# 线程基础

**CPU核心数和线程数的关系**

核心数:线程数=1:1 ;使用了超线程技术后---> 1:2

**CPU时间片轮转机制**

又称RR调度，会导致上下文切换

**什么是进程和线程**

进程：程序运行资源分配的最小单位，进程内部有多个线程，会共享这个进程的资源

线程：CPU调度的最小单位，必须依赖进程而存在。

**澄清并行和并发**

并行：同一时刻，可以同时处理事情的能力

并发：与单位时间相关，在单位时间内可以处理事情的能力

**高并发编程的意义、好处和注意事项**

好处：充分利用cpu的资源、加快用户响应的时间，程序模块化，异步化

问题：

线程共享资源，存在冲突；

容易导致死锁；

启用太多的线程，就有搞垮机器的可能

**新启线程的方式**

三种

**怎么样才能让Java里的线程安全停止工作呢**

线程自然终止：自然执行完或抛出未处理异常

stop()，resume(),suspend()已不建议使用，stop()会导致线程不会正确释放资源，suspend()容易导致死锁。

java线程是协作式，而非抢占式

调用一个线程的interrupt() 方法中断一个线程，并不是强行关闭这个线程，只是跟这个线程打个招呼，将线程的中断标志位置为true，线程是否中断，由线程本身决定。

isInterrupted() 判定当前线程是否处于中断状态。

static方法interrupted() 判定当前线程是否处于中断状态，同时中断标志位改为false。

方法里如果抛出InterruptedException，线程的中断标志位会被复位成false，如果确实是需要中断线程，要求我们自己在catch语句块里再次调用interrupt()。

**线程常用方法和线程的状态**

线程只有5种状态。整个生命周期就是这几种状态的切换。

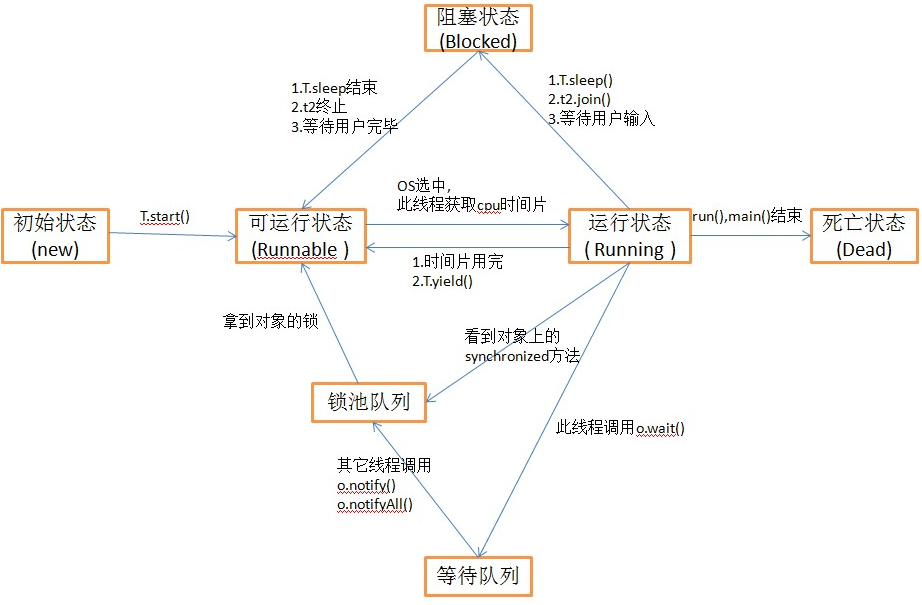
1. **新建(NEW)**：新创建了一个线程对象。

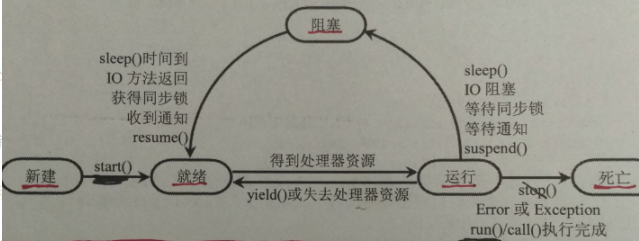
2. **可运行(RUNNABLE)**：线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu 的使用权 。

3. **运行(RUNNING)**：可运行状态(runnable)的线程获得了cpu 时间片（timeslice） ，执行程序代码。  
4. **阻塞(BLOCKED)**：阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：

(一). 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waitting queue)中。  
(二). 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。  
(三). 其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。

5. **死亡(DEAD)**：线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。





run()和start() ：run方法就是普通对象的普通方法，只有调用了start()后，Java才会将线程对象和操作系统中实际的线程进行映射，再来执行run方法。

yield() ：让出cpu的执行权，将线程从运行转到可运行状态，但是下个时间片，该线程依然有可能被再次选中运行。

**线程的优先级**

取值为1~10，缺省为5，但线程的优先级不可靠，不建议作为线程开发时候的手段

**守护线程**

和主线程共死，finally不能保证一定执行

## 线程间的共享

#### synchronized内置锁

对象锁，锁的是类的对象实例。

类锁 ，锁的是每个类的的Class对象，每个类的的Class对象在一个虚拟机中只有一个，所以类锁也只有一个。

#### volatile关键字

适合于只有一个线程写，多个线程读的场景，因为它只能确保可见性。

#### ThreadLocal

线程变量。可以理解为是个map，类型 Map<Thread,Integer>

## 线程间协作

轮询：难以保证及时性，资源开销很大，

### 等待和通知

wait() 对象上的方法

notify/notifyAll 对象上的方法

**等待和通知的标准范式**

等待方：

1. 获取对象的锁；
2. 循环里判断条件是否满足，不满足调用wait方法，
3. 条件满足执行业务逻辑

通知方来说

* 1. 获取对象的锁；
  2. 改变条件
  3. 通知所有等待在对象的线程

notify和notifyAll应该用谁？

应该尽量使用notifyAll，使用notify因为有可能发生信号丢失的的情况

**等待超时模式实现一个连接池**

假设 等待时间时长为T，当前时间now+T以后超时

long overtime = now+T;

long remain = T;//等待的持续时间

while(result不满足条件&& remain>0){

wait(remain);

remain = overtime – now;//等待剩下的持续时间

}

return result;

