**多线程**

# 基本概念

线程状态图：



**线程共包括以下5种状态。**

**1. 新建状态(New): 线程对象被创建后，就进入了新建状态。例如，Thread thread = new Thread()。**

**2. 就绪状态(Runnable): 也被称为“可执行状态”。线程对象被创建后，其它线程调用了该对象的start()方法，从而来启动该线程。例如，thread.start()。处于就绪状态的线程，随时可能被CPU调度执行。**

**3. 运行状态(Running) : 线程获取CPU权限进行执行。需要注意的是，线程只能从就绪状态进入到运行状态。**

**4. 阻塞状态(Blocked) : 阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：**

**(01) 等待阻塞 -- 通过调用线程的wait()方法，让线程等待某工作的完成。**

**(02) 同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。**

**(03) 其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。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，线程重新转入就绪状态。**

**5. 死亡状态(Dead) : 线程执行完了或者因异常退出了run()方法，该线程结束生命周期。**

# 多线程的代价

## 上下文切换开销

当CPU从执行一个线程切换到执行另外一个线程的时候，它需要先存储当前线程的本地的数据，程序指针等，然后载入另一个线程的本地数据，程序指针等，最后才开始执行。这种切换称为“上下文切换”(“context switch”)。CPU会在一个上下文中执行一个线程，然后切换到另外一个上下文中执行另外一个线程。

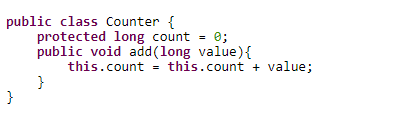
上下文切换并不廉价。如果没有必要，应该减少上下文切换的发生。

例如：服务端响应客户端的简单请求（耗时很小），这里没有使用线程池，导致大量的请求到服务端，然后服务端会给每个请求创建一个线程，然后导致大量的线程上下文切换，严重影响性能。

## 增加资源消耗

1. 每个线程都需要占用CPU资源。
2. 都需要占用一定内存。

# 竞态条件与临界区



竞态条件：即同时竞争获取count变量。

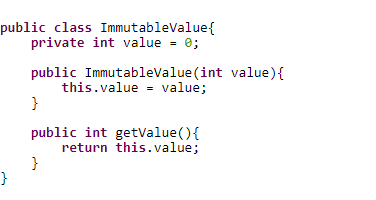
临界区：add方法会产生竞争。

说明：由于add操作非原子性，所以在多线程下，会导致计数不准确，例如count初始为1，A,B两个线程执行add，理想状态是count=3，但是A,B可能同时读取count=1，然后最终结果是2；这里涉及到了JMM（java内存模型）。

# 线程安全与不可变性

当多个线程同时访问同一个资源，并且其中的一个或者多个线程对这个资源进行了写操作，才会产生竞态条件。多个线程同时读同一个资源不会产生竞态条件。

我们可以通过创建不可变的共享对象来保证对象在线程间共享时不会被修改，从而实现线程安全。如下示例：



ImmutableValue类的成员变量value是通过构造函数赋值的，并且在类中没有set方法。这意味着一旦ImmutableValue实例被创建，value变量就不能再被修改，这就是不可变性。但你可以通过getValue()方法读取这个变量的值。

引申：多线程下线程安全的单利模式（Singleton）

# Java内存模型

1. 线程间的通信
   1. 共享内存：通过共享对象来完成隐式通信；java并发就是用的共享内存。
   2. 消息传递：典型的消息传递为线程间的wait/notify。
2. Java内存模型



* 1. 线程间的共享变量存储在主存，每个线程都会有这个共享变量的本地拷贝（带来线程不安全的原因，例如A还没来得及刷入主存，B就获取了老值）
  2. A线程对X赋值1，然后将X刷入主存；B线程读取主存，然后将X的值存储在本地内存。
  3. JMM通过控制主存与每个线程的本地内存之间的交互，来提供内存可见性保证。

1. JVM对JMM的实现



* 1. 每个线程都会有自己的线程栈，然后分配一定的本地内存（JVM中线程太多，也会导致内存不够用）。
  2. 基本数据类型直接保存在线程栈中；对象存储在堆中，栈中只存储一个引用（即存储对象的内存物理地址）；
  3. 基本数据类型，线程间如果要共享，必须通过主存来实现互相通信；共享对象则不用，因为对象都在堆（Object3，如果有变更操作，需要对obj3加锁，否则会导致数据不一致），每个线程中保存一个对象的引用。

1. 内存屏障
   1. 读屏障：强制线程去主存里面获取最新数据。（在指令前插入Load Barrier，可以让高速缓存中的数据失效，强制从新从主内存加载数据）
   2. 写屏障：强制线程将写后的最新数据从线程本地内存刷入主存。（在指令后插入Store Barrier，能让写入缓存中的最新数据更新写入主内存，让其他线程可见）

# Java同步块（SyncMethod）

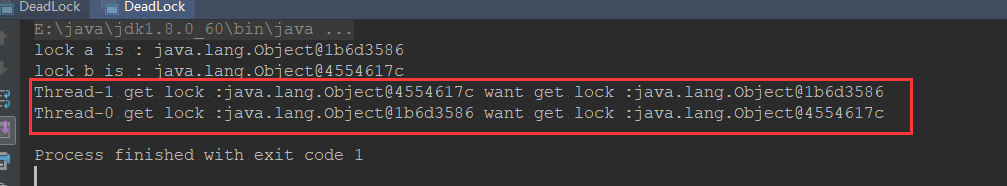
1. 实例方法同步
   1. 加的是对象锁，多线程使用不同的对象不用等待，使用了相同对象才等待。
2. 静态方法同步
   1. 加的是类锁，所线程只要调用了该方法，都必须等待。
3. 实例方法同步块（区别为方法可以并发访问，只有到同步块才等待）
   1. 同步块锁的是对象，同实例方法同步
   2. 同步块锁的是类，同静态方法同步
4. 静态方法同步块（区别为方法可以并发访问，只有到同步块才等待）
   1. 同步块锁对象，同实例方法同步
   2. 同步块锁类，同静态方法同步

# 死锁/避免方法

## java死锁

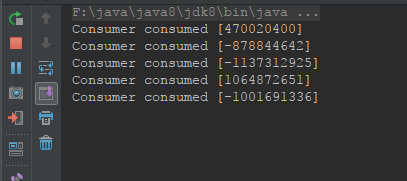
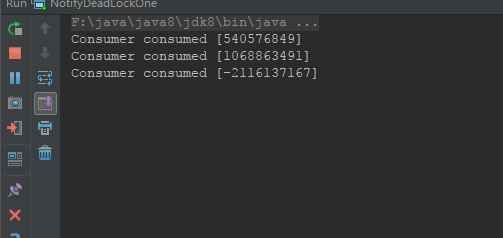
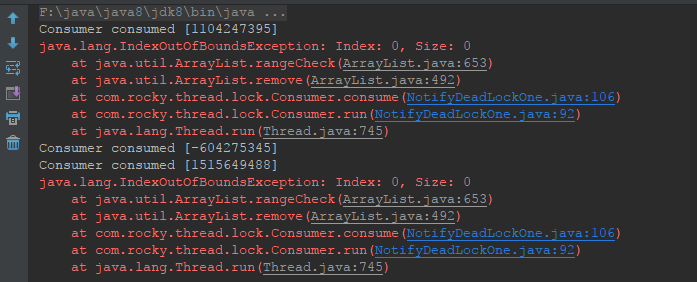
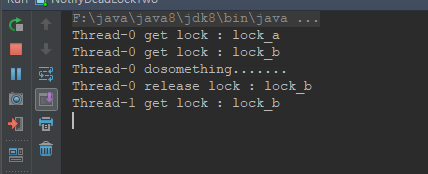
### 非顺序获取锁(DeadLock)

