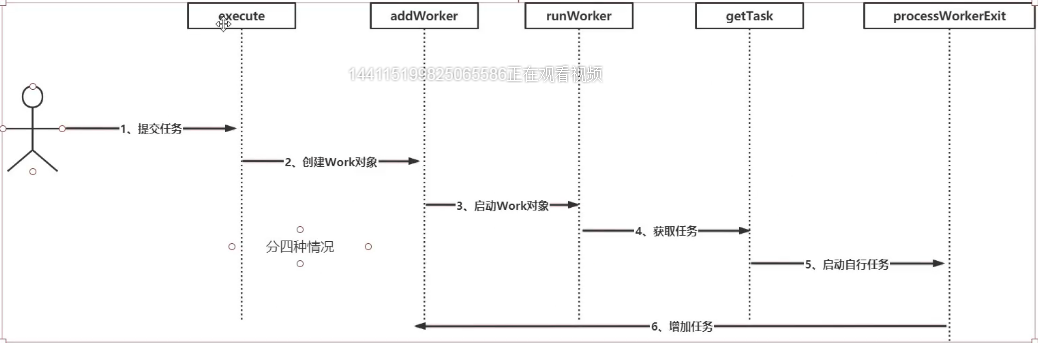
Executors.newFixedThreadPool(3)使用LinkedBlockingQueue创建线程池。

Executors.newSingleThreadExecutor()使用LinkedBlockingQueue创建线程池

c、有界队列（如ArrayBlockingQueue）

有界队列（如ArrayBlockingQueue）有助于防止资源耗尽当最大线程数有限时，但是可 能较难调整和控制。队列大小和最大池大小可能需要相互折衷。

1. 总结 线程池执行任务的 执行过程



1. 自定义线程池，如何设置线程数

线程如果少了没有尽量的压榨cpu资源，程序性能不足

线程如果多了 线程上下文频繁切换，导致性能很低 。 甚至还没有单线程处理的快

* 1. IO密集型 ： 基本都是IO读写操作，大部分时间cpu不用于计算处于等待状态

线程数 = ((线程等待时间 + cpu运行时间) /cpu的运行时间 ) \* 核数

一般都是配两倍CPU核数

cpu运行时间 ： 通过压测获取。

线程等待时间 ： 在进行IO数据操作时，线程是处于等待的

* 1. CPU密集型 ： 多用于计算数据的业务 ，尽量减少线程上下文切换 让cpu专注于处理数据 建议 线程数 = cpu核数 + 1

+1：表示不让cpu核空闲 。

1. 如何关闭线程池
2. . 关闭线程池 需要理解 thread.inteurpter 方法

对线程执行 thread.inteurpter 方法时分为两种情况

1. 正在执行任务的线程

正在执行的线程不会中断执行任务，但是会将中断isinterpreteed信号职位true

当线程执行完任务是就会自动关闭了， 如果想在执行过程中关闭线程可以根 据isinterpreteed 进行判断 ，如果等于true 就return或者break

1. 处于阻塞等待的线程(wait , io等待， park的线程)

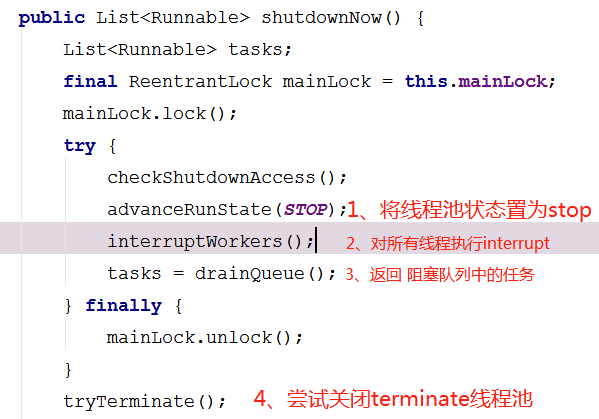
处于阻塞等待线程直接关闭然后 抛断异常 interpreterException