### join()方法

面试点

线程A，执行了线程B的join方法，线程A必须要等待B执行完成了以后，线程A才能继续自己的工作

### 调用yield() 、sleep()、wait()、notify()等方法对锁有何影响？

面试点

线程在执行yield()以后，持有的锁是不释放的

sleep()方法被调用以后，持有的锁是不释放的

调动方法之前，必须要持有锁。调用了wait()方法以后，锁就会被释放，当wait方法返回的时候，线程会重新持有锁

调动方法之前，必须要持有锁，调用notify()方法本身不会释放锁的

# 原子操作和CAS

**利用了现代处理器都支持的CAS的指令，循环这个指令，直到成功为止**

## 什么是原子操作？如何实现原子操作？

syn基于阻塞的锁的机制，1、被阻塞的线程优先级很高，2、拿到锁的线程一直不释放锁怎么办？3、大量的竞争，消耗cpu，同时带来死锁或者其他安全。

**CAS的原理**

CAS(Compare And Swap)，指令级别保证这是一个原子操作

三个运算符： 一个内存地址V，一个期望的值A，一个新值B

基本思路：如果地址V上的值和期望的值A相等，就给地址V赋给新值B，如果不是，不做任何操作。

循环（死循环，自旋）里不断的进行CAS操作

**CAS的问题**

A---》B----》A，版本号: A1🡪B2-🡪A3

CAS操作长期不成功，cpu不断的循环，

* **Jdk中相关原子操作类的使用**
  + 更新基本类型类：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference
  + 更新数组类：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray
  + 更新引用类型：AtomicReference，AtomicMarkableReference，AtomicStampedReference
  + 原子更新字段类： AtomicReferenceFieldUpdater，AtomicIntegerFieldUpdater，AtomicLongFieldUpdater

# Lock、Condition和显示锁

## 显式锁

**Lock接口和核心方法**

**Lock接口和synchronized的比较**

**synchronized 代码简洁，Lock：获取锁可以被中断，超时获取锁，尝试获取锁，读多写少用读写锁**

虽然在性能上ReentrantLock和synchronized没有什么区别，但ReentrantLock相比synchronized而言功能更加丰富，使用起来更为灵活，也更适合复杂的并发场景

**可重入锁ReentrantLock、所谓锁的公平和非公平**

如果在时间上，先对锁进行获取的请求，一定先被满足，这个锁就是公平的，不满足，就是非公平的

非公平的效率一般来讲更高

非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。

非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

公平锁和非公平锁就这两点区别，如果这两次 CAS 都不成功，那么后面非公平锁和公平锁是一样的，都要进入到阻塞队列等待唤醒。

**ReadWriteLock接口和读写锁ReentrantReadWriteLock**

ReentrantLock和Syn关键字，都是排他锁，

读写锁：同一时刻允许多个读线程同时访问，但是写线程访问的时候，所有的读和写都被阻塞，最适宜与读多写少的情况

**Condition接口**

**用Lock和Condition实现等待通知**

**可重入锁: 同一个线程中,可以加多次锁,而不用等待上一个锁的释放.常见的可重入锁**

**synchronized**

**java.util.concurrent.locks.ReentrantLock**

## 了解LockSupport工具

**park开头的方法**

负责阻塞线程

**unpark(Thread thread)方法**

负责唤醒线程

# AbstractQueue Synchronizer分析

#### 什么是AQS？学习它的必要性

AQS是JDK1.5提供的一个基于FIFO等待队列实现的一个用于实现同步器的基础框架，这个基础框架的重要性可以这么说，JCU包里面几乎所有的有关锁、多线程并发以及线程同步器等重要组件的实现都是基于AQS这个框架。AQS的核心思想是基于volatile int state这样的一个属性同时配合Unsafe工具对其原子性的操作来实现对当前锁的状态进行修改。当state的值为0的时候，标识改Lock不被任何线程所占有

<http://www.importnew.com/24006.html>

<https://blog.csdn.net/a724888/article/details/60955965>

<https://javadoop.com/post/AbstractQueuedSynchronizer>

**AQS使用方式和其中的设计模式**

继承，模板方法设计模式

AQS属性简介

State: 初始值为0,表示资源空闲,>0表示有线程持有锁,<0会抛异常

Head:头结点,也可以理解为持有锁的线程结点.

Tail:阻塞的尾结点,每次有新的结点进来,都插入到尾结点.

exclusiveOwnerThread:当前持有锁的线程

Node属性简介

volatile int waitStatus:>0表示线程取消等待,

-1:当前节点的后续节点需要被唤醒.是被后置节点将状态从0设为1

-2:与condition有关,当其他线程调用了condition的signal方法后,节点从等待队列转移到同步队列,等待获取资源.

-3:与共享模式相关,在共享模式下当前线程处于可运行状态.

插入新节点,会插入到尾节点之后,尾节点的waitstatus为0,同时将前置节点的waitstatus置为-1,表示有后续节点需要通知.