线程1获取A后，再次获取B，线程先获取B，再次获取A，然后两个线程互相等待。



解决方法：**顺序获取锁**，尽量避免在获取了一个临界资源后，再次获取另外一个，即**避免锁嵌套**。

### Notify导致的死锁

1. 生产者/消费者模式，在单个线程下使用notify唤醒对方线程，一切运行正常,每秒消费者消费一个消息。(NotifyDeadLockOne)
   1. 
2. 生产者/消费者模式，在多线程下使用notify唤醒对方线程，会造成互相等待唤醒而造成死锁,成功消费了3个消息就卡住不动了。(NotifyDeadLockOne)
   1. 
   2. 原因：假设有P1-P3三个生产者，C1-C3三个消费者，第一次P1放入一个消息，然后P1进入等待队列，并唤醒C1进行消费，C1消费完后进入等待，唤醒P2进行生产，P2生产完后使用notify随机唤醒了P3，P3判断队列里面有消息，然后自己又会进入等待队列，这样就导致了所有的线程都在等待。（notify是随机唤醒一个线程，如果随机唤醒了生产者，就会导致死锁）
   3. 解决方案：使用notifyAll来唤醒所有等待线程。
   4. 引伸问题：如果代码中的while判断换成if判断是否可行？
      1. 如果换成if多线程并发下会导致数组越界(单个生产者/消费者不会有问题，因为消费者被唤醒了，队列里面肯定有数据；多线程下，notifyall时候，可能消费者再次获取锁，如果使用if就不会再次判断队列是否有值，而直接或消费)
      2. 
3. Notify另外一种死锁情况，线程1获取了A,B两个锁，然后调用b.wait()释放B锁，进入等待队列，线程2获取B锁，然后再获取A锁，造成死锁。(NotifyDeadLockTwo)
   1. 
   2. 解决方案：类似非顺序获取锁，因此还是需要顺序获取锁，以及尽量避免锁嵌套获取。

## 数据库死锁

### 两个事务互相等待

出现原因：

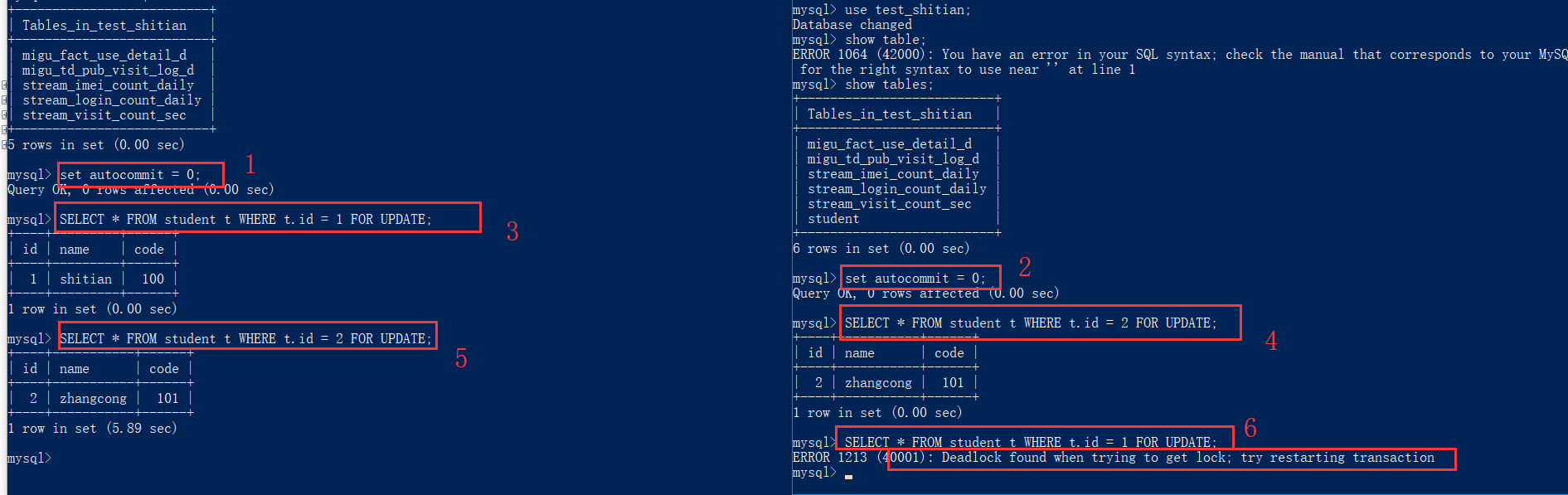
一个用户A 访问表A（锁住了表A），然后又访问表B；另一个用户B 访问表B（锁住了表B），然后企图访问表A；这时用户A由于用户B已经锁住表B，它必须等待用户B释放表B才能继续，同样用户B要等用户A释放表A才能继续，这就死锁就产生了。

|  |  |
| --- | --- |
| TX1 | TX2 |
| SET autocommit = 0;(取消自动提交) |  |
| SELECT \* FROM carmod\_brand t WHERE t.brand\_id = 4 FOR UPDATE; |  |
|  | SET autocommit = 0; |
|  | SELECT \* FROM carmod\_brand t WHERE t.brand\_id = 5 FOR UPDATE; |
| SELECT \* FROM carmod\_brand t WHERE t.brand\_id = 5 FOR UPDATE; |  |
|  | SELECT \* FROM carmod\_brand t WHERE t.brand\_id = 4 FOR UPDATE; |
|  | Lock wait timeout exceeded; try restarting transaction |

解决方案：

这种死锁比较常见，是由于程序的BUG产生的，除了调整的程序的逻辑没有其它的办法。仔细分析程序的逻辑，对于数据库的多表操作时，尽量按照相同的顺序进行处理，尽量避免同时锁定两个资源，如操作A和B两张表时，总是按先A后B的顺序处理， 必须同时锁定两个资源时，要保证在任何时刻都应该按照相同的顺序来锁定资源。

引伸：

1. InnoDB引擎会把写操作(insert/update/delete/select for update)当做事务来处理；
2. 所以单条UPDATE语句是不需要显式开启事务的.
3. MySQL默认会自动提交事务,每一条单独的查询都是一个事务；所以使用 SELECT FOR UPDATE 时应该关闭自动提交后手动开启事务 START TRANSACTION；否则 SELECT FOR UPDATE 的锁在事务自动提交后就释放了.
4. 

### 共享锁升级独占锁

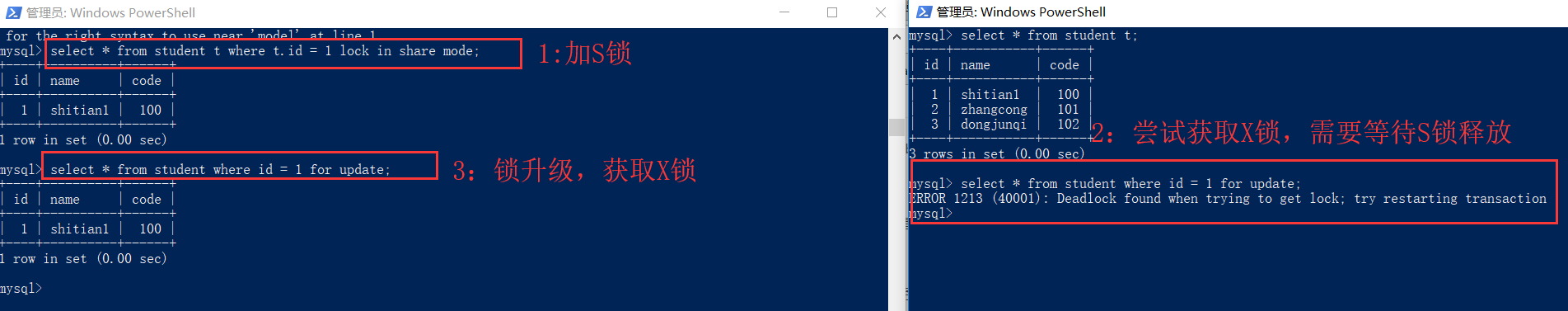
出现原因：

用户A查询一条纪录（获取共享锁S）；用户B修改该条纪录（获取排它锁X，但是需要等待A的共享锁S释放），这时用户A需要修改该条记录，锁的性质由查询的共享锁企图上升到独占锁（尝试获取排它锁X，不会释放S锁），而用户B里的独占锁由于A有共享锁存在所以必须等A释放掉共享锁，而A由于B的独占锁而无法上升的独占锁也就不可能释放共享锁，于是出现了死锁。这种死锁比较隐蔽。