1. 执行 thread.inteurpter 后 ， 后续线程才阻塞 interpreterException 也会抛出
2. . 线程池提供的关闭方法

A . shutDownNow() 拒绝提交的任务并立即关闭线程池 丢弃阻塞队列中的任务

B. shutDown() 拒绝提交的任务 等待所有任务执行完毕 关闭线程池

1. shutDownNow() 关闭线程池的逻辑



A、shutDownNow 主要是将 线程池状态置为stop

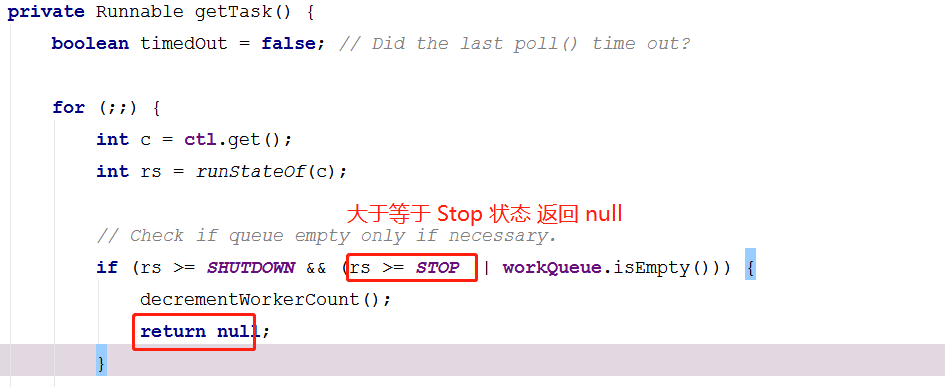
B、然后对 线程池中所有出创建的线程进行循环 interrupt 根据thread.interrupt 的 特性会关闭阻塞或者等待任务的线程 并抛出 InterruptException异常

1. 对于正在运行的线程 ， 在worker对象执行任务是 会调 run方法

Run方法中进行线程复用时会调用 getTask() 方法获取任务 ，如果getTask() 返回不为null 就对 worker对象上锁 进行方法级别的调用

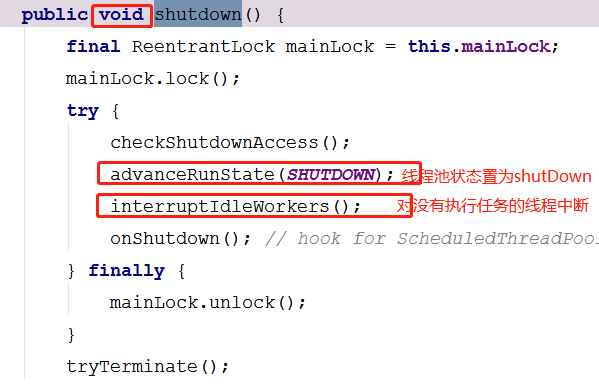
在关闭线程池是 将线程池状态置为 stop 目的就是让 getTask() 返回null

这样 线程执行完当前任务 ，就可以跳出死循环 终止执行任务

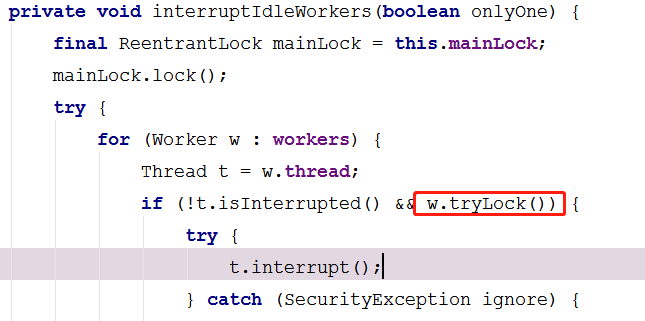


1. 接着 terminate
2. shutDown() 关闭线程池的逻辑
3. shutDown 方法中 会将线程池状态置为 shutDown 状态

而shutDown状态在getTask时不会返回 null， 而是会到阻塞队列中获取任务



1. shutDown 在 interrupet线程池中的所有线程时 不是直接循环interrupt 而是 判断worker对象是否是上锁状态的 ，如果上锁表示线程正在处理任务，则不 会去 interrupt



