**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

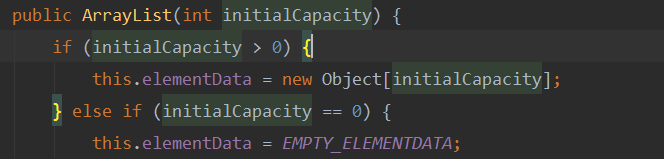
* + 1. empty\_elementData = {}

创建空的new ArrayList(0)时 ， 引用指向的数组。那么无论创建多少个list都是指向这个 数组。从而节省内存

* + 1. defaultCapacity\_empty\_elementData

创建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

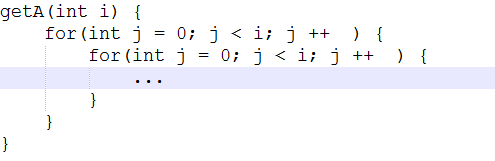


* + 1. 为什么 创建list指定大小跟不指定大小 ，指向的 数组不是同一个

因为list扩容时 ，是在原来容量的大小上扩容 1.5倍 ， 在首次扩容时要根据初始大小

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)

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

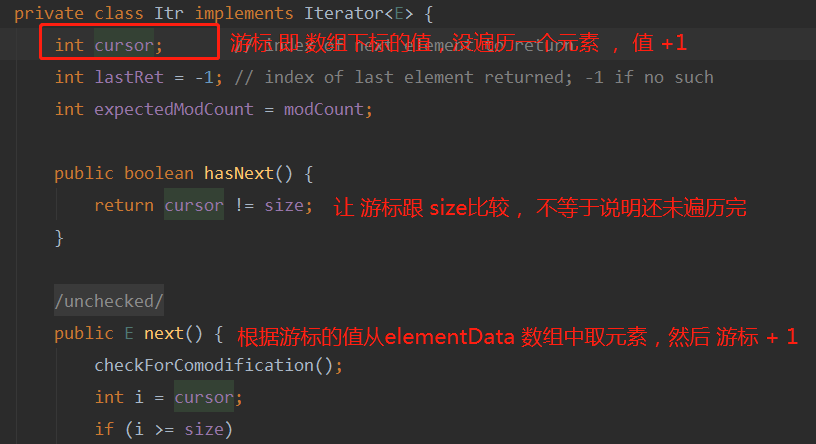
类似添加元素 ： 将删除的元素 置为null ， 然后后面的元素 下标有 -1

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. 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

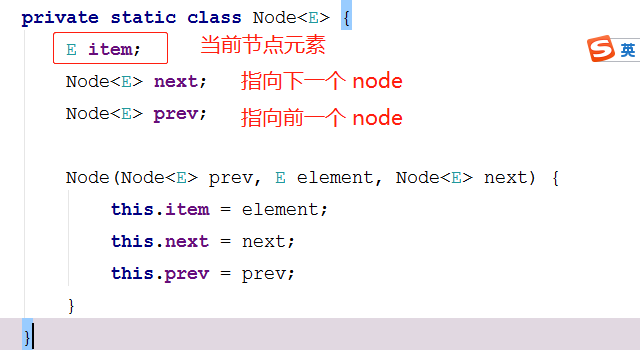
但是ArrayList() 在 存放基本类型的数组时 ，会将数组对象放在 list的一个元素中

也就是说不能将数组转成 List 。引用类型是可以的

**LinkedList**

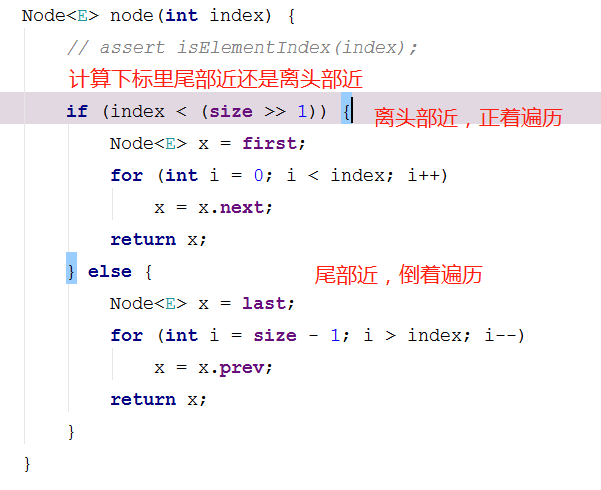
1. linkedList 链表 内部 维护了一个 node 内部类用于 ，关联前后节点。