解决方案：

A）使用乐观锁：通过版本控制实现无锁并发（查询时候获取版本号，更新时候比对版本号是否相同），当高并发下比较容易大量的更新不成功。

B）使用悲观锁：查询时候就使用select for update获取排它锁，性能严重下降。



### 9.2.3 mysql更新导致死锁

出现原因： mysql的innodb默认使用的是行锁，支持表锁；行锁锁的是索引，并不是记录；

update table set num = 100 where normal\_index = 1；

第一步：使用普通索引列normal\_index，所以获取了这个普通索引的锁。

第二步：获取主键锁，然后根据主键索引锁定该记录，修改数据。

Update table set normal\_index = 2 where pk = 100；

第一步：根据主键索引修改数据，所以先获取主键索引。

第二步：因为需要更新普通索引列数据，所以还需要获取普通索引。

如果上述的两个sql同时执行，就会导致死锁，互相等待。

解决方案：更新尽量带上主键，保证优先获取主键锁，避免死锁。

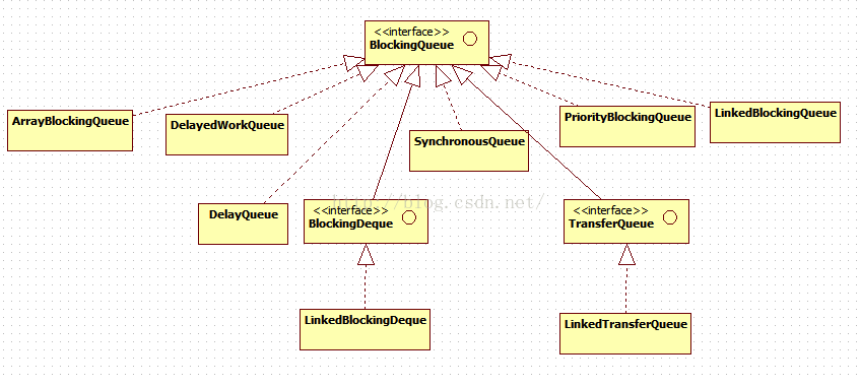
# 阻塞队列

## 8.1概述

阻塞队列与普通队列的区别在于，当**队列是空**的时，从队列中**获取元素的操作将会被阻塞**，或者当**队列是满**时，往队列里**添加元素的操作会被阻塞**。试图从空的阻塞队列中获取元素的线程将会被阻塞，直到其他的线程往空的队列插入新的元素。

阻塞队列的简单实现：BlockingQueueTest

## 8.2阻塞队列的实现（BlockingQueueTest）



### 8.2.1 ArrayBlockingQueue

ArrayBlockingQueue是一个**阻塞式的有界队列**，继承自AbstractBlockingQueue,间接的实现了Queue接口和Collection接口。底层以数组的形式保存数据。常用的操作包括 add ,offer,put，remove,poll,take,peek。

前三者add offer put 是插入的操作。后面四个方法是取出的操作。他们之间的区别和关联：

1. add: 内部实际上调用offer方法，当Queue已经满了时，抛出一个异常。不会阻塞。
2. offer:当Queue已经满了时，返回false。不会阻塞。
3. put:当Queue已经满了时，会进入等待，只要不被中断，就会插入数据到队列中。会阻塞，可以响应中断。

取出方法中 remove和add相互对应。也就是说，调用remove方法时，假如队列为空，则抛出异常。另外的，poll与offer相互对应。take和put相互对应。peek方法比较特殊，前三个取出的方法，都会将元素从Queue的头部弹出，但是peek不会，实际上只是查看队列头的元素。peek方法也不会阻塞。当队列为空时，直接返回Null。

### 8.2.2 LinkedBlockingQueue

LinkedBlockingQueue是一个**阻塞式的无界队列**，继承自AbstractBlockingQueue,间接的实现了Queue接口和Collection接口；底层以链表的形式保存数据。LinkedBlockingQueue构造的时候若没有指定大小，则默认大小为Integer.MAX\_VALUE，当然也可以在构造函数的参数中指定大小。

添加元素的方法有三个：add,put,offer,且这三个元素都是向队列尾部添加元素。

1. add方法在添加元素的时候，若超出了度列的长度会直接抛出异常（AbstructQueue）
2. put方法，若向队尾添加元素的时候发现队列已经满了会发生阻塞一直等待空间，以加入元素，会阻塞，可以响应中断
3. offer方法在添加元素时，如果发现队列已满无法添加的话，会直接返回false

从队列中取出并移除头元素的方法有：poll，remove，take。

1. poll: 若队列为空，返回null
2. remove:若队列为空，抛出NoSuchElementException异常
3. take:若队列为空，发生阻塞，等待有元素

### 8.2.3 PriorityBlockingQueue

PriorityBlockingQueue是一个**基于数组实现的线程安全的无界队列**，继承结构与上面两种完全一样，原理和内部结构跟PriorityQueue基本一样，只是多了个线程安全。javadoc里面提到一句：

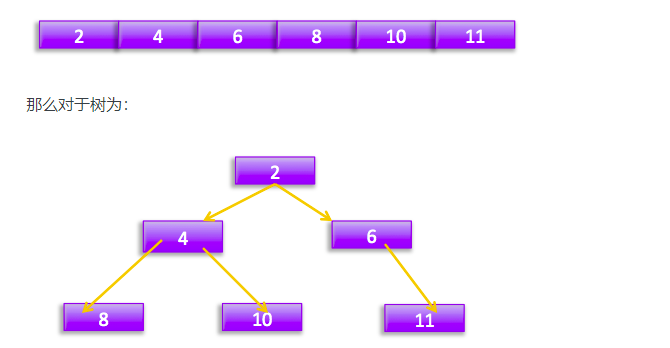
1：理论上是无界的，所以添加元素可能导致out of memory error；

2.不容许添加null；

3.添加的元素使用构造时候传入Comparator排序，要不然就使用元素的自然排序。

**二叉堆，实现内部优先级队列**，底层还是数组实现(private transient Object[] queue;)：

对于**queue[n]的左右孩子节点为：queue[2\*n+1]（左）,queue[2\*(n+1)]（右），每次出队都是queue[0]**



add、put、offer都是最终调用offer()方法，实现逻辑如下:

1. 首先检查添加的元素是否为null，如果为null，则抛空指针异常。否则进行 2
2. 加锁
3. 检查数组是否已满，如果已满，则先进行扩容，否则进行 4
4. 然后将元素存储在数组的末尾即可。(由于优先队列底层借助于二叉堆来实现的，因此在插入之后，要进行一定的调整，使之维持二叉堆的特性。具体为：检查比较器是否为null，如果为null，则调用siftUpComparable方法将元素加入到数组末尾并进行相应的调整使之维持二叉堆的特性。如果比较器不为null。则根据指定的比较器进行调整）
5. 该计数器进行加一操作，由于成功的添加了一个元素队列不再为空，因此发出一个非空信号来唤醒正在等待的消费者
6. 释放锁

数组扩容逻辑tryGrow()：

1. 释放锁(扩容需要时间，如果还占有锁，则别的出队线程必须等待，释放锁增加效率)
2. 根据CAS设置扩容的共享变量，保证只会有一个扩容线程成功
3. oldcap<64则扩容新增oldcap+2,否者扩容50%，并且最大容量为MAX\_ARRAY\_SIZE
4. 设置扩容共享变量为0
5. if (newArray == null) Thread.yield();该逻辑为让后续也进行扩容的线程让出CPU时间，尽量让第一个扩容的线程获取锁
6. 再次获取queue的锁
7. 进行数组拷贝

数组顺序调整逻辑siftUpComparable()：

1. 根据队列末尾加入的元素下标，找到对应的父节点
2. 两者比较
3. 如果子大于父，则两者交换位置，否则循环结束，新增的元素直接加入队尾
4. 循环3，直到子小于父，将子节点设置到对应的位置

出队操作take最终还是调用dequeue():

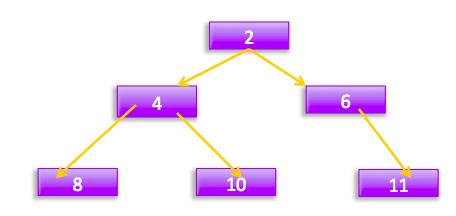
1. 数组为空，返回null
2. 返回数组第一个元素
3. 将数组最后一个元素放入array[0]，并设置数组最后一个元素为null
4. siftDownComparable调整数组的顺序

首先说下思想，其中k一开始为0，x为数组里面最后一个元素，由于第0个元素为树根，被出队时候要被搞掉，所以建堆要从它的左右孩子节点找一个最小的值来当树根，子树根被搞掉后，会找子树的左右孩子最小的元素来代替，直到树节点为止.