1. 等任务队列中的任务执行完后 ，接着 terminate

**总结 ：**

**shutDownNow() 会丢弃未完成的任务并抛中断异常然后返回未执行任务的集合**

**shutDown 会执行完所有任务然后停止 。 如果在执行任务的过程中出现了阻塞或 者执行时间过长线程池会一直等待下去**

**因此可以使用 awaiteTermination 方法阻塞主线程，等线程池彻底关闭了继续往下执行**

1. 接口幂等 : 多次请求结果是一致的 ，不会因为多次请求发生业务错误的现象

1. 类加载机制
2. 双亲委派的加载流程

假如是自定义ClassLoader去加载一个类 ,首先先到自己开启的方法区内存中去找，是 否有 如果没有，自己不会去加载 则调.parent获取父类的classloader 使用 app去加 载 ， app此时并不会加载 在调 .parent获取父类classLoder让ext去加载 ， ext也不 会加载而是通过 boot去加载 (因为boot是加载jdk核心内部类的) 所以也加载不到 ， 然后再让boot调用ext去加载 ，还是加载不到 再让ext调用 app去加载 。App能加 载到，然后将加载到的 class 缓存至方法区

问题 ： 此时 自定义的classLoader 是没用的，因为默认是双亲委派的机制 自动类是 通过app去加载的 ，如果要让自 定的classloader是有用的话 那么就打破双亲委派

1. 双亲委派的 好处

一个类不存在被不同的加载器重复加载 ，因为找到就返回

沙箱安全:保证系统核心的 class 不被串改(因为自己不去加载先让父加载器加载)

1. 类加载器存在 父子关系 （不是继承关系 ，通过 ApplicationClassLoader.parent = Ex 体现）

ApplicationClassLoader.parent = Ext

Ext.parnet = null 。

1. 同一个 class被不同的 classloader， 不是同一个类型(不是同一个)
2. Jdk 动态代理实现（jdk动态代理基于 接口）
   1. 实现 InvocationHandler 接口 重写 invoke(Object obj) 方法
   2. 将 目标对象传入 实现InvocationHandler 的类中
   3. 然后在 invoke方法中 扩展方法

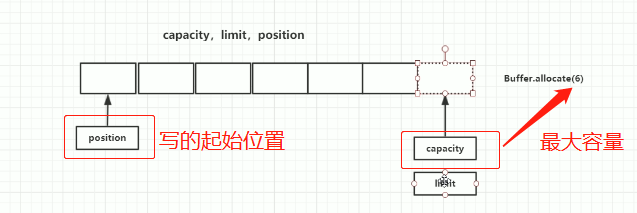
**NIO**

1. Java 中搬运数据是通过流IO ，但是有了 nio后 没有流的概念 也能传输数据

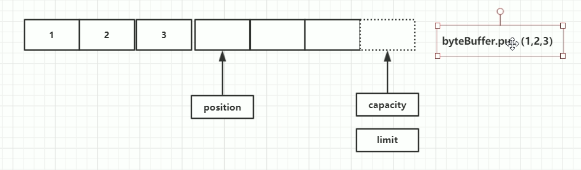
1. 使用带 缓冲池buffer 的 bio 传输数据时， 每次写之前 要buffer.clear()
2. NIO3个核心
   1. **Buffer**  存放数据，支持put 各种类型的数据

重要属性：

* + 1. Capacity 最大容量且不变的 （大小根据容量决定）
    2. Limit 数据的大小 ，不会大于Capacity = limit 初始值
    3. Position 标记下一个读或写的位置Position = 0 初始值