volatile Node prev:前置节点

volatile Node next:后置节点

volatile Thread thread:持有线程本身

Node nextWaiter 用于实现条件队列的单向链表

**了解其中的方法**

模板方法：

独占式获取

accquire

acquireInterruptibly

tryAcquireNanos

共享式获取

acquireShared

acquireSharedInterruptibly

tryAcquireSharedNanos

独占式释放锁

release

共享式释放锁

releaseShared

需要子类覆盖的流程方法

独占式获取 tryAcquire

独占式释放 tryRelease

共享式获取 tryAcquireShared

共享式释放 tryReleaseShared

这个同步器是否处于独占模式 isHeldExclusively

同步状态state：

getState:获取当前的同步状态

setState：设置当前同步状态

compareAndSetState 使用CAS设置状态，保证状态设置的原子性

#### AQS中的数据结构-节点和同步队列

竞争失败的线程会打包成Node放到同步队列，Node可能的状态里：

***CANCELLED 1：***线程等待超时或者被中断了，需要从队列中移走

***SIGNAL -1 ：***后续的节点等待状态，当前节点，通知后面的节点去运行

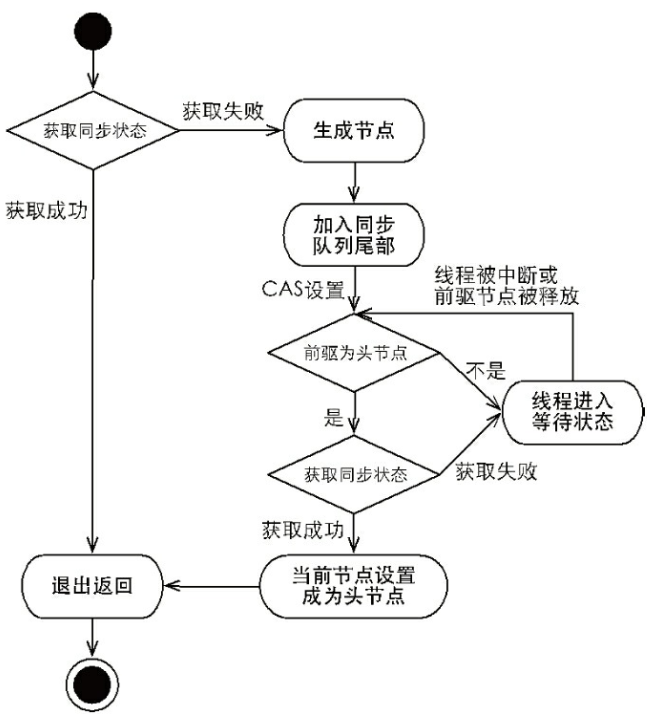
CONDITION -2 :当前节点处于等待队列

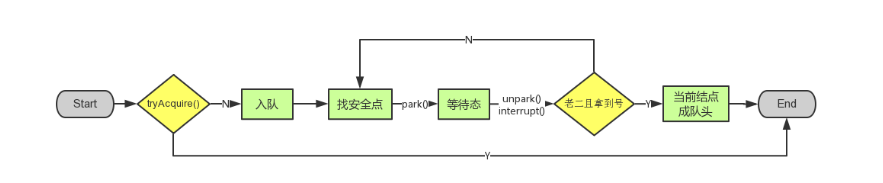
***PROPAGATE：共享，表示状态要往后面的节点传播***

1. ***表示初始状态***

#### 节点在同步队列中的增加和移出

#### 独占式同步状态获取与释放





#### 其他同步状态获取与释放

#### Condition分析

# 并发工具类和并发容器

## Fork-Join

**什么是分而治之？**

规模为N的问题，N<阈值，直接解决，N>阈值，将N分解为K个小规模子问题，子问题互相对立，与原问题形式相同，将子问题的解合并得到原问题的解

动态规范

**工作密取**

workStealing

**Fork/Join使用的标准范式**

## 常用的并发工具类

**CountDownLatch**

作用：是一组线程等待其他的线程完成工作以后在执行，加强版join

await用来等待，countDown负责计数器的减一

**CyclicBarrier**

让一组线程达到某个屏障，被阻塞，一直到组内最后一个线程达到屏障时，屏障开放，所有被阻塞的线程会继续运行CyclicBarrier(int parties)

CyclicBarrier(int parties, Runnable barrierAction)，屏障开放，barrierAction定义的任务会执行

CountDownLatch和CyclicBarrier辨析

1、countdownlatch放行由第三者控制，CyclicBarrier放行由一组线程本身控制  
2、countdownlatch放行条件》=线程数，CyclicBarrier放行条件=线程数

**Semaphore**

控制同时访问某个特定资源的线程数量，用在流量控制

**Exchange**

两个线程间的数据交换，

**Callable、Future和FutureTask**

isDone，结束，正常还是异常结束，或者自己取消，返回true；

isCancelled 任务完成前被取消，返回true；

cancel（boolean）：

1. 任务还没开始，返回false
2. 任务已经启动，cancel（true），中断正在运行的任务，中断成功，返回true，cancel（false），不会去中断已经运行的任务
3. 任务已经结束，返回false

包含图片和文字的文档的处理：图片（云上），可以用future去取图片，主线程继续解析文字。

### ConcurrentHashMap

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

扩容方法导致的互为next.

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

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

## 预备知识

### Hash

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

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

ConcurrentHashMap在发生hash冲突时采用了链地址法。

md4,md5,sha-hash算法也属于hash算法，又称摘要算法。

### 位运算

**int类型的位**

**高位 低位**

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

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

由上面的表格可以看出，数字类型在数字渐渐变大时，是由低位慢慢向高位扩展的。

Java实际保存int型时 正数 第31位 =0 负数：第31位=1

常用位运算有：

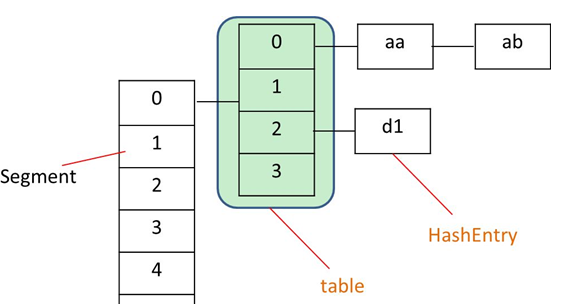
* 位与 & (1&1=1 1&0=0 0&0=0)
* 位或 | (1|1=1 1|0=1 0|0=0)
* 位非 ~ （ ~1=0 ~0=1）
* 位异或 ^ (1^1=0 1^0=1 0^0=0)
* <<有符号左移 >>有符号的右移 >>>无符号右移 例如：8 << 2 = 32 8>>2 = 2
* 取模的操作 a % (Math.pow(2,n)) 等价于 a&( Math.pow(2,n)-1)

**位运算适用**：权限控制，物品的属性非常多时的保存

## 1.7中原理和实现

### ConcurrentHashMap中的数据结构

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



**面试常问：**

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

答：ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了**锁分离**技术。它使用了多个锁来控制对hash表的不同部分进行的修改。内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的hash table，只要多个修改操作发生在不同的段上，它们就可以并发进行。Segment默认16,初始化后不可更改.

### 初始化做了什么事？

初始化有三个参数

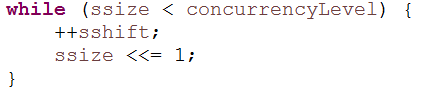
**initialCapacity**：初始容量大小 ，默认16。

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

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

**构造方法中部分代码解惑：**

1、



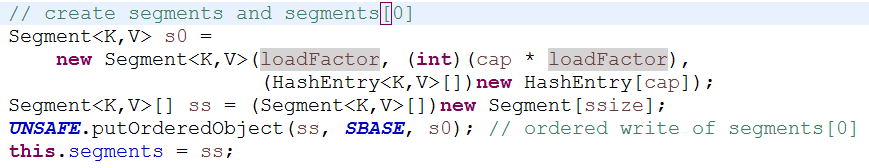
保证Segment数组的大小，一定为2的幂，例如用户设置并发度为17，则实际Segment数组大小则为32

2、



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

3、



初始化Segment数组，并实际只填充Segment数组的第0个元素。

4、

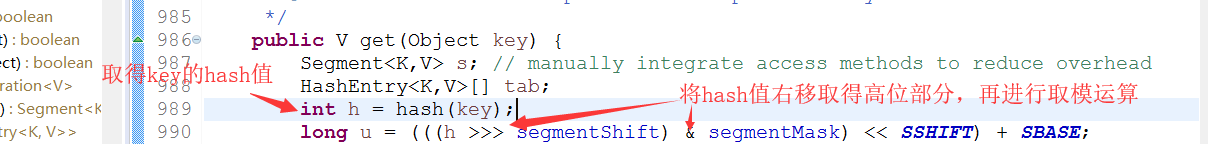


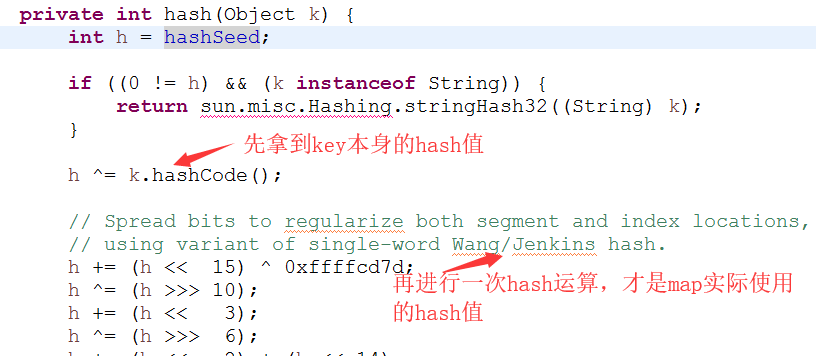
用于定位元素所在segment。segmentShift表示偏移位数，通过前面的int类型的位的描述我们可以得知，int类型的数字在变大的过程中，低位总是比高位先填满的，为保证元素在segment级别分布的尽量均匀，计算元素所在segment时，总是取hash值的高位进行计算。segmentMask作用就是为了利用位运算中取模的操作： a % (Math.pow(2,n)) 等价于 a&( Math.pow(2,n)-1)