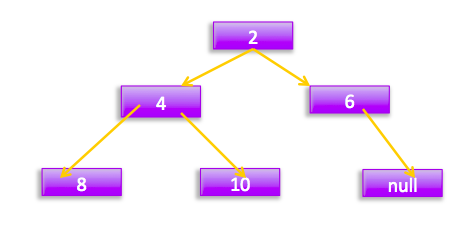
假如当前队列元素：



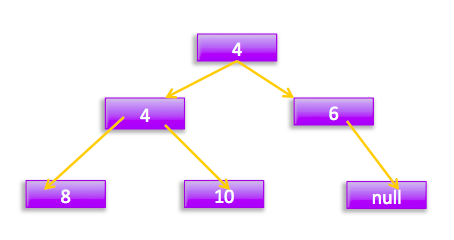
那么对于树为：



这时候如果调用了poll();那么result=2;x=11;现在树为：

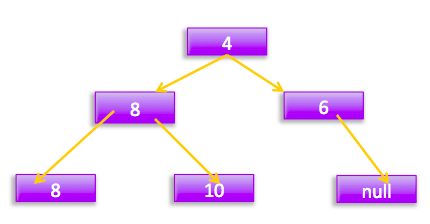


然后看leftChildVal = 4;rightChildVal = 6; 4<6;所以c=4;也就是获取根节点的左右孩子值小的那一个； 然后看11>4也就是key>c；然后把c放入树根，现在树为：

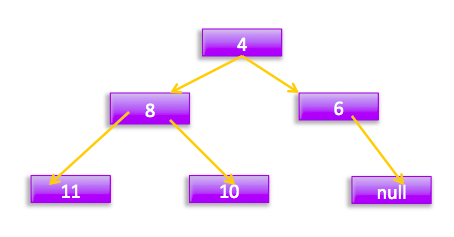


然后看根的左边孩子4为根的子树我们要为这个子树找一个根节点

看leftChildVal = 8;rightChildVal = 10; 8<10;所以c=8;也就是获取根节点的左右孩子值小的那一个； 然后看11>8也就是key>c；然后把c放入树根，现在树为：



这时候k=3;half=3所以退出循环，结果为：



这时候队列为：



### 8.2.4 DelayQueue

DelayQueue是**无界的阻塞队列，其特点是实现队列元素的延迟出队**，通俗点说就是队列元素可以设置延迟时间，时间不到，就待在队列中，很有意思的东西，跟redis设置过期时间一样。队列元素不容许添加null元素。DelayQueue可以用来实现调度（ScheduledThreadPoolExecutor使用DelayedWorkQueue<SchedualedFutureTask>）的定时任务或者缓存的过期（到期了就出队，这样就查询不到了）。

加入DelayQueue的对象，必须实现Delayed接口，实现getDelay（返回当前对象还需等的时间）与compareTo（用于优先队列中排序使用）

成员变量：

1. transient ReentrantLock lock；重入锁，出队入队都需要加锁
   1. transient修饰的变量不会被序列化，例如为了安全，客户的密码不想在网络上传输，可以增加该修饰，这样对端反序列化出来的对象中，密码为空
2. PriorityQueue<E> q = new PriorityQueue<E>()；数据的存储使用优先级队列存储，保证等待时间小的元素在队头
3. Thread leader；leader-follower模式，即多线程下并发获取节点时候，保证只有一个线程获得锁成为leader，并等待元素到期，别的线程直接无限等待，直到leader唤醒
4. Condition available = lock.newCondition()；使用条件锁唤醒其他等待的线程，然后来获取队列元素

入队方法add、offer、put三个加入队列方法，其中add和put都是直接调用offer方法（添加元素会排序，算法同8.3.2）

1. 

出队方法poll()、poll(..)、take()、peek()这四个方法都可以实现从队列头获取一个对象

1. Peek非阻塞，直接返回队列的第一个节点，但是不会删除。
2. Poll非阻塞，获取第一个节点，然后判断是否到期，到期则返回，然后删除第一个节点（删除后会重新排序，算法同8.2.3），如果没有到期，则返回null
3. Poll（..）原理同poll，只是支持等待传入的时间到期后，才返回，期间一直轮询判断队列的第一个节点是否到期可以返回
4. 

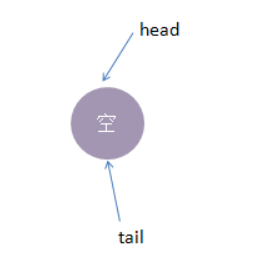
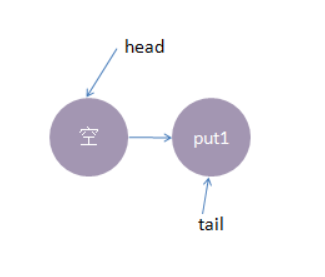
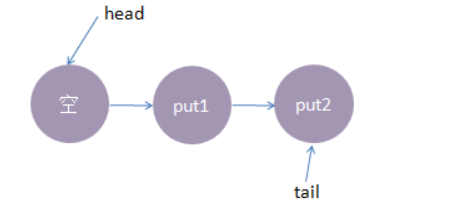
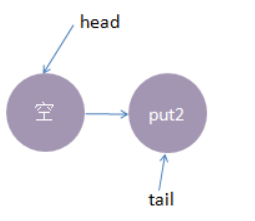
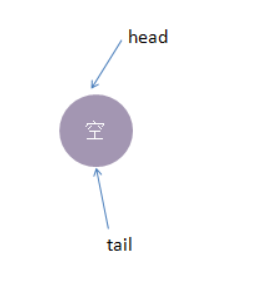
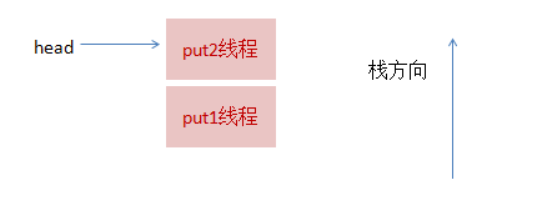
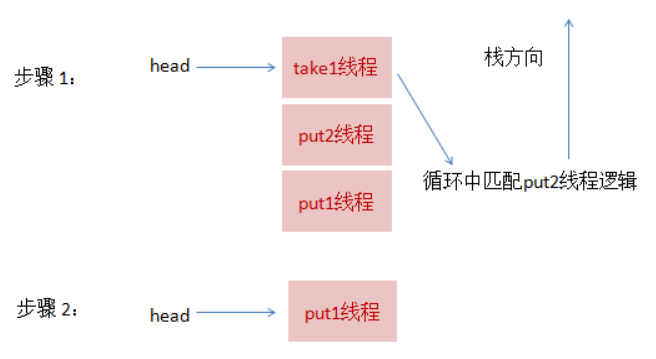
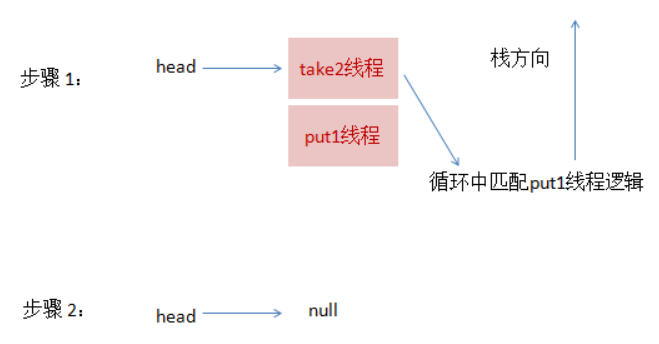
备注：