当写数据时position 就会往右移动



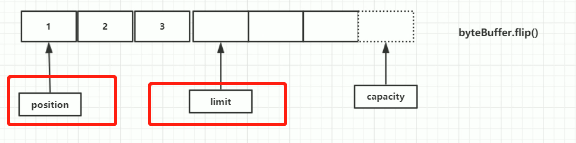
写完后，position位置 ，然后调用 byteBuffer.flip(); 切换读写状态后

Limit 移动Position 位置， Position 还原最初位置 。 position和limit之间的 数据表示 写的数据

注意： A、position的位置 永远不会大于limit 当等于limit时 就移动不了

1. 当往 buffer中存不同类型的数据时 ， 取 也是根据对应的类型来取的

|  |
| --- |
| ByteBuffer buffer = ByteBuffer .allocate(100);  buffer.putChar(“xx”);  buffer.putInt(123);  buffer.flip()  buffer.getInt(); ---->123  buffer.getChar(); ----> xx |



注意： 切换完状态后 buffer中的数据会被覆盖

重要方法 ：

1. Buffer.clear() 清空缓冲区

position = 0 ;

limit = capacity

1. buffer.flip() 将buffer 的写模式 切换成 读模式
2. buffer.Array() ; 直接返回hb 数组 （hb：数组是 buffer堆内 内存数组对象）
3. buffer.slice(); 创建新的缓冲区 返回一个新 buffer 但是数据指向原来的buffer 内存，当改变任意一个buffer 中的数据 ， 所有buffer中的数据都会变化
4. buffer.asReadOnlyBuffer(); 返回一个只能 读 的byteBuffer， 一旦有写操作就报异常
5. buffer.mark() ; 将position中的值 赋值给临时变量 mark 保存。用于备份position值 。 当写数据时 postion会移动 ， 但是mark 会备份位置， 用于后面使用。
6. buffer.reset(); 将临时保存的mark中的值 ，赋值给 position也就是还原 position的值。
7. Buffer.allocate();   
   [ˈæləkeɪt] 内部创建的**堆内内存**heapByteBuffer(capacity,capacity); 本质是byte数组
8. Buffer.allocateDirect(); 内部创建的是 堆外内存 DirectByteBuffer(capacity);

堆内 内存 和 堆外内存的区别 ：

heapByteBuffer ：数据是存在 jvm堆内存中效率低：使用时需要拷贝到操作系统

1. heapByteBufferR ：堆内存 只读 一旦有写操作就报异常

DirectByteBuffer： (堆外)仅仅对象引用是在jvm中 ，数据存放在操作系统内存中 效率高(netty都是基于堆外内存)

1. buffer.wrap(byte [] byte); 创建的**堆内内存。**接收byte数组 返回一个存放byte数组的byteBuffer对象。

即:byte 和 buffer 对象共享一个内存, **数据动态都会互相影响**

如果直接修改任意一对象数据 ，都会发送改变

* 1. **Channel ：**管道 运输一个个 buffer，用于读写数据  
     通过File对象获取 channel ，chanel是连接 buffer 和 文件的载体

如可以获取 file 管道 ： new File().getCannel() 、 socket管道等

方法有 ：

1. channel.read(buffer) 将buffer读到channel
2. channel.write(buffer) 将buffer写到channel(写之前 buffer一定flip)
3. Bchannel.transderform (position,limit, Achannel)

从Achannel中对数据， 写到Bchannel文件中

1. Bchannel.tansferTo(position,limit, Achannel) 和 transderform 方法相反

从Bchannel中对数据， 写到Achannel文件中

E、channel.register(selector) 注册 channel到 selector

**Selector ：** 用于注册chanel

1. .为什么使用 selector ?