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）



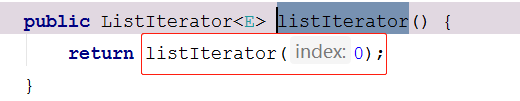
1. linkedList 查询方式
2. Get(int index)方法

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**

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 &

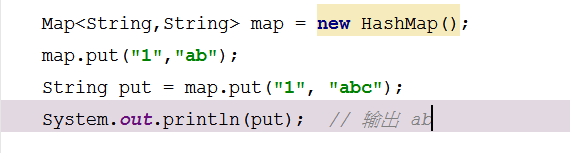
为了性能高 ，之前是取余

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

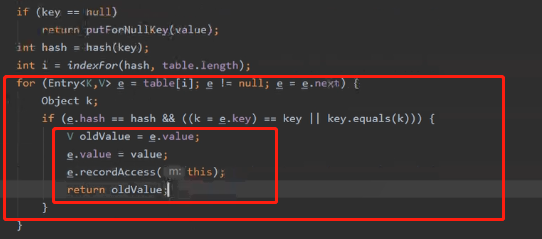
如果 put的key已存在 就覆盖原有的value值， 返回原有的 value值 。如果是添加 新 值就是返回null



(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

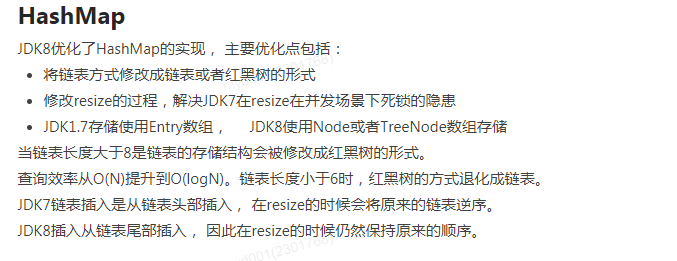
threshold=capacity\*loadFactor

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

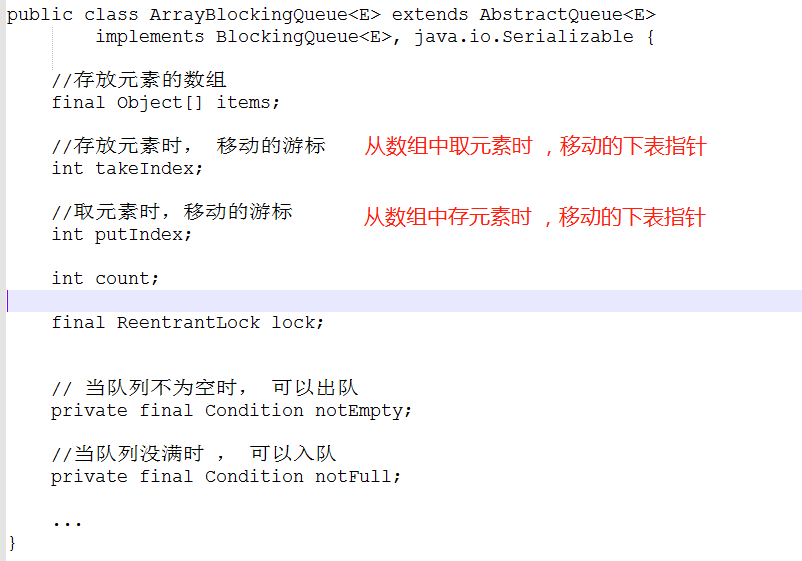
* 1. currentHashMap、CurrentSkipHashMap、hashtable

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。

**ArrayBlockingQueue**

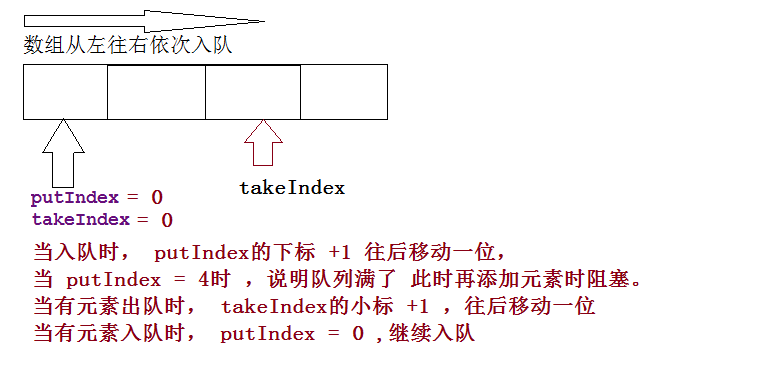
1. 队列操作相关
   1. 存
      * 1. add 添加 满了抛异常
        2. offer 添加 满了返回false
        3. put 添加 满了阻塞 ， 且可以设置阻塞时间
   2. 取
      * 1. remove 取元素，没了抛异常
        2. poll 取元素，没了返回null
        3. tack 取元素，没了阻塞等待 ，可以设置等待时间
   3. 查看队首元素
      * 1. Element 队列为空时， 抛异常
        2. peek 队列为空时， 返回false
2. ArraylistBlockingQueue 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();