1. first = null; // don't retain ref while waiting（等待期间不保持first的引用，避免内存泄漏）
   1. 线程A进来获取first,然后进入 else 的else（20行，释放锁并等待） ,设置了leader为当前线程A
   2. 线程B进来获取first,进入else（15行，自己不是leader，进入无限期等待）的阻塞操作,然后无限期等待
   3. 如果线程A阻塞完毕,获取对象成功,出队,这个对象理应被GC回收,但是他还被线程B持有着,GC链可达,所以不能回收这个first
   4. 假设还有线程C 、D、E.. 持有对象1引用,那么无限期的不能回收该对象1引用了,那么就会造成内存泄露
2. Leader-follower模式
   1. A,B两个线程同时take元素
   2. A先获取锁，然后A设置自己为leader，然后释放锁等待delay时间
   3. B再获得锁，发现自己不是leader，直接无限等待
   4. 如果没有leader模式
      1. B也会等待第一个元素的delay时间
      2. 到了delay时间后，AB都醒了，然后同时去竞争锁
      3. A成功后，获取第一个元素并删除
      4. B获得锁，然后获取第一个元素，发现元素的delay时间还没有到，又要等待
      5. 所以leader模式可以避免不必要的锁竞争

### 8.2.5 SynchronousQueue(SynchronousQueueDemo)

SynchronousQueue 也是一个队列，但它的特别之处在于它**内部没有容器**，一个生产线程，当它生产产品（即put的时候），如果当前没有人想要消费产品(即当前没有线程执行take)，此生产线程必须阻塞，等待一个消费线程调用take操作，take操作将会唤醒该生产线程，同时消费线程会获取生产线程的产品（即数据传递），这样的一个过程称为一次配对过程(当然也可以先take后put,原理是一样的)。

所有操作都是直接使用CAS算法保证线程安全，没有使用AQS队列。

1. 公平模式：使用TransferQueue（队列，FIFO）
   1. 初始状态
   2. 线程put1执行 put(1)操作
   3. 线程put2执行了put(2)操作，跟前面一样，put2线程入队列
   4. 这时候，来了一个线程take1，执行了 take操作，由于tail是put2，所以take1与put2配对成功，但是由于公平模式，所以真正唤醒的是put1线程（队尾匹配，队头出）
   5. 最后，再来一个线程take2，执行take操作
2. 非公平模式：使用TransferStack（栈，LIFO）
   1. 线程put1执行 put(1)操作
   2. 线程put2再次执行了put(2)操作，跟前面一样，put2线程入栈
   3. 这时候，来了一个线程take1，执行了take操作，这时候发现栈顶为put2线程，匹配成功，但是实现会先把take1线程入栈，然后take1线程循环执行匹配put2线程逻辑，一旦发现没有并发冲突，就会把栈顶指针直接指向 put1线程
   4. 最后，再来一个线程take2，执行take操作，这跟步骤3的逻辑基本是一致的，take2线程入栈，然后在循环中匹配put1线程，最终全部匹配完毕，栈变为空

### 8.2.6 LinkedTransferQueue

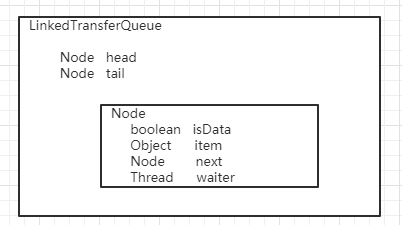
TransferQueue是一个继承了BlockingQueue的接口，并且增加若干新的方法。LinkedTransferQueue是TransferQueue接口的实现类，其定义为一个无界的队列，具有先进先出(FIFO)的特性，同时也是个阻塞队列，但是与传统阻塞队列不同的是，传统阻塞队列需要队列满后，再入队会阻塞，但是LinkedTransferQueue支持队列不满也能阻塞，使用CAS保证线程安全。

LinkedTransferQueue实现了一个重要的接口TransferQueue，该接口含有下面几个重要方法：

1. tryTransfer(E)：将元素立刻给消费者。准确的说就是立刻给一个等待接收元素的线程，如果没有消费者就会返回false，而不将元素放入队列（非阻塞）。
2. transfer(E)：将元素给消费者，如果没有消费者就会等待（阻塞）。
3. tryTransfer(E,long,TimeUnit)：将元素立刻给消费者，如果没有就等待指定时间。给失败返回false（定时阻塞）。
4. hasWaitingConsumer()：返回当前是否有消费者在等待元素。
5. getWaitingConsumerCount()：返回等待元素的消费者个数。

**备注：LinkedTransferQueue采用的一种预占模式。意思就是消费者线程取元素时，如果队列为空，那就生成一个节点（节点元素为null）入队，然后消费者线程阻塞，后面生产者线程入队时发现有一个元素为null的节点，生产者线程就不入队了（没有消费者线程，生产者入队，等待消费者线程匹配），直接就将元素填充到该节点，唤醒该节点上阻塞的线程，被唤醒的消费者线程拿货走人。这就是预占的意思：有就拿货走人，没有就占个位置等着，等到或超时。**

LinkedTransferQueue简要数据结构



Node节点数据结构（基于单链表实现）：



1. isData：表示当前是存数据（生产者）还是取数据（消费者）。
2. item：存放数据地方；如果isData=false（消费者）；item=null；
3. next：指向下一个node节点
4. waiter：存放阻塞的线程

处理逻辑：

private static final int NOW = 0; // for untimed poll, tryTransfer

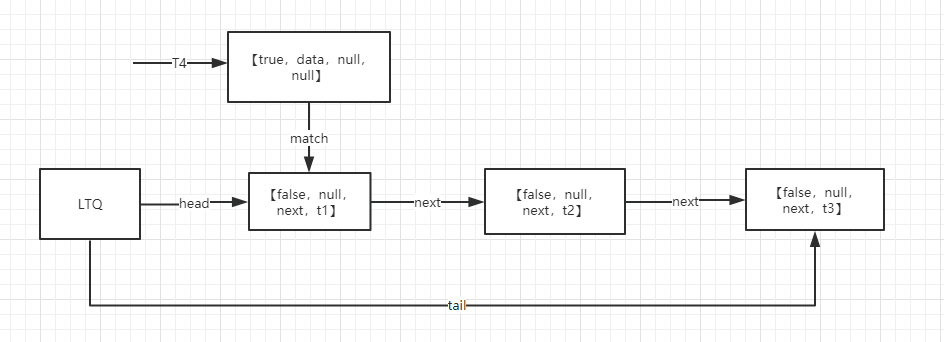
private static final int ASYNC = 1; // for offer, put, add

private static final int SYNC = 2; // for transfer, take

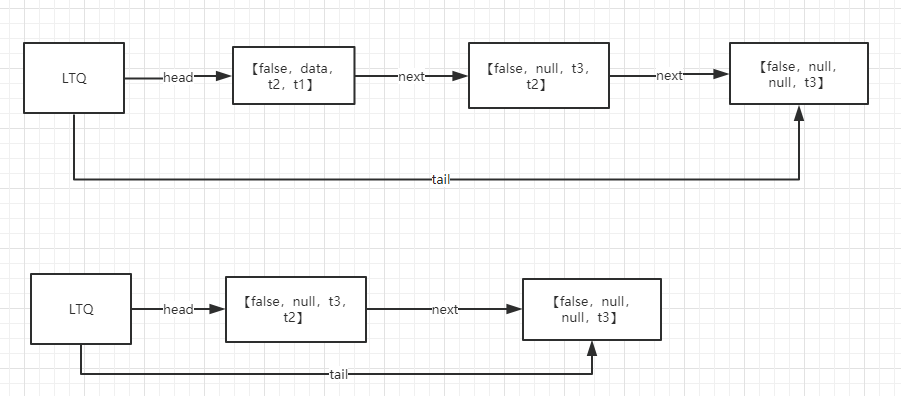
private static final int TIMED = 3; // for timed poll, tryTransfer(..)