### 在get和put操作中，是如何快速定位元素放在哪个位置的？

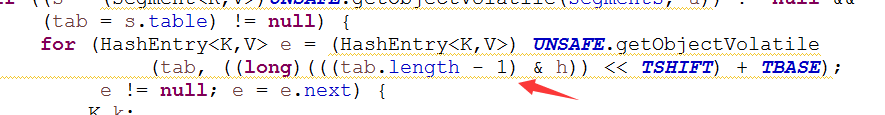
对于某个元素而言，一定是放在某个segment元素的某个table元素中的，所以在定位上，

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





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

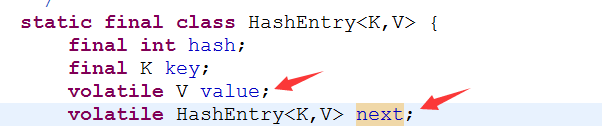


### get（）方法

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

**在高并发下的情况下如何保证取得的元素是最新的？**

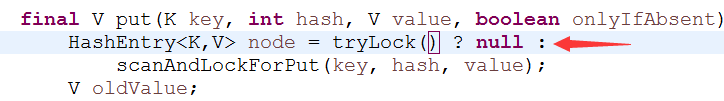
答：用于存储键值对数据的HashEntry，在设计上它的成员变量value等都是volatile类型的，这样就保证别的线程对value值的修改，get方法可以马上看到。



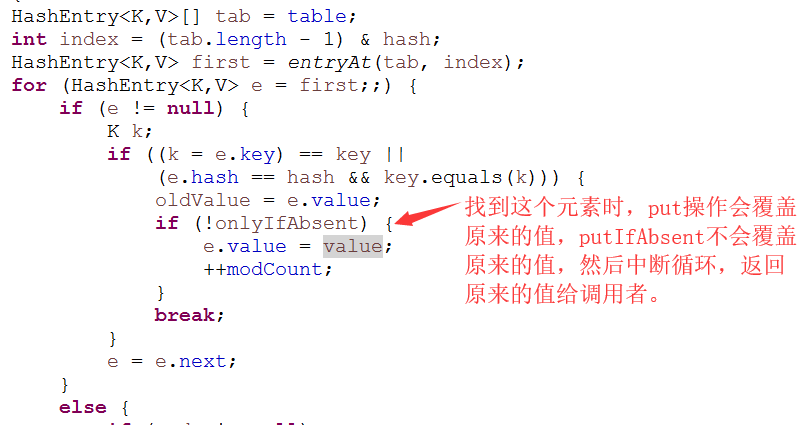
### put()方法

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

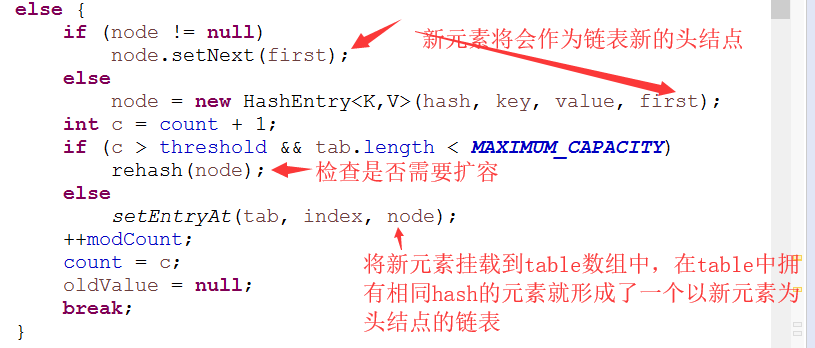
1. 对Segment 加锁



3、定位所在的table元素，并扫描table下的链表，**找到时：**

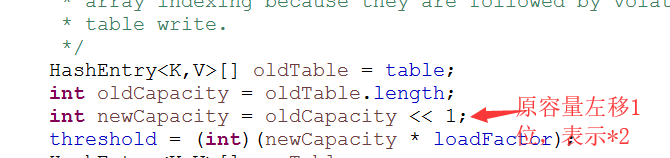


**没有找到时：**



### 扩容操作

Segment 不扩容，扩容下面的table数组，每次都是将数组翻倍



**带来的好处**

假设原来table长度为4，那么元素在table中的分布是这样的：

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

扩容后table长度变为8，那么元素在table中的分布变成：

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

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

### size方法

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

### 弱一致性

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

## 1.8

### 与1.7相比的重大变化

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

### 主要数据结构和关键变量

Node类存放实际的key和value值。

sizeCtl：

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

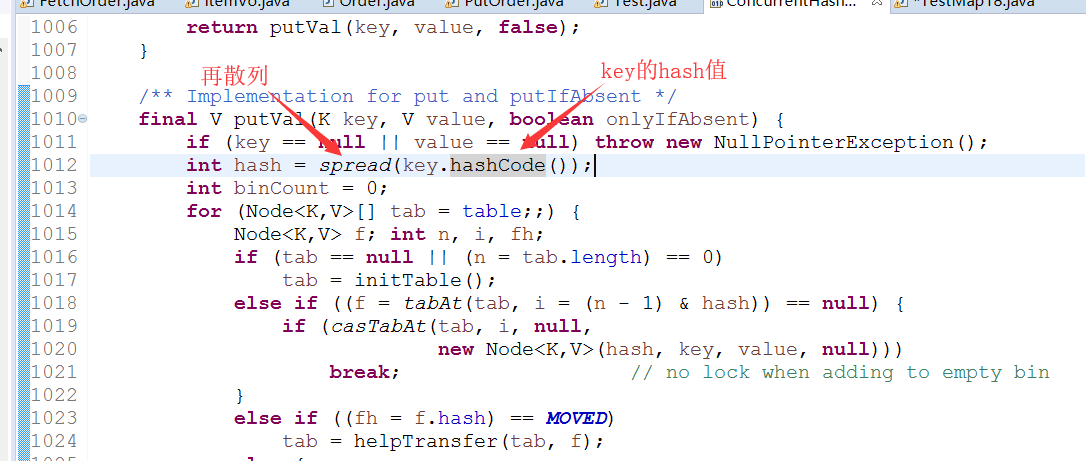
正数：0 表示还没有被初始化，>0的数，初始化或者是下一次进行扩容的阈值

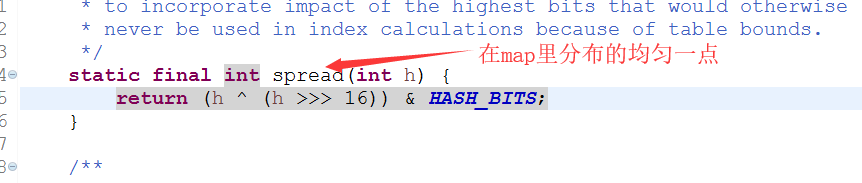
TreeNode 用在红黑树，表示树的节点, TreeBin是实际放在table数组中的，代表了这个红黑树的根。