单线程无法实现socket并发的 ， 当多个客户端连接 socket服务 ，假如有一个客 户端**连接服务端**后死活不发数据 ，服务端读不到数据。 那么服务端一直等待 那 么就会 阻塞 。

两个阻塞点 ： 连接阻塞 和 服务端读取数据阻塞 (跟客户端 无关)

Netty 中对于这个阻塞点 是绑定 reactor去异步执行



如何实现 单线程下 非阻塞 ？

获取连接后 不为 null ，说明有客户端连接，然后读取客户端数据 ，如果数据为 null ，就将 当前连接保存至 set集合中 。 每次读数据时就 循环集合。 客户端 一旦发送了数据 就将 连接从set集合中剔除。而nio的selector组件就充当了 set 集合的功能(客户端连接是一个 个channel)。

selector 之所以能实现 单线程的非阻塞 ，是将这个 循环连接的操作交给操作系 统去处理了。

selector可以实现一个线程管理多个网络连接 。 用于夸应 用socket通 信时使 用 。 管道channel存放至 selector的 set集 合中 以SelecttionKey 的形式存在 Set<SelectionKey> selectedKeys = new HashSet();

selector内部 遍历selector中的 channle ，即使有些channe读不到客户端连接发 送的数据 ，

那么就遍历下一个 ，这样就实现了 非阻塞。

1. 什么是 selectionKey

Channel往 selector注册时 ， 是以selectionKey的形式存在， selectionKey就是一 个个被注册的channel.其中维护了4个事件进行监听 accept/read/write/connect

selectionKey方法 ：

1. selectionKey.select() 返回int 。 用于轮询获取注册的事件 ，

一旦有事件触发 就 > 0 ;

1. isVaild 检查是否有效
2. cancael 注销
3. attach 添加附加对象，将一个Accepttor对象（线程）绑定到 此 selectionKey上， 从而关联当前绑定的事件 ,当前事件触发时 就调 Accepttor 的run方法
4. attachMent 返回Accepttor 附加对象 (线程)
5. interestOps(事件) 改变注册事件
6. selectionKey.selector().wakeUp() 可以**让下一次 select() 方法立即返回结** 果（可能会阻塞）

一般有新事件注册 ，或者事假修改 才调用此方法刷新

1. 用法
2. 创建服务端 channel 并绑定端口

ssc.socket().bind(new InetSocketAddress("127.0.0.1", 8000));

1. Selector.open() 创建 selector对象

Selector selector = Selector.open();

1. 一定设置非阻塞 ， 此处是设置 连接方法 非阻塞。

channel.configureBlocking(false)

1. 注册**serverSocketCannel**， **并指定连接事件**，

channel.register(selector, SelectionKey.OP\_ACCEPT);

1. Selector.select(2s) // 表示每隔2s， 去检查是否有读事件发生 如果不指定

Selector.select()

则一直等到有客户端 连接才会走下面的代码(变成阻塞的了)

1. Set<SelectionKey> selectedKeys = selector.selectedKeys();

是获取感兴趣的accept连接事件 ， 被封装set集合中 ， 如果读取数据的话就要遍历

1. Iterator<SelectionKey> it = selectedKeys.iterator();

使用迭代器进行迭代 set 集合

1. **判断是连接事件 还是 读事件 ， 如果是连接事件就 注册channel** 。

如果是读事件 就创建 bffer 获取数据

if(key.isAcceptable()) { // 如果是连接事件

channel.accept() // 获取服务channel

channel.register(selector, SelectionKey.OP\_READ);**//注册客户channel，**

**绑定读事件**

channel.configureBlocking(false) //此处设置非阻塞， 读取数据非阻塞

} else if (key.isReadable()) { //如果是 读事件

// 创建buffer

// 往 buffer写数据

// 切换状态 flip

// 读数据

} else if (key.isWritable()) { //如果是 写事假(转发给其他客户端)