寻找以及操作匹配的节点（LinkedTransferQueue中还存放一个head和tail分别指向头和尾节点）

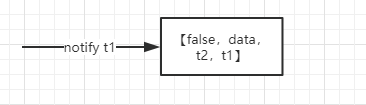
1. 生产者存放数据：（isData=true；item！=null；）从head开始遍历链表匹配
   1. 匹配成功（存在一个node：isData=false&&item=null），然后把生产者的item（即数据）赋给该(CAS赋值)node（此时node： isData=false&&item！=null，表示匹配成功），唤醒消费者线程，然后head往后移一位。
   2. 遍历链表后，没有匹配到消费者，now（非阻塞）的就直接返回，其他的3种先入队，然后ASYNC（非阻塞）入队后返回，SYNC和TIMED阻塞等待匹配。
2. 消费者消费数据：（isData=false；item=null；）从head开始遍历链表匹配
   1. 匹配成功，将自身的item赋值给对应的生产者节点（此时node： isData=true&&item=null，表示匹配成功），唤醒生产者线程，然后head往后移。
   2. 匹配失败，同上述。
3. 综上
   1. 匹配：主要是判断匹配条件，节点本身还未匹配，且isData类型和待匹配的不一样就行，匹配后就是cas设置tem，唤醒匹配节点waiter（存放当前线程），然后返回，head后移；
   2. 不匹配：如果没找到，那就根据不同方法入参how处理了，now的就直接返回，其他的3种先入队，ASYNC入队后返回，SYNC和TIMED阻塞等待匹配。
   3. 备注：尽量避免使用SYNC，因为是无界队列，当大量线程都阻塞时，会占用大量资源。



生产者线程匹配成功



生产者先将数据给head的data，然后head指针后移



唤醒匹配的消费者线程，消费data数据

# 饥饿与公平

1. 导致饥饿的原因
   1. 高优先级线程吞噬所有的低优先级线程的CPU时间
      1. 每个线程都有优先级，可以人工设置，优先级（1-10）越高，获取CPU时间越多，优先级表示的行为状态依赖运行的平台，尽量不要人工修改优先级。
   2. 线程永远阻塞在同步代码块外
      1. Synchronized为非公平锁，所以可能存在某一线程永远获取不到锁
   3. 线程等待被唤醒
      1. Notify是随机唤醒一个线程，所以也可能存在某一线程永远唤醒不过来
2. 使用公平锁，解决线程饥饿
   1. 使用ReentrantLock（true），默认是非公平锁，传true表示使用公平锁
3. 公平锁：所有线程按照先后顺序进入等待队列，新来的线程永远加入队尾
   1. 公平锁性能一般会比非公平锁低
4. 非公平锁：线程释放锁后，唤醒队列中的第一个线程，然后来尝试获取锁；此时来了一个新线程，非公平锁允许该新线程也来获取锁，成功则执行，失败则加入队尾。
   1. 非公平锁快的原因：恢复一个阻塞的线程到该线程变为可运行，之间存在比较多的延迟，例如加载上次执行的中间信息等。
      1. A释放锁，然后唤醒B，B处于阻塞到就绪的恢复状态；C新到（C耗时很小），然后获取锁成功，并且执行完，然后释放锁；此时B恢复完成，然后获取锁成功，然后运行；这就是一个双赢的局面。
      2. 如果线程占有锁的时间很长，或者在获取锁的耗时也很长，则使用公平锁（这种情况下，插队带来的好处基本不会出现，因为获取锁的时间很长，唤醒的线程已经彻底恢复了）。

# 常用的实现多线程方式

1. 创建方式
   1. 继承Thread对象
   2. 实现Runnable接口
   3. 实现Callable接口（CallableTest）
2. 不同之处
   1. 继承Thread不适合资源共享，每个new出来的Thread都是新的。
   2. Callable规定的方法是call()，而Runnable规定的方法是run()
   3. 运行Callable任务可拿到一个Future对象，Future表示异步计算的结果。它提供了检查计算是否完成的方法,以等待计算的完成,并检索计算的结果.通过Future对象可了解任务执行情况,也可取消任务的执行,还可阻塞式获取任务执行的结果
      1. Future的cancel方法，入参为boolean，如果为true，则表示中断线程；为false，如果该线程已经在运行则不处理，让线程跑完，线程在等待队列，则后续线程池不处理该线程（**简单来说，传入false参数只能取消还没有开始的任务，若任务已经开始了，就任由其运行下去**。）。
   4. call()方法可抛出异常，而run()方法是不能抛出异常的，只能try/catch

# 常用方法解析

## 线程等待/唤醒（NotifyDeadLockOne）

1. 涉及方法
   1. Wait：让当前线程进入阻塞状态，同时释放锁
   2. Wait(long time)：当前线程进入阻塞状态，等待别的线程唤醒，或者超过规定毫秒时间自动唤醒；或者别的线程中断该线程，抛出interruptException
   3. Wait（long time，int nanos）：同上一个方法，只是后面增加了纳秒，nanos表示纳秒数，能指定线程等待规定的毫秒+纳秒数（jdk8发现nanos大于0，直接加1毫秒）
   4. Notify：随机唤醒一个等待线程
   5. notifyAll：唤醒所有等待线程

## 线程让步(YieldTest)

1. 涉及方法
   1. Yield：它能让当前线程由“运行状态”进入到“就绪状态”，从而让其它具有相同优先级的等待线程获取执行权；但是，并不能保证在当前线程调用yield()之后，其它具有相同优先级的线程就一定能获得执行权；也有可能是当前线程又进入到“运行状态”继续运行！**不会释放锁**。

## 线程休眠(SleepTest)

1. 涉及方法
   1. Sleep：线程从运行状态进入阻塞状态，休眠时间过了后，自动唤醒，**不会释放锁**。

## 线程加入(JoinTest)

1. 涉及方法
   1. Join：让“主线程”等待“子线程”结束之后才能继续运行。Join后，主线程会进入阻塞状态，直到子线程结束，**join会释放锁**。

# 如何中断线程(InterruptTest, FutureCancel)

1. 通过InterruptException中断线程。
   1. 线程处于阻塞状态（**sleep，wait，join**），通过调用线程的interrupt方法来中断线程，此时线程会抛出异常，从而终止线程
2. 通过线程内部自己判断中断标识来判断是否终止线程。
3. 通过额外添加标记实现线程中断。
4. 中断线程池中的线程。
   1. 通过返回的Future对象，调用cancel中断线程。
   2. Cancel只能中断阻塞状态的线程，如果是IO阻塞（关闭IO抛异常中断），或者sync同步代码块（使用中断标识判断），则不能中断。

# 并发编程的三个概念

## 原子：一个操作要么全部成功，要么全部失败。

## 可见：多线程模式下，A线程对共享对象做了修改，其他线程要能立刻看到

## 有序：程序执行的顺序按照代码的顺序（JVM为了提高效率会指令重排序，JVM保证重排序后，结果不会有变化）

**//线程1:**

context = loadContext(); //语句1

inited = true; //语句2

**//线程2:**

while(!inited ){

sleep()

}

doSomethingwithconfig(context);

说明：上述情况多线程下可能会有问题

1. 线程1先执行，然后由于指令重排序，语句2先执行
2. 线程2执行while循环，发现不满足条件，跳出循环获取context对象操作
3. 线程1此时还没有来得及初始化context
4. 线程2会报空指针

# Volatile关键字

## 保证可见性

通过内存屏障实现，对于volatile修饰的关键词做如下两步操作：

1. 线程写入操作，会在写入后，加入写屏障，强制数据刷入主存。
2. 线程读操作时候，会在读之前，加入读屏障，强制从主存读数据。

## 保证部分有序性

**//线程1:**

**Volatile context = loadContext(); //语句1**

**inited = true; //语句2**

还是以上面的线程1为例，给context加上volatile修饰，jvm在指令重排序的时候，保证context上面的指令与context下面的指令不能重排序，即上面的指令不能到下面，下面的指令不能到上面，上面的指令可以重排序，下面的指令也能重排序；