### 初始化做了什么事？

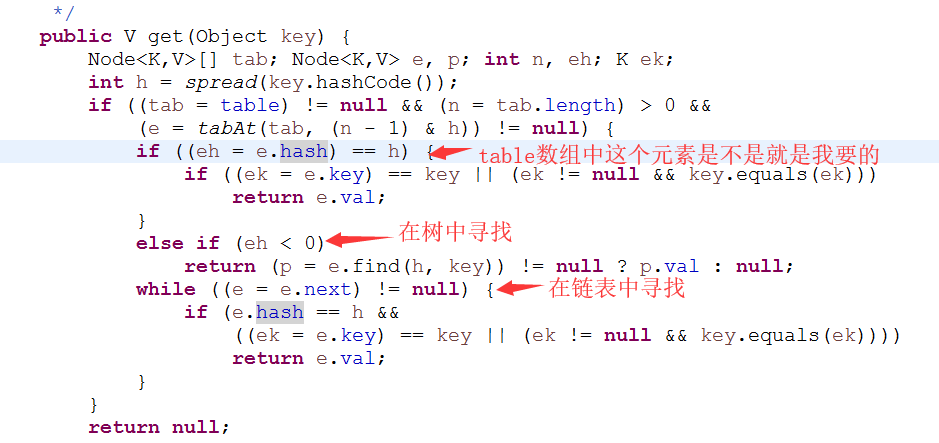
只是给成员变量赋值，put时进行实际数组的填充

### 在get和put操作中，是如何快速定位元素放在哪个位置的？



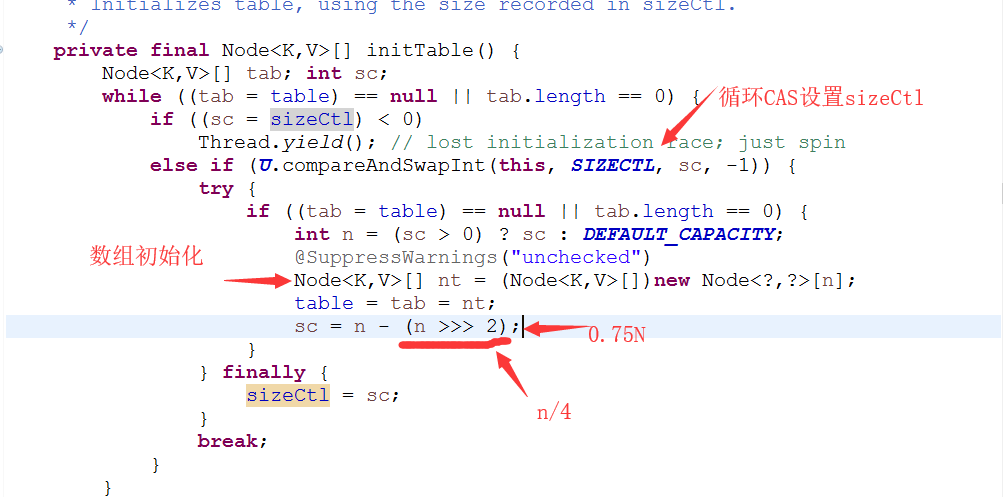


### get（）方法

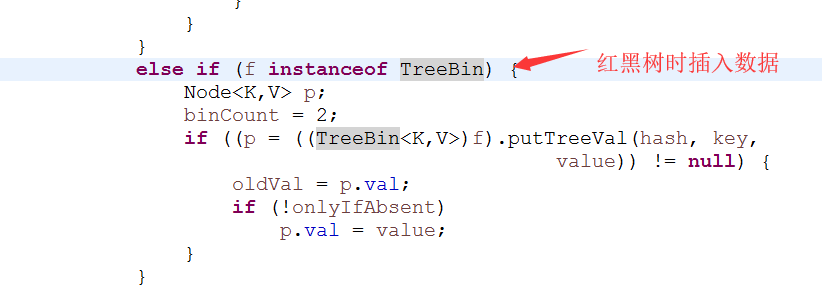


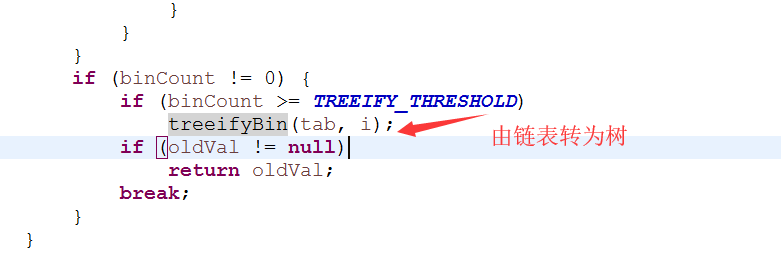
### put()方法

数组的实际初始化









### 扩容操作

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

### size方法

估计的大概数量，不是精确数量

### 一致性

弱一致

## 更多的并发容器

## ConcurrentSkipListMap 和 ConcurrentSkipListSet

TreeMap和TreeSet有序的容器，这两种容器的并发版本

### 跳表

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

## ConcurrentLinkedQueue

无界非阻塞队列，底层是个链表，遵循先进先出原则。

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

## 写时复制容器

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

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

存在内存占用问题。

## 阻塞队列

## 概念、生产者消费者模式

1)当队列满的时候，插入元素的线程被阻塞，直达队列不满。

2)队列为空的时候，获取元素的线程被阻塞，直到队列不空。

### 生产者和消费者模式

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

## 常用方法

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

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

## 常用阻塞队列

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

按照先进先出原则，要求设定初始大小

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

按照先进先出原则，可以不设定初始大小，Integer.Max\_Value

**ArrayBlockingQueue和LinkedBlockingQueue不同：**

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

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

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

·**DelayQueue**：一个使用优先级队列实现的无界阻塞队列。

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

·**SynchronousQueue**：一个不存储元素的阻塞队列。

每一个put操作都要等待一个take操作

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

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

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

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

## 阻塞队列的实现原理

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

# 线程池和Executor框架

#### 什么是线程池？为什么要用线程池？

1. 降低资源的消耗。降低线程创建和销毁的资源消耗；
2. 提高响应速度：线程的创建时间为T1，执行时间T2,销毁时间T3，免去T1和T3的时间
3. 提高线程的可管理性。

#### 实现一个我们自己的线程池

1. 线程必须在池子已经创建好了，并且可以保持住，要有容器保存多个线程；

2、线程还要能够接受外部的任务，运行这个任务。容器保持这个来不及运行的任务.