// 创建buffer写数据

}

**it.remove(); //处理完之后 删除当前channel ， 不然会重复操作数据**

完整 ：

ServerSocketChannel ssc = ServerSocketChannel.open();

ssc.socket().bind(new InetSocketAddress("127.0.0.1", 8000));

ssc.configureBlocking(false); //设置非阻塞

Selector selector = Selector.open();

// 注册 channel，并且指定感兴趣的事件是 Accept

ssc.register(selector, SelectionKey.OP\_ACCEPT);

selector.select(2s) ;

while(true) {

int readyNum = selector.select();

if (readyNum == 0) {

continue;

}

Set<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> it = selectedKeys.iterator();

while(it.hasNext()) {

SelectionKey key = it.next();

if(key.isAcceptable()) {

// 接受连接

} else if (key.isReadable()) {

// 创建buffer读数据

} else if (key.isWritable()) { //如果是 写事假(转发给其他客户端)

// 创建buffer写数据

}

it.remove(); //处理完之后 删除channel

}

1. Selector.select(2s)、 Selector.select()、Selector.selectNow() 表示隔多久 检查感兴趣事件

配合 Set<SelectionKey> selectedKeys = selector.selectedKeys(); 使用

1. selectNow() 调用一次 检查一次感兴趣事件
2. select()不设置时间进行 空轮询，阻塞
3. 设置时间不阻塞，**但是 由于 jdk bug 会偶尔会导致设置的时间无效 就会一直阻 塞。导致nio 变成阻塞的。netty就规避了这个 bug , netty中的解决方法 ， 解决 逻辑**

**由于设置了检查时间， 就在**Selector.select(2s) 方法前获取系统时间

然后相减 ， 如果 大于 间隔时间 ，表示没有 出现bug ，如果小于间隔时间表 示 出现了 bug , 就获取 Selector.keys 所有注册的 事件 ， 然后遍历所有事件 进行cancel永久删除 ， 注册给新的 Selector对象 (等于换了一个新的 selector 对象)

1. selector.selectedKeys 和 selector.keys 的区别

keys ： 表示 注册到 当前selector上的所有 事件。 (所有事件的全集)

selectedKeys ： 获取设置事件后的 对应的感兴趣的事件

如 ： channel.register(selector, SelectionKey.OP\_READ);

调用selector.selectKeys() ; 方法 自动获取的读事件，并返回set集合

1. Canel 和 remove 方法的区别

Set selectedKeys = selector.selectKeys()；

Iterator<SelectionKey> it = selectedKeys.iterator();