# ThreadLocal

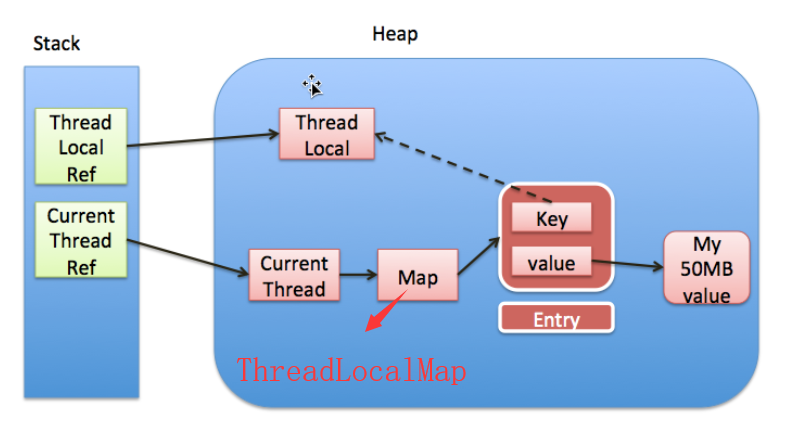
## 概述

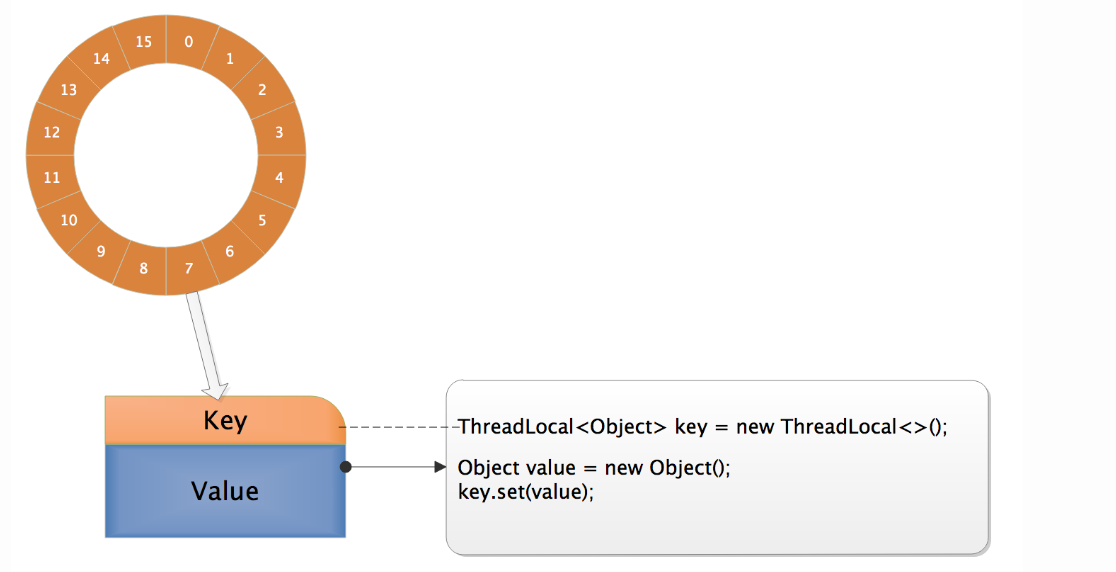
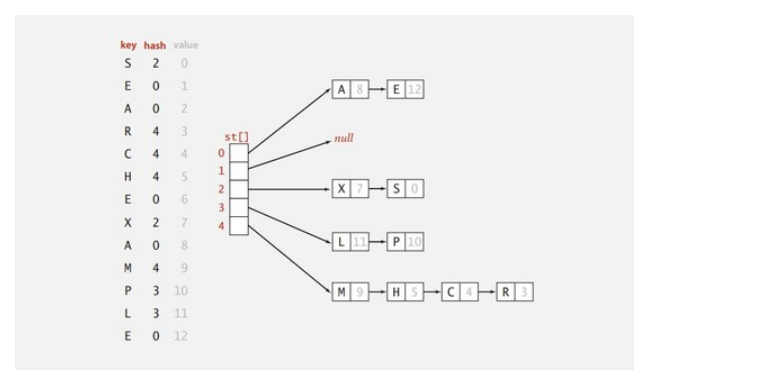
1. 什么是ThreadLocal

ThreadLocal类顾名思义可以理解为线程本地变量。也就是说如果定义了一个ThreadLocal，每个线程往这个ThreadLocal中读写是线程隔离，互相之间不会影响的。**它提供了一种将可变数据通过每个线程有自己的独立副本从而实现线程封闭的机制**。

1. 它大致的实现思路是怎样的

Thread类有一个类型为ThreadLocal.ThreadLocalMap的实例变量threadLocals，也就是说每个线程有一个自己的ThreadLocalMap。ThreadLocalMap有自己的独立实现，可以简单地将它的key视作ThreadLocal，value为代码中放入的值（实际上key并不是ThreadLocal本身，而是它的一个弱引用）。每个线程在往某个ThreadLocal里塞值的时候，都会往自己的ThreadLocalMap里存，读也是以某个ThreadLocal作为引用，在自己的map里找对应的key，从而实现了线程隔离。



1. ThreadLocalMap内部存储：使用线性探测解决hash冲突，所以可以简单理解为entry数组是一个环。
   1. 
2. 引申解决hash冲突的方法
   1. 拉链法（HashMap使用）：实现比较简单，将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表的表头
      1. 
   2. 线性探测法：直接使用数组来存储数据。可以想象成一个停车问题。若当前车位已经有车，则你就继续往前开，直到找到下一个为空的车位；若到了数组末尾可循环到数组起始位置。
      1. 需要注意扩容，否则数组满了，会导致死循环
      2. 

## 导致内存泄漏的情况

ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用来引用它，那么系统 GC 的时候，这个ThreadLocal势必会被回收，这样一来，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry的value，如果当前线程再迟迟不结束的话，**这些key为null的Entry的value就会一直存在一条强引用链：Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value永远无法回收，造成内存泄漏**。

因为ThreadLocalMap与Thread生命周期相关，如果线程一直不销毁（例如线程池），map中的数据就一直无法回收。

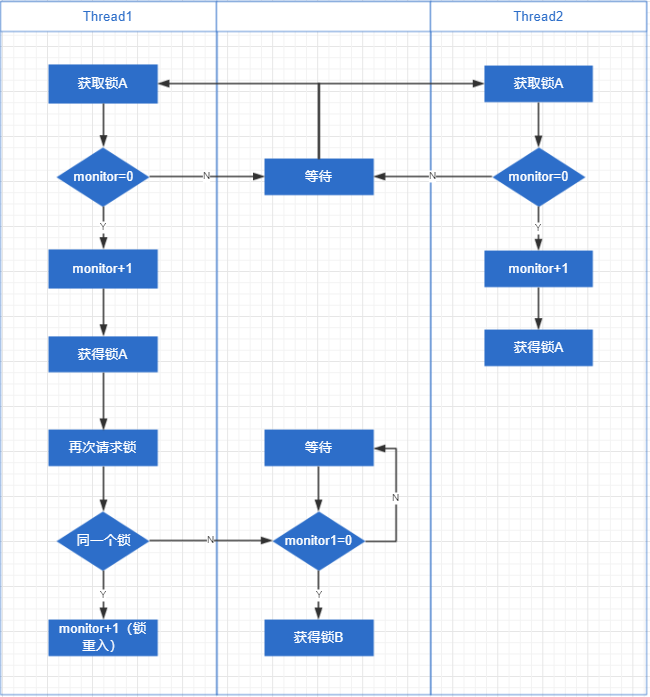
其实，ThreadLocalMap的设计中已经考虑到这种情况，也加上了一些防护措施：在ThreadLocal的get(),set(),remove()的时候都会调用expungeStaleEntry函数清除线程ThreadLocalMap里所有key为null的value。

综上，使用ThreadLocal，推荐显示的调用remove方法。

# 锁

## 内置锁（Sync）

1. syncrhoized加锁的同步代码块在字节码引擎中执行时，其实是通过锁对象的monitor的取用与释放来实现的。由上面我们知道Monitor是内置于任何一个对象中的，一个对象只有一个monitor，syncrhoized利用monitor来实现加锁解锁，故syncrhoized又叫做内置锁。
   1. 线程进入对象monitor，如果monitor为0（没有被占用），则线程将monitor计数加一，该线程为该对象的拥有者。【线程获取对象锁】
   2. 线程再次访问了对象的同步方法，则monitor计数再次加一，这个是锁重入机制。【锁重入机制】
   3. 其他线程发现对象monitor不为0，则等待，直到monitor为0，再尝试获取锁。【线程等待锁释放】
   4. 释放锁后，monitor计数减一，当monitor计数为0时，则表示该线程释放锁，别的线程能获取该对象了。【释放锁】
2. 多线程下，A线程获取锁，B线程必须一直等待，直到A结束，期间B线程不能中断
3. 当抛出异常时，由于在jvm基础上实现，所以会自动释放锁