### JDK中的线程池和工作机制

### 线程池的创建

ThreadPoolExecutor，jdk所有线程池实现的父类

#### 各个参数含义

**int** corePoolSize ：线程池中核心线程数，< corePoolSize ，就会创建新线程，= corePoolSize ，这个任务就会保存到BlockingQueue，如果调用prestartAllCoreThreads（）方法就会一次性的启动corePoolSize 个数的线程。

**int** maximumPoolSize, 允许的最大线程数，BlockingQueue也满了，< maximumPoolSize时候就会再次创建新的线程

**long** keepAliveTime, 线程空闲下来后，存活的时间，这个参数只在> corePoolSize才有用

TimeUnit unit, 存活时间的单位值

BlockingQueue<Runnable> workQueue, 保存任务的阻塞队列

ThreadFactory threadFactory, 创建线程的工厂，给新建的线程赋予名字

RejectedExecutionHandler handler ：饱和策略

AbortPolicy ：直接抛出异常，默认；

CallerRunsPolicy：用调用者所在的线程来执行任务

DiscardOldestPolicy：丢弃阻塞队列里最老的任务，队列里最靠前的任务

DiscardPolicy ：当前任务直接丢弃

实现自己的饱和策略，实现RejectedExecutionHandler接口即可

### 提交任务

execute(Runnable command) 不需要返回

Future<T> submit(Callable<T> task) 需要返回

### 关闭线程池

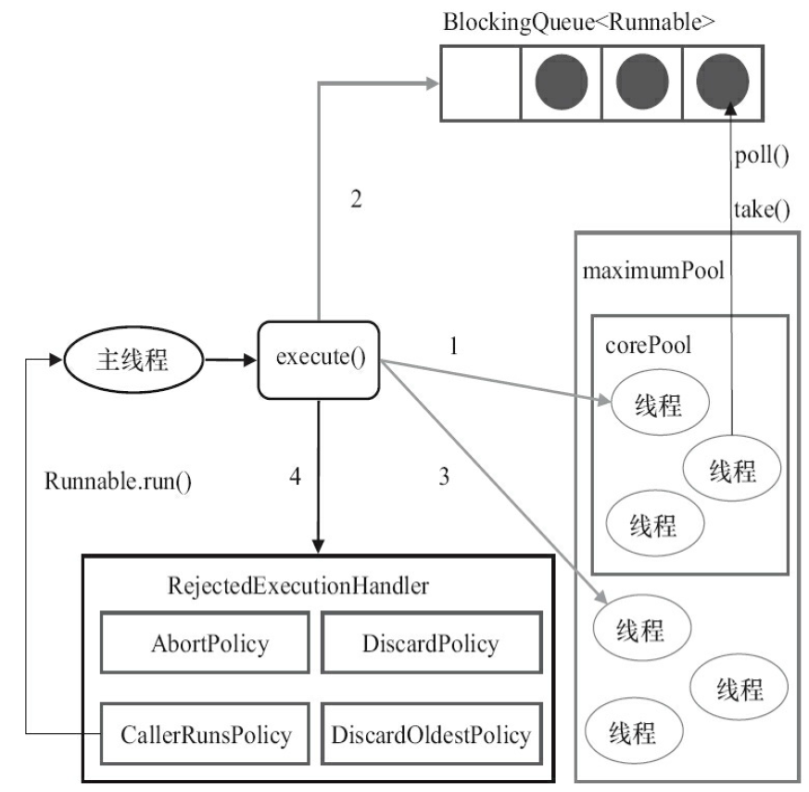
shutdown(),shutdownNow();

shutdownNow():设置线程池的状态，还会尝试停止正在运行或者暂停任务的线程

shutdown()设置线程池的状态，只会中断所有没有执行任务的线程

### 工作机制





### 合理配置线程池

根据任务的性质来：计算密集型（CPU），IO密集型，混合型

计算密集型：加密，大数分解，正则…….， 线程数适当小一点，最大推荐：机器的Cpu核心数+1，为什么+1，防止页缺失，(机器的Cpu核心=Runtime.*getRuntime*().availableProcessors();)