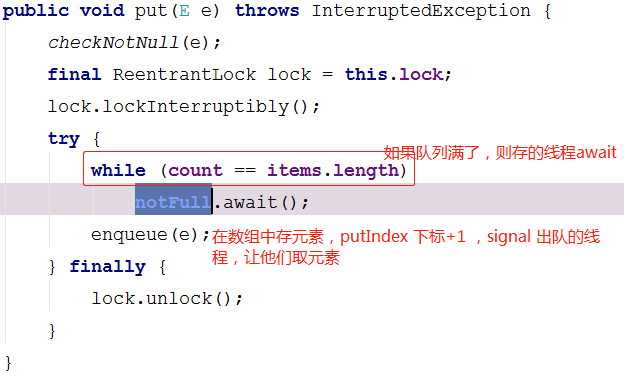
1. ArrarListBlockingQueue的原理



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

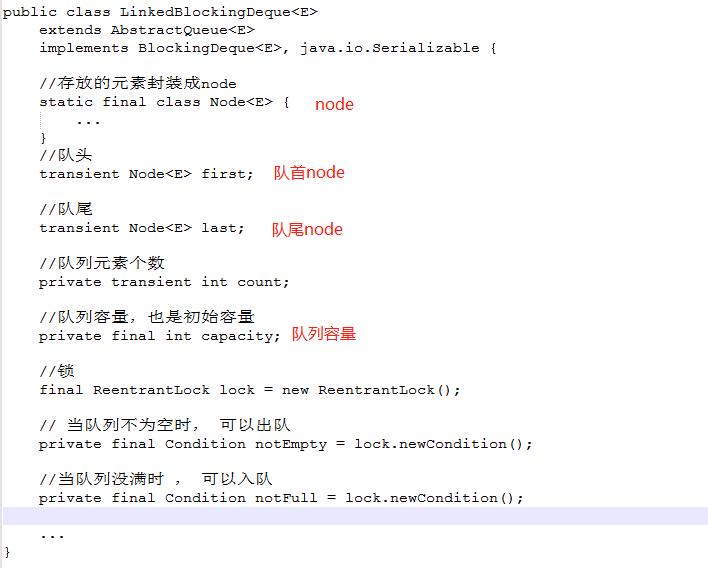
**LinkedBlockingQueue**

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

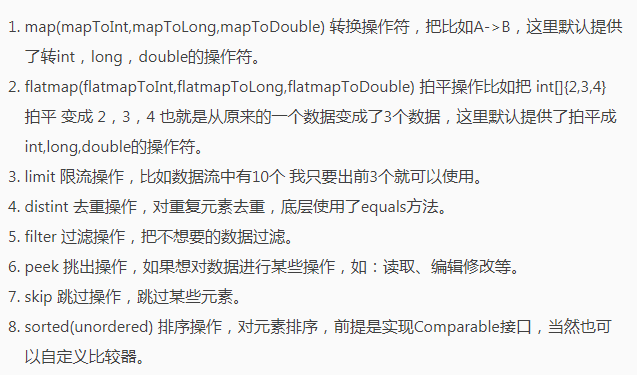
如果一个接口中此注解注释 ， 就表明此接口中不能再添加方法包括 default方法

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

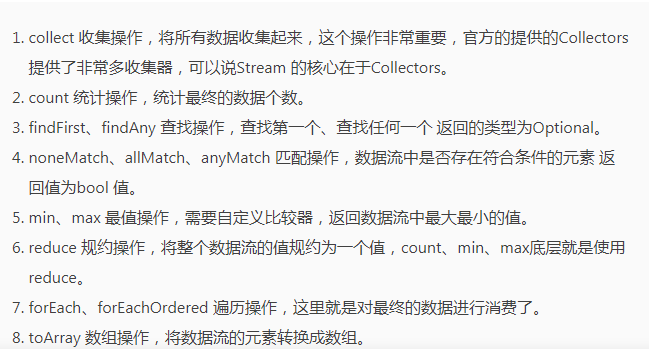


延迟执行 ： 做终止操作的时候 ，操作做真正的操作处理数据

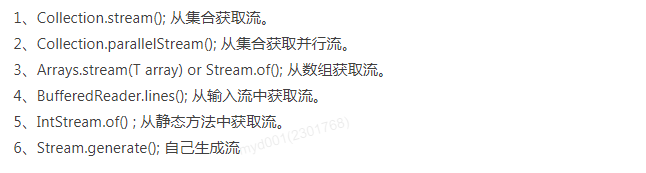
* 1. 操作分为两种 中间操作符 和 终止操作符
     + 1. 中间操作（特点：处理结果可以继续交给下一个中间操作继续加工数据）