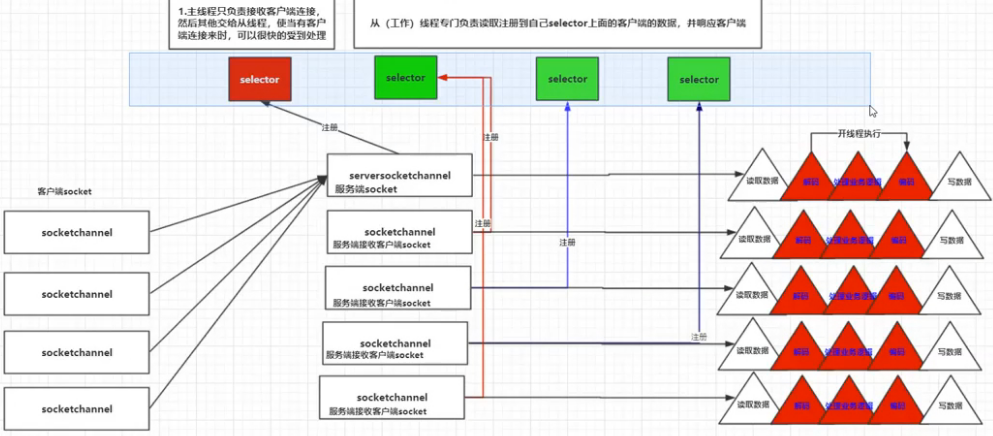
it.remove() 表示处理完后，将channel 移除当前集合

it.cannel() 不再监听此客户端， 并不监听其任何事件 将channel添加到cancelledKeys集合中

1. 如果 需要将 客户端 发来的数据进行广播 ， 就获取所有的 channel ，再剔除自己

进行轮询 写到其他客户端 channel

1. 注意 ： 注册chanel时
   1. 先注册服务端channel 绑定连接事件
   2. 再注册客户端channel 绑定读事件
   3. 然后根据不同的事件 ， 处理不同的事 如果是连接事件 就注册channel，如果是读事件就获取数据
2. NIO reactor 模型
   1. Nio 单线程的缺点 ， 服务端只能同时 处理 一个客户端发来的数据
   2. Netty 中的 reactor 模型



Reactor 模型主要有两个地方 开启多线程 ：

1. Selector

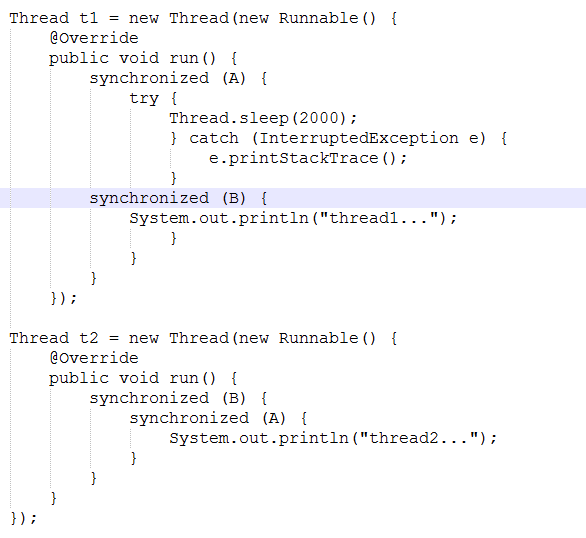
Main线程上的 selector 主要注册用于客户端的连接

客户端读取事件则开启线程池 cpu核心数\*2 (IO密集型) ， 以轮询的方法注册

1. 读取客户端发送信息后 ， 处理业务逻辑开启多线程
2. Synchronized 死锁情况示例

private static String A = "A";

private static String B = "B";



1. i ++ 和 ++i 的区别

这两种 都可以写成 i = i + 1 ; 的形式 ， 区别是复制先后的问题

1. i ++ 是 先赋值再运算 i = i + 1

如 ： int i = 1;

Int j = i ++ ;

等于 ：

int i = 1;

Int j = i;

i = i + 1;

或者：

Int i = 1;

System.***out***.println(**i++**); // 结果为 1

1. ++ i 是 先运算 i = i + 1 再赋值(正常顺序)
2. Mysql binlog的3中模式
   1. Row level 行模式
   2. Statment level 默认
   3. Mixed 自动模式
3. 死锁产生的条件

互斥条件；环路等待；保持条件；不剥夺条件

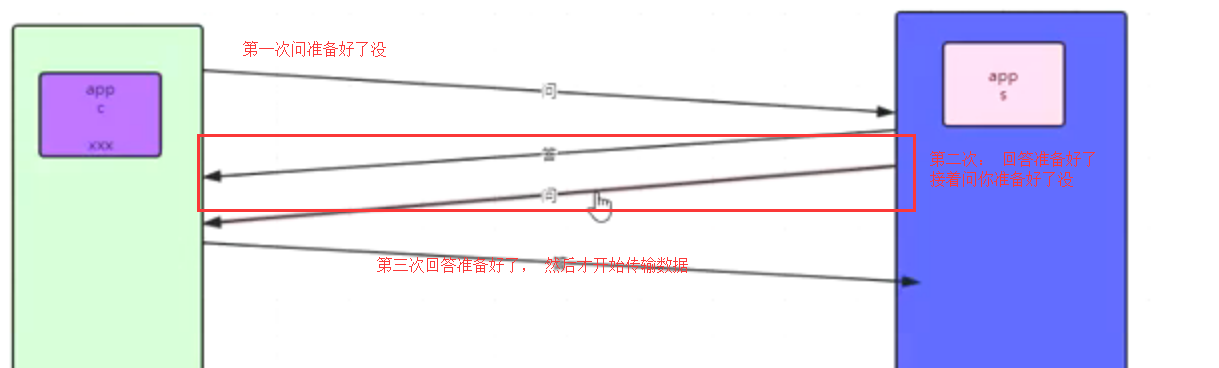
1. 如何实现 线程安全
   1. 加锁
   2. cas
   3. threadLocal
2. 大文件 分片下载，上传