IO密集型：读取文件，数据库连接，网络通讯, 线程数适当大一点，机器的Cpu核心数\*2,

混合型：尽量拆分，IO密集型>>计算密集型，拆分意义不大，IO密集型~计算密集型

队列的选择上，应该使用有界，无界队列可能会导致内存溢出，OOM

### 预定义的线程池

#### FixedThreadPool

创建固定线程数量的，适用于负载较重的服务器，使用了无界队列

#### SingleThreadExecutor

创建单个线程，需要顺序保证执行任务，不会有多个线程活动，使用了无界队列

#### CachedThreadPool

会根据需要来创建新线程的，执行很多短期异步任务的程序，使用了SynchronousQueue

#### WorkStealingPool（JDK7以后）

基于ForkJoinPool实现

#### ScheduledThreadPoolExecutor

需要定期执行周期任务，Timer不建议使用了。

newSingleThreadScheduledExecutor：只包含一个线程，只需要单个线程执行周期任务，保证顺序的执行各个任务

newScheduledThreadPool 可以包含多个线程的，线程执行周期任务，适度控制后台线程数量的时候

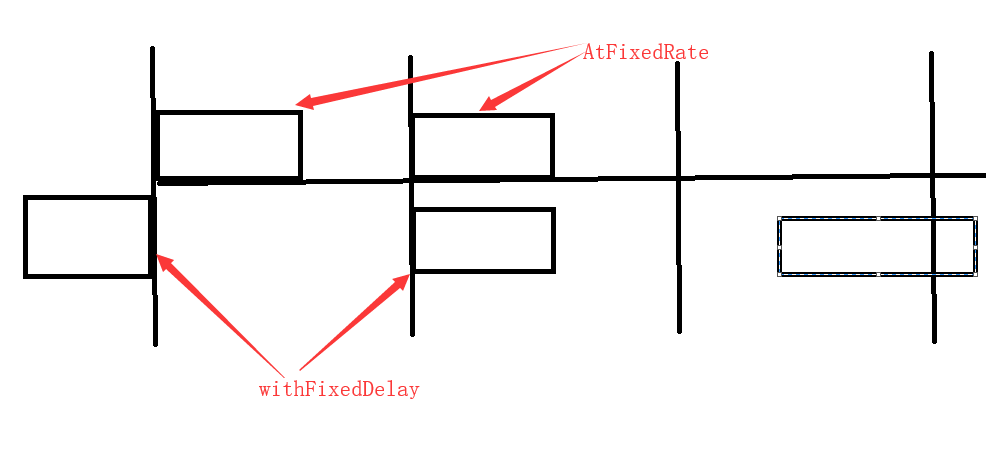
方法说明：

schedule：只执行一次，任务还可以延时执行

scheduleAtFixedRate：提交固定时间间隔的任务

scheduleWithFixedDelay：提交固定延时间隔执行的任务

两者的区别：



scheduleAtFixedRate任务超时：

规定60s执行一次，有任务执行了80S，下个任务马上开始执行

第一个任务 时长 80s，第二个任务20s，第三个任务 50s

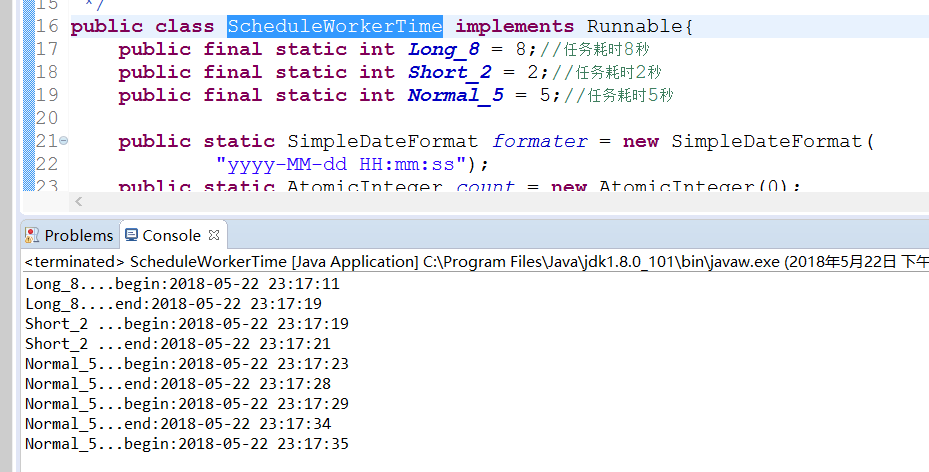
第一个任务第0秒开始，第80S结束；

第二个任务第80s开始，在第100秒结束；

第三个任务第120s秒开始，170秒结束

第四个任务从180s开始

参加代码：ScheduleWorkerTime类，执行效果如图：



建议在提交给ScheduledThreadPoolExecutor的任务要住catch异常。

### Executor框架

### 了解CompletionService

# 实现原理和java内存模型

# 线程安全

## 类的线程安全定义

如果多线程下使用这个类，不过多线程如何使用和调度这个类，这个类总是表示出正确的行为，这个类就是线程安全的。

类的线程安全表现为：

* 操作的原子性
* 内存的可见性

不做正确的同步，在多个线程之间共享状态的时候，就会出现线程不安全。

## 怎么才能做到类的线程安全？

### 栈封闭

所有的变量都是在方法内部声明的，这些变量都处于栈封闭状态。

### 无状态

没有任何成员变量的类，就叫无状态的类

### 让类不可变

让状态不可变，两种方式：

1，加final关键字，对于一个类，所有的成员变量应该是私有的，同样的只要有可能，所有的成员变量应该加上final关键字，但是加上final，要注意如果成员变量又是一个对象时，这个对象所对应的类也要是不可变，才能保证整个类是不可变的。

2、根本就不提供任何可供修改成员变量的地方，同时成员变量也不作为方法的返回值

### volatile

保证类的可见性，最适合一个线程写，多个线程读的情景，

### 加锁和CAS

### 安全的发布

类中持有的成员变量，特别是对象的引用，如果这个成员对象不是线程安全的，通过get等方法发布出去，会造成这个成员对象本身持有的数据在多线程下不正确的修改，从而造成整个类线程不安全的问题。

### TheadLocal

### Servlet

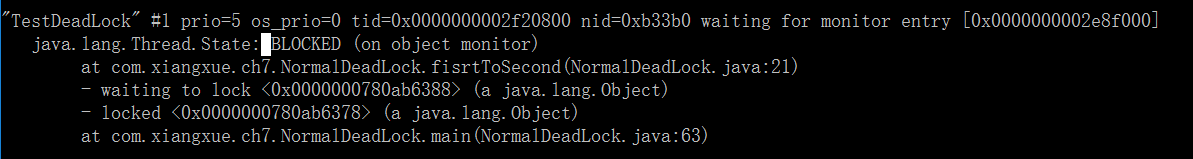
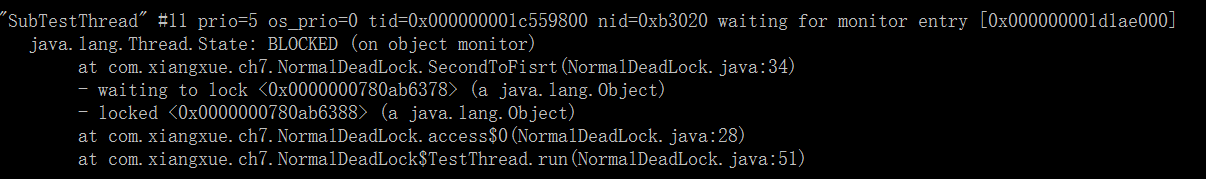
不是线程安全的类，为什么我们平时没感觉到，：2、在需求上，很少有共享的需求，第二，接收到了请求，返回应答的时候，都是由一个线程来负责的。

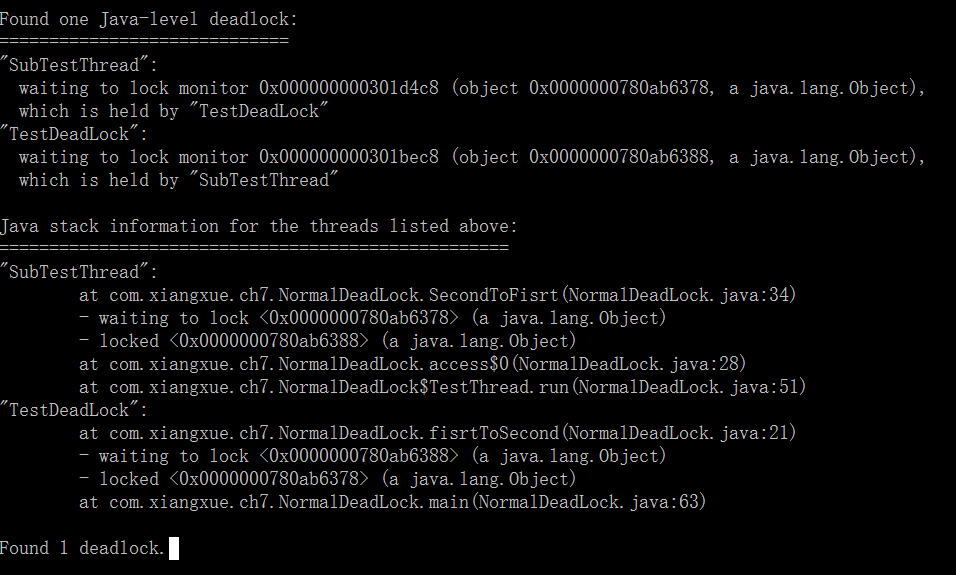
## 死锁

资源一定是多于1个，同时小于等于竞争的线程数，资源只有一个，只会产生激烈的竞争。

死锁的根本成因：**获取锁的顺序不一致导致。**

### 简单的





怀疑发送死锁：

通过jps 查询应用的 id，

再通过jstack id 查看应用的锁的持有情况

**解决办法：保证加锁的顺序性**

#### 动态的

动态顺序死锁，在实现时按照某种顺序加锁了，但是因为外部调用的问题，导致无法保证加锁顺序而产生的。

解决：

1. 通过内在排序，保证加锁的顺序性
2. 通过尝试拿锁，也可以。

## 其他安全问题

### 活锁

尝试拿锁的机制中，发生多个线程之间互相谦让，不断发生拿锁，释放锁的过程。

解决办法：每个线程休眠随机数，错开拿锁的时间。

### 线程饥饿

低优先级的线程，总是拿不到执行时间

## 性能和思考

使用并发的目标是为了提高性能，引入多线程后，其实会引入额外的开销，如线程之间的协调、增加的上下文切换，线程的创建和销毁，线程的调度等等。**过度**的使用和不恰当的使用，会导致多线程程序甚至比单线程还要低。