* + - 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）

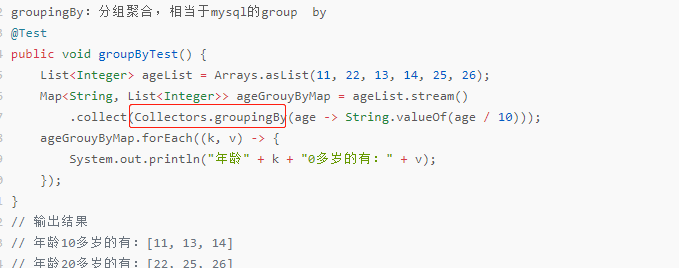


* 1. 创建数据源的方式有哪些 （如果将 数组、集合转成stream对象）





* 1. 分组统计 类似 mysql 中的 group by



**线程安全得单例模式**

1. 较优的单例实现 ： 高效、安全、懒加载
   1. **Holder模式**

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，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

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 ，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

* 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

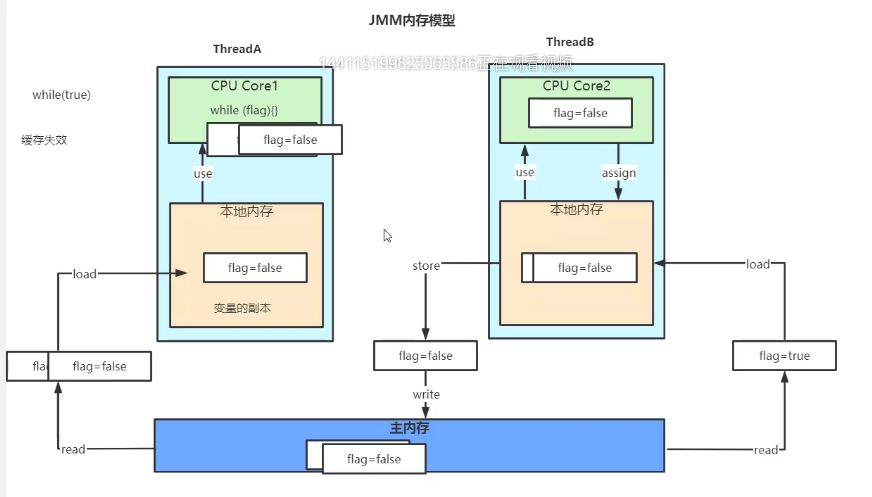
1. 调用线程为什么要 调用statr()方法 ，而不是直接调用run 方法

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
   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

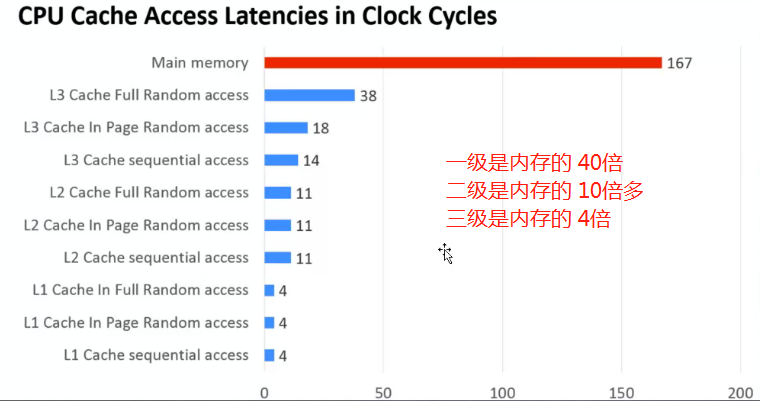
注意： 添加 jvm参数-XX:-RestrictContended 才起作用

1. 指令的重排序问题
   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()

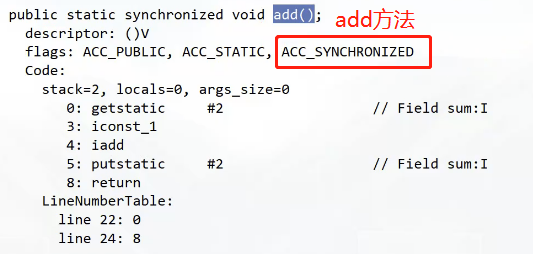
**synchronize**

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

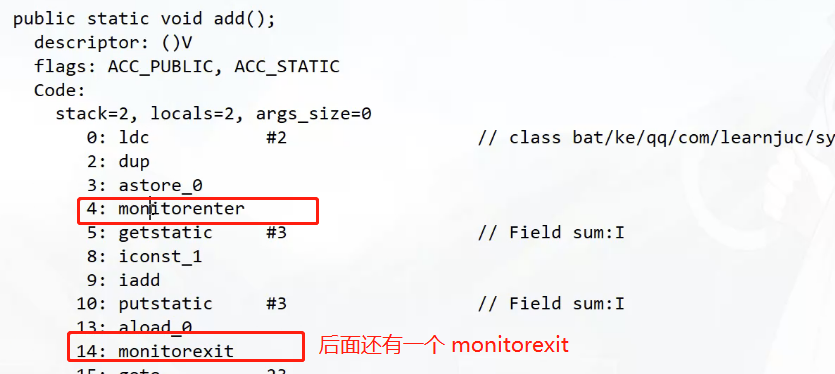
1. 使用
   1. 静态方法上加锁
   2. 方法上加锁(重量级锁)