下载主要思路 ： 多线程下 输出到文件后 ， 主线程往外输出

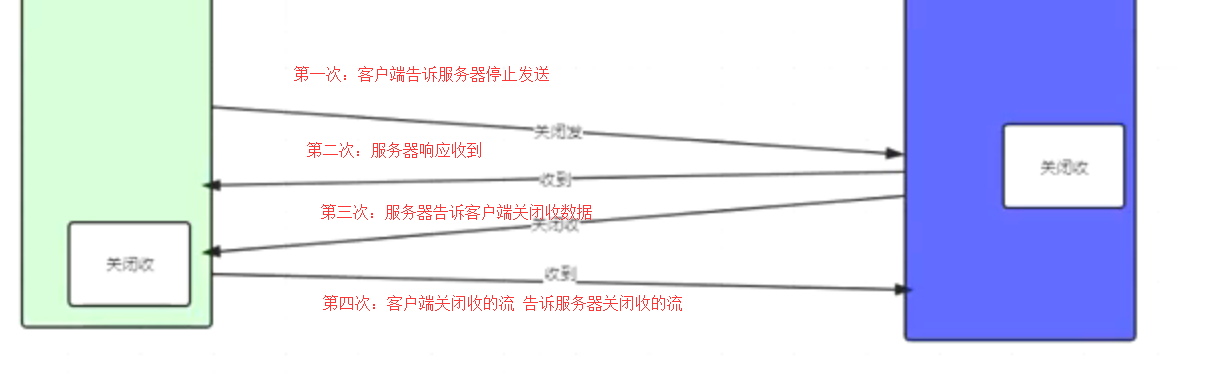
1. 先获取大文件 总字节数 ， 然后平均分几块
2. 根据分好的块 ，使用 CountDownLatch 让线程下载文件
3. 下载对应的块后 , 再写入到主文件 ，在写时 可以使用RandmonAccessFile

他有seek 随意移动指针的方法 ，将下载好的块写入到文件 ， 不用按照顺 序写入 。 每个线程写入后 都进行 CountDown

1. 当state = 0 时， 主线程往外输出数据
2. tcp的3次握手 4次挥手 ： 数据 以包的形式进行发送



为什么是3 次握手 ？ 第二次时 包括了响应和问



第一次 ： 当客户端发送关闭接收数据的socket.close后 ，服务器端收到 此时 socket 的read还是处于阻塞状态 ，当收到这个包后 服务器端的socket会解除read阻塞 操作系统内核会将socket read解阻塞（因为此时客户端不会再往服务端发送数据了，再阻塞没有数据过来）。

第二次 ：服务器端的socket响应给客户端的socket的包（收到）

第三次 ：告诉客户端关闭发送数据服务端socket.close

第四次：客户端告诉服务器关闭了接收数据的socket （在丢包的情况下 或者 客户端程序直接关闭 或者延迟发送，此时服务器端就处于等待状态 并触发服务器的延迟机制再次向客户端发送包。 如果客户端直接关闭进程 ，服务器端也不知道发送给谁了 然后客户端的底层的socket端口会等待两倍于(此时底层的端口会被强制占用) 服务器端的最大发送间隔时间 如果时间到了 服务器还未发送数据 端口关闭 ）