衡量应用的程序的性能：服务时间，延迟时间，吞吐量，可伸缩性等等，其中服务时间，延迟时间（多快），吞吐量（处理能力的指标，完成工作的多少）。多快和多少，完全独立，甚至是相互矛盾的。

对服务器应用来说：多少（可伸缩性，吞吐量）这个方面比多快更受重视。

我们做应用的时候：

1. **先保证程序正确，确实达不到要求的时候，再提高速度。（黄金原则）**
2. **一定要以测试为基准。**

一个应用程序里，串行的部分是永远都有的。

Amdahl定律 ： 1/(F+(1-N)/N) F:必须被串行部分,程序最好的结果， 1/F。

### 影响性能的因素

#### 上下文切换

是指CPU 从一个进程或线程切换到另一个进程或线程。一次上下文切换花费5000~10000个时钟周期，几微秒。在上下文切换过程中，CPU会停止处理当前运行的程序，并保存当前程序运行的具体位置以便之后继续运行。从这个角度来看，上下文切换有点像我们同时阅读几本书，在来回切换书本的同时我们需要记住每本书当前读到的页码。

上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。

### 内存同步

一般指加锁，对加锁来说，需要增加额外的指令，这些指令都需要刷新缓存等等操作。

### 阻塞

会导致线程挂起【挂起：挂起进程在操作系统中可以定义为暂时被淘汰出内存的进程，机器的资源是有限的，在资源不足的情况下，操作系统对在内存中的程序进行合理的安排，其中有的进程被暂时调离出内存，当条件允许的时候，会被操作系统再次调回内存，重新进入等待被执行的状态即就绪态，系统在超过一定的时间没有任何动作】。很明显这个操作包括两次额外的上下文切换。

### 减少锁的竞争

#### 减少锁的粒度

使用锁的时候，锁所保护的对象是多个，当这些多个对象其实是独立变化的时候，不如用多个锁来一一保护这些对象。但是如果有同时要持有多个锁的业务方法，要注意避免发生死锁

#### 缩小锁的范围

对锁的持有实现快进快出，尽量缩短持由锁的的时间。将一些**与锁无关的代码**移出锁的范围，特别是一些耗时，可能阻塞的操作

##### 避免多余的缩减锁的范围

两次加锁之间的语句非常简单，导致加锁的时间比执行这些语句还长，这个时候应该进行锁粗化—扩大锁的范围。

#### 锁分段

ConcurrrentHashMap就是典型的锁分段。

#### 替换独占锁

在业务允许的情况下：

1. 使用读写锁，
2. 用自旋CAS
3. 使用系统的并发容器

## 线程安全的单例模式

### 双重检查锁定



解决办法，加**volatile关键字**

### 解决之道

#### 懒汉式

类初始化模式，也叫延迟占位模式。在单例类的内部由一个私有静态内部类来持有这个单例类的实例。

延迟占位模式还可以用在多线程下实例域的延迟赋值。

#### 饿汉式

在声明的时候就new这个类的实例，因为在JVM中，对类的加载和类初始化，由虚拟机保证线程安全。

或者使用枚举

# 并发项目实践

## 架构师是什么？

一个软件项目开发过程中，将客户的需求转换为规范的开发计划及文本，并制定这个项目的总体架构，指导整个开发团队完成这个计划的那个人，就是 架构师。一般是一个项目里的最资深的专业技术人员，可以说架构师首先一定是个Java高级开发人员。

### 架构设计、软件开发

#### 确认需求

需求规格说明书必须得到架构师的认可。架构师需要和分析人员反复交流，以保证自己完整并准确地理解用户需求。

#### 系统分解

依据用户需求，整个系统是否需要分层，如何进行分层,架构师将系统整体分解为更小的子系统和组件，从而形成不同的逻辑层或服务。随后，架构师会确定各层的接口，层与层相互之间的关系。架构师不仅要对整个系统分层，进行“纵向”分解，还要对同一逻辑层分块，进行“横向”分解。软件架构师的功力基本体现于此。

#### 技术选型

架构师通过对系统的一系列的分解，最终形成了软件的整体架构。技术选择主要取决于软件架构，就是不断找到系统的瓶颈和弱点，采用分而治之、缓存、异步、集群等手段逐渐化解，并平衡处理系统各项要求（性能、安全、可用性、伸缩性、扩展性…）的过程。由此形成了架构。

**什么样的架构才是好的架构？**

答案：是适用于当前业务和团队成员，并保留适当前瞻性(最多半年的业务增长)的就是好架构。

#### 制定技术规格说明

架构师在项目开发过程中，是技术权威。他需要协调所有的开发人员，与开发人员一直保持沟通，始终保证开发者依照它的架构意图去实现各项功能。

### 开发管理

#### 深深介入开发的方方面面

规划产品路线，估算人力资源和时间资源，安排分工，确定里程碑点，指导工程师工作都需要架构师参与。

### 沟通协调

#### 与用户，与产品，与上级，与团队成员

与用户沟通需求；与产品讨论需求，安排进度；与上级汇报进度，争取资源；团结团队成员等等

## 架构师的方方面面

### 作用

#### 设计架构

负责架构设计，负责架构的实施落地，演化发展，推广重构。

#### 救火

系统出现难解决的故障，架构师出手解决。

#### 布道

与他人分享自己的知识，培训团队成员。

### 效果

#### 攻关

挑战整个项目中最具技术难度和挑战性的模块。

#### 信念

不管项目有多么艰难复杂，架构师在，团队成员就会坚信，项目一定能顺利完成。

### 职责

#### 产品架构

整个产品的技术架构

#### 基础服务架构

负责开发基础框架，公共组件，通用服务等平台类产品。

## 我们需要做什么

### 提高性能，采用多线程，屏蔽细节

封装线程池和阻塞队列

### 每个批量任务拥有自己的上下文环境

需要一个并发安全的容器保存每个任务的具体信息

### 自动清除已完成和过期任务

定时轮询？不够优雅，使用延迟队列更好

### 具体实现 – 可查询进度的并发任务执行框架

#### 用户业务方法的结果？

#### 如何执行用户的业务方法？

#### 用户如何提交他的工作任务和查询任务进度？