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

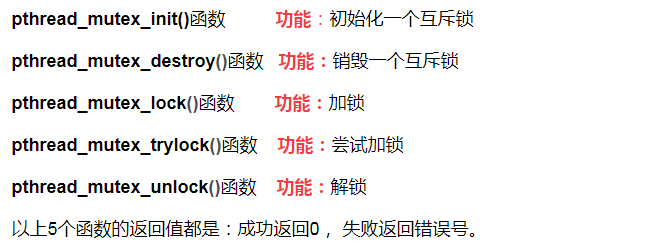


* 1. 对象上加锁(重量级锁)

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 ： 当synchronize被编译成acc\_synchronize、monitorEnter、monitorExit指 令后底层加锁是通过 mutex 进行加锁的



1. 重量级锁上锁的底层实现

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时 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、锁获取的次数、重入次数、阻塞线程集合、等待唤醒集合等 。 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

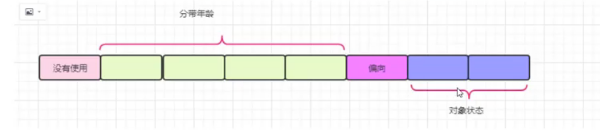
ObjectMinitor类中 thread 等于当前线程，表示持有锁。当然真正持有锁的操作是执行mutex函数 ，如果返回 0 表示当前线程锁持有成功 。 接着进行可重入判断 就是判断ObjectMinitor类中 thread 是否是当前线程 ，如果是就进行重入 可重入属性 + 1

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 monitorObject对象的内存地址

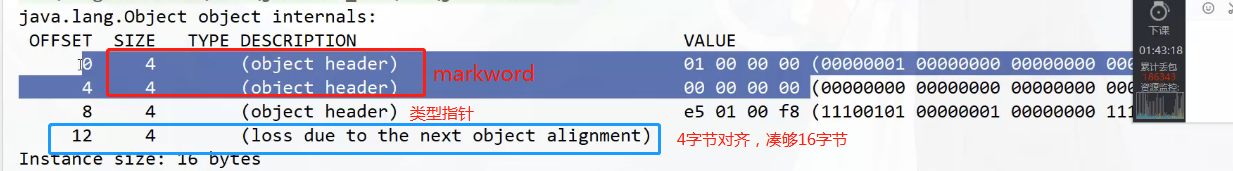
倒数第3位 ： 1 偏向 0不偏向 **偏向锁 101**

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**

**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

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 ， 检查对象头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，而是等线程1到达安全点 ，才能撤销成功，

撤销后此时判断线程是否执行完代码块释放了锁 ，如果执行完 markword中的线程id是为空的，然后cas设置成自己的thread id。

就可以争抢到锁

如果线程没有执行结束并没有释放锁，将升级成轻量级锁。此时cas修改对象的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) ：不是线程执行同步块结束，而是必要的代码执行结束， 如for循环执行结束，属性赋值结束等

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

自适应自旋 ： 早起无自适应 ，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

1. **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
2. 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

1. 为什么 synchronize 是非公平竞争的方式取锁，并且在唤醒是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. 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()

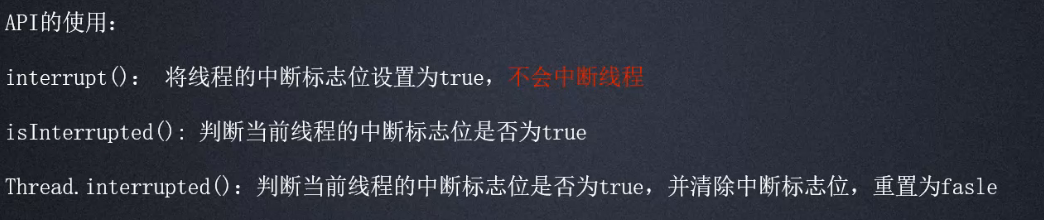
特点 ： 无先后顺序， unpark 也可以在 park之前执行

* + - 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



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 并一个异常 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 ，如果该所被其他线程持有 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

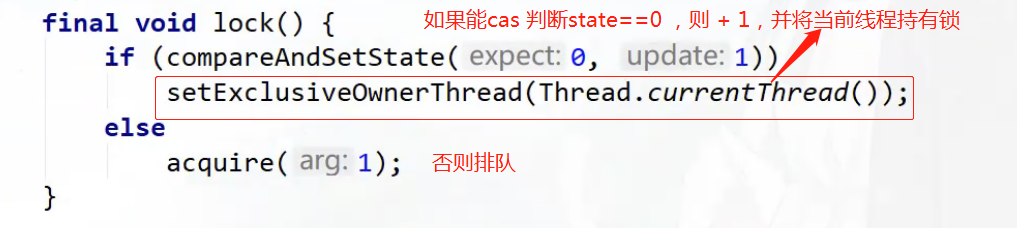
boolean tryLock(); 尝试获取锁如果成功返回true，如果是失败返回false

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

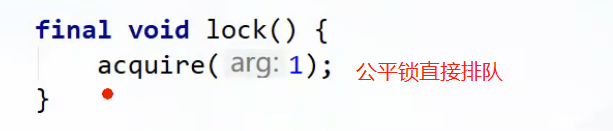
void unlock(); 释放锁

Condition newCondition();

1. AQS
2. 非公平锁



1. 公平锁

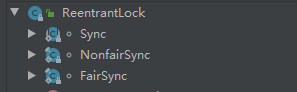


1. ReentrantLock 的1个属性 ， 3个内部类
2. 1个属性

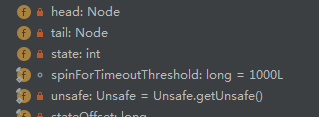


只有一个 sync 属性，即 AQS 类

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

tail ： aqs链表头节点node

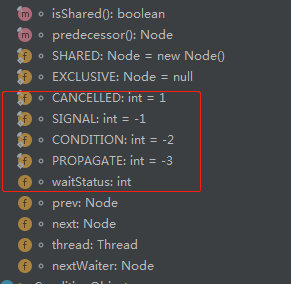
state ： 线程持有锁的状态

Unsafe ：用于底层对线程操作 唤醒阻塞等

* + 1. .内部类



a.Node : 队列（双向链表实现）



thread

waitStatus保存线程执行任务的各个状态

线程的种类(共享和独占)

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实现线程写作

1. AQS上锁过程
2. AQS解锁过程
3. 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

waitStatus = -1 表示当前线程是个可以被唤醒的线程，接着就开始interruppt中断线程然后 park阻塞

1. 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

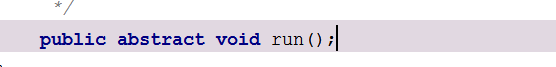
1. .

**线程**

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

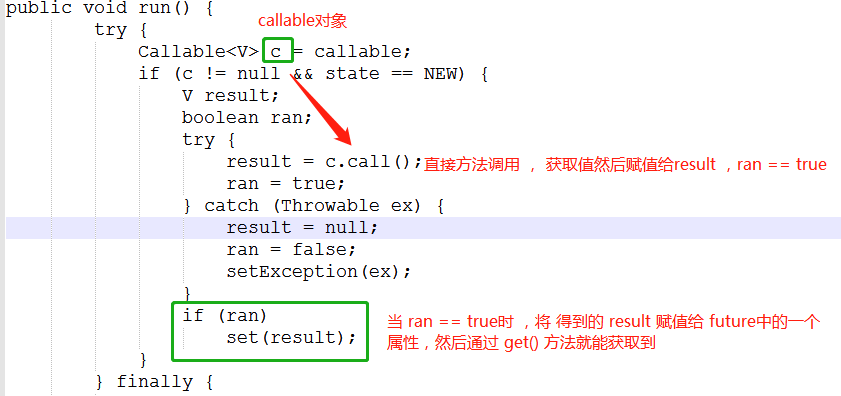


* 1. Callable 创建一个带返回值的线程，内部还是使用 Runnable接口



* 1. Future

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 会调run方法。当线程start()后,由于在实例化 Future对象时 构造器 依赖了 Callable 就 能获取Callable对象 ，然后通过方法调用call 方法 获取到值 。



* 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

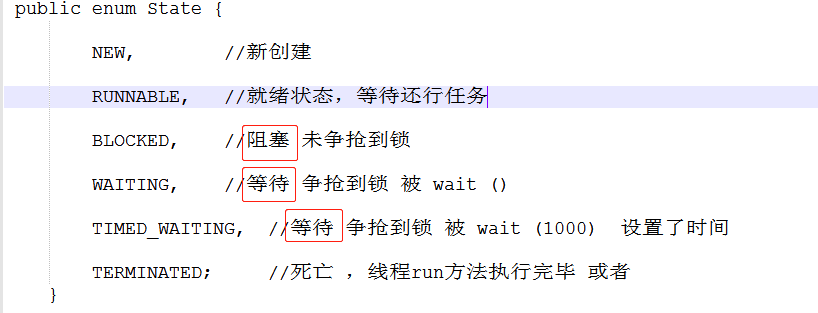
boolean cancel(boolean mayInterruptIfRunning);

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,

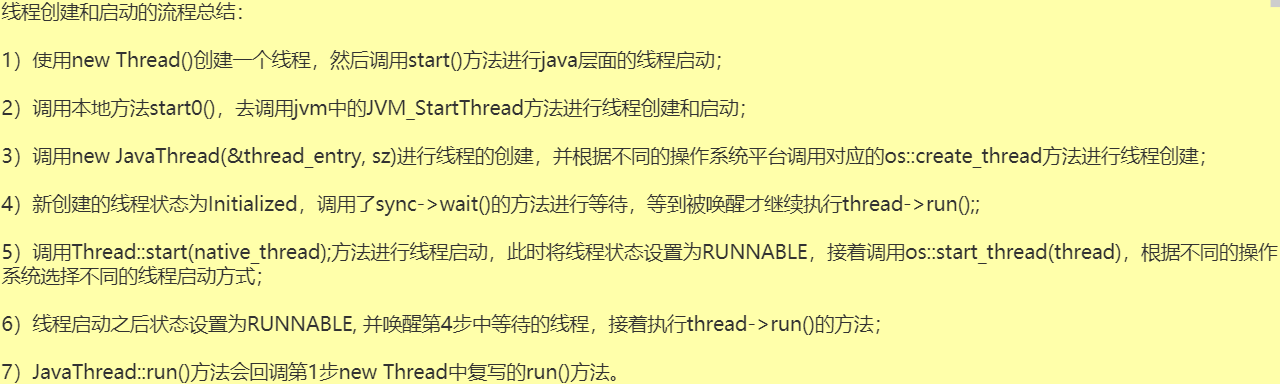
timed\_waiting 是超时等待 ，是设置了时间的

**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

**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

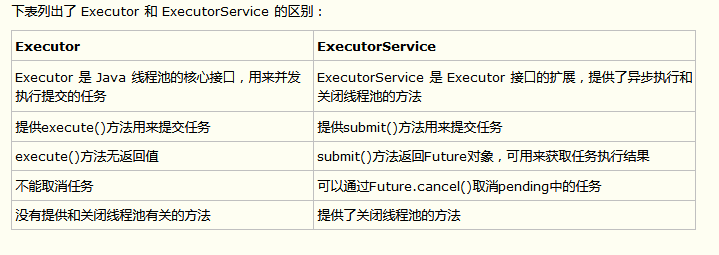
特点 ： 核心线程数是 0 ， 最大线程数是无限大 。 根据任务个数来创建线程去执行 。 执行完毕后时间小于60s线程可自动回收可以复用 大于60秒销毁。

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
   3. 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 当来的任务超过 corePooSize ，线程池开始创建新的线程

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：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，一旦有核心线程空闲 则直接使用execute提交任务 ()

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

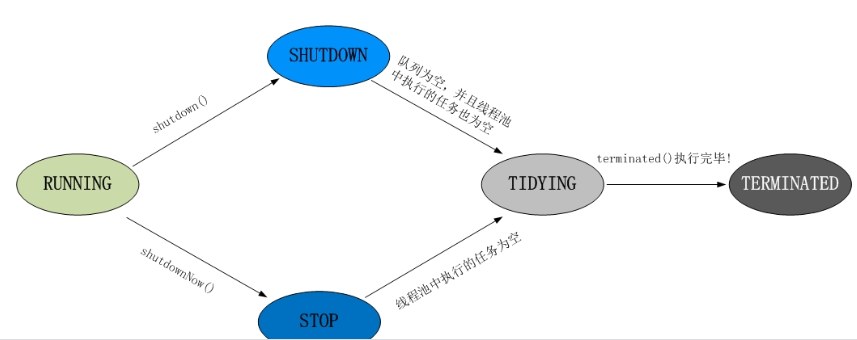
当触发拒绝策略，只要线程池没有关闭的话，丢弃阻塞队列 workQueue 中最老的一个任务，并将新任务加入

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 。

为什么线程池分提交 和 执行两个阶段 。

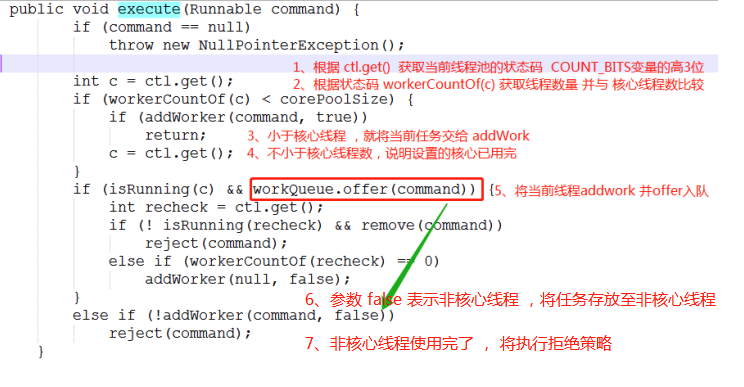
类似生产消费者 ， execute方法只专注提交任务 提交后然后return

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



**WorkQueue对**象：阻塞队列

execute提交线程后返回 ， 然后addwork() 添加任务封装成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对象添加至

HashSet<Worker> **workers 中 ，接着启动这个work对象的run方法**

* 1. **执行任务的顺序**

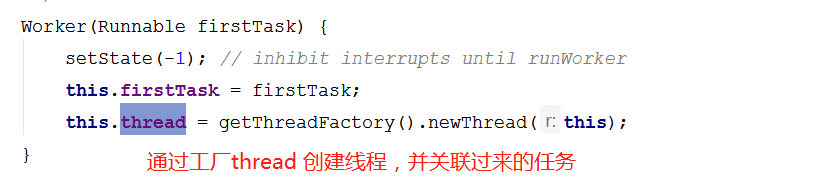
核心线程---> 非核心线程--->阻塞队列

**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

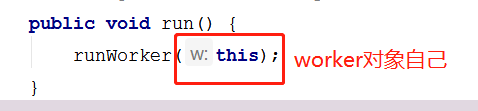
**Work 对象**就是将添加的任务关联一个线程封装成work对象



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()



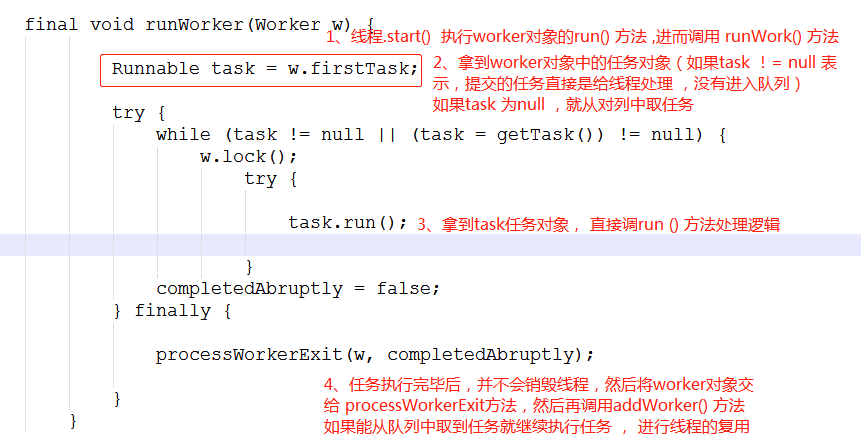
1. runWork(worker) 方法
   1. **死循环 处理任务**

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

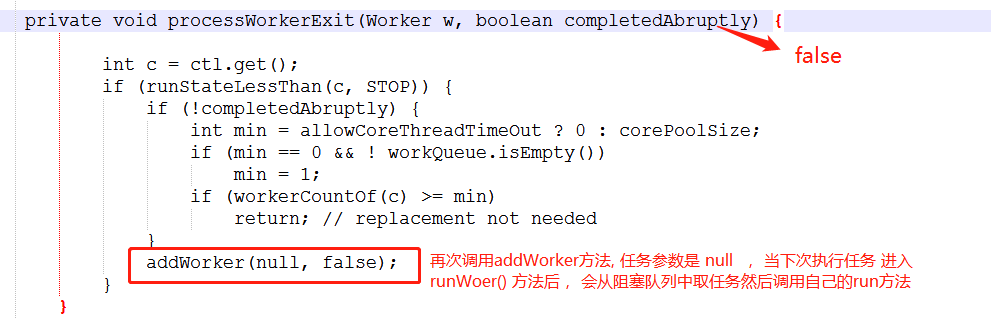
**注意 ： 在处理任务时 由于能获取到我们提交的 task任务对象**

**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**

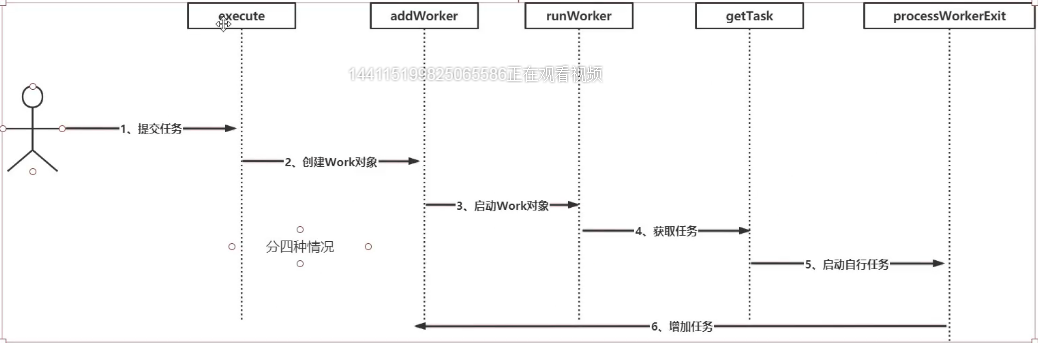


* 1. processWorkerExit() 方法



**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， 这个worker对象并不销毁 会将这个worker对 象再次添加至 addWorker(null，false) 任务置为空 ，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. 总结 线程池执行任务的 执行过程



1. .
2. .