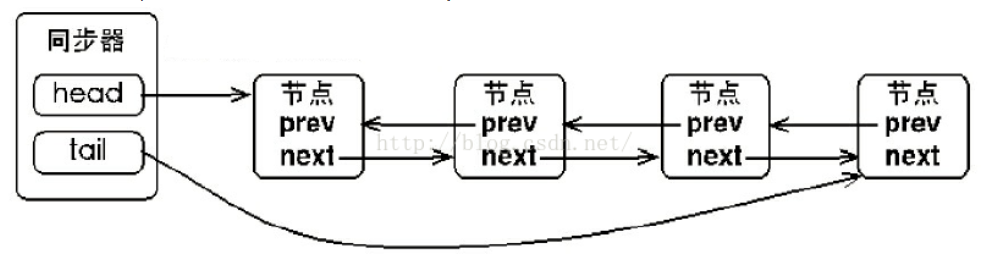
## ReentrantLock

### AQS队列

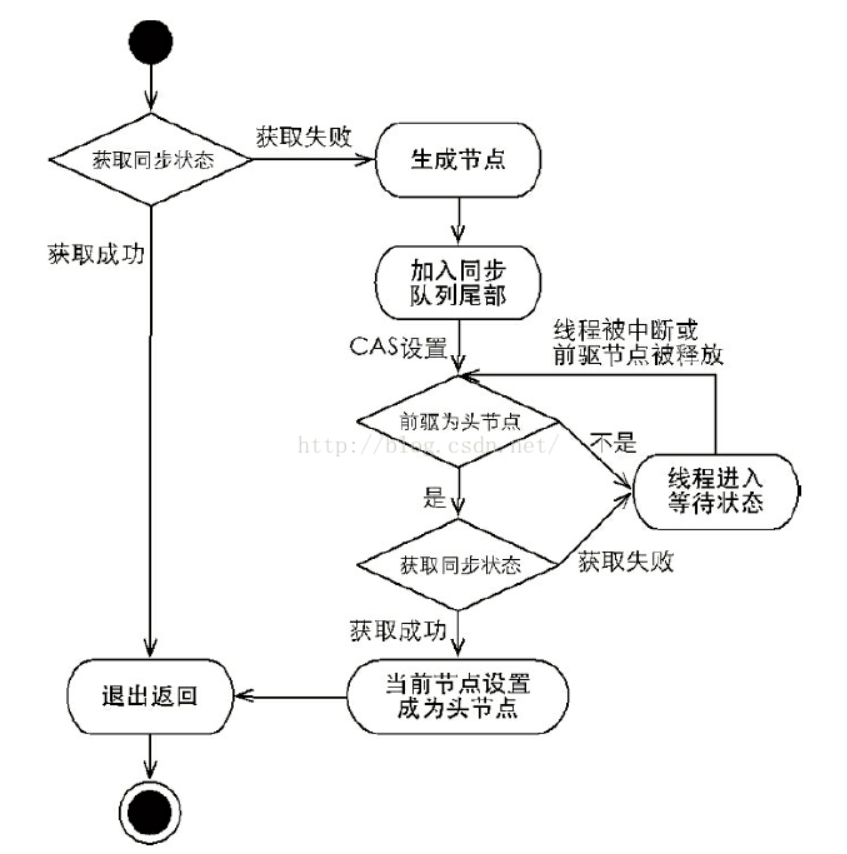
队列同步器AbstractQueuedSynchronizer（以下简称同步器），是用来构建锁或者其他同步组件的基础框架，它使用了一个int成员变量表示同步状态，通过内置的FIFO队列来完成资源获取线程的排队工作。

组成部分：

1. 节点（Node）用来保存"获取同步状态失败的线程"引用、等待状态以及前驱和后继节点；
2. Head头节点，指向队头
3. Tail尾节点，指向队尾
4. State：同步状态，通过CAS成功设置该值的线程，表示获取同步状态成功，在锁里面就表现为获取锁成功

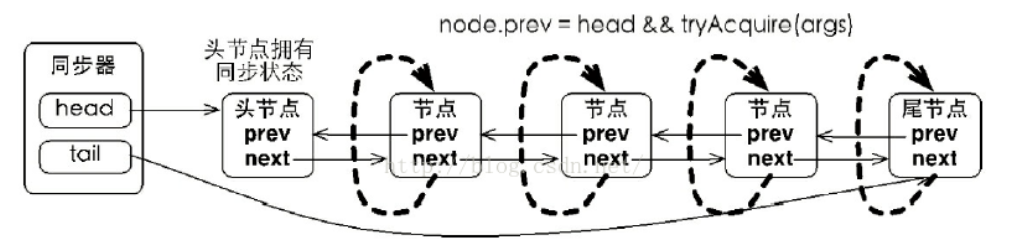


获取同步状态逻辑如图：



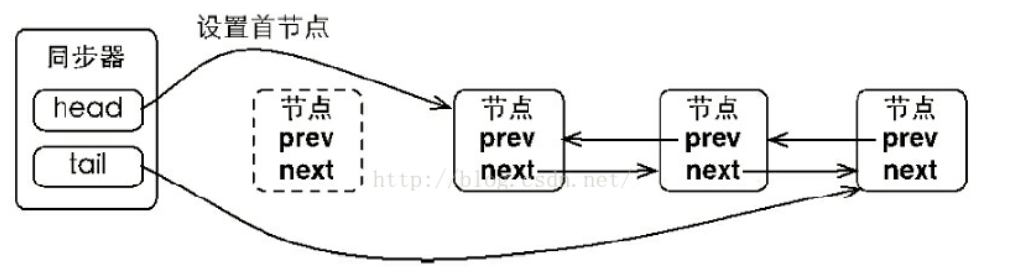
当线程加入队列后，会进入自旋状态（死循环判断条件），**如果前驱节点是head节点，并且取得了同步状态**，则退出自旋。

自旋的好处是线程不需要睡眠和唤醒，减小了系统调用的开销；而且也符合FIFO。



同步队列遵循FIFO，首节点是获取同步状态成功的节点，首节点的线程在释放同步状态时，将会唤醒后续节点，而后续节点将会在获取同步状态成功时将自己设置为首节点。

设置首节点的逻辑由获取了同步状态的线程设置（**只有一个线程能获取同步状态），因此不需要进行CAS操作**，直接设置下个节点为首节点，同时断开next引用。



### 16.2.2 公平锁

1. 获取锁
   1. state=0
      1. 当前线程是否是队列中的head，如果是，则使用cas设置state加一
      2. 设置当前线程独占锁
2. state！=0
   1. 当前线程是否独占锁，如果是，则state再加一（锁重入）
   2. 否则获取锁失败
   3. 将线程加入队列尾部并阻塞
3. 释放锁（同下）

### 非公平锁

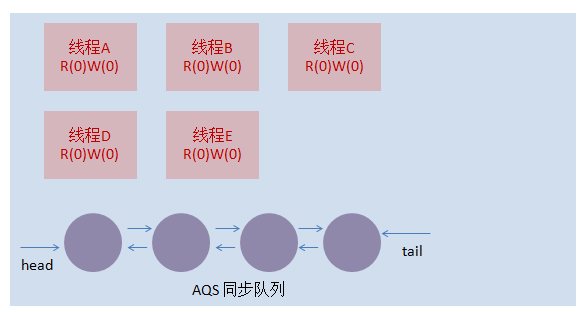
1. 成功获取锁
   1. state通过CAS设置为1
   2. 锁的owner设置为当前线程
2. 已经有线程获取了锁
   1. tryAcquire再次获取一次锁
      1. 获取state是否等于0，如果是则获取锁成功（有可能上个线程执行比较快，一会就释放了锁）
      2. state不等0，则判断当前线程是否是锁的拥有者，如果是，state加一（锁重入），成功获取锁
   2. 获取锁失败，则将该线程加入AQS队列
      1. 如果队列中已经有线程排队，则通过cas将当前线程加入到tail节点后面
      2. 如果没有，则创建一个节点，当前线程为head，也为tail
      3. 如果当前线程是head节点，再次尝试获取锁，成功返回，不成功则将当前线程设置为中断
   3. 释放锁
      1. state减一，如果为0则释放成功，不为0，则还需等下次释放
      2. 释放成功后，需要将当前锁的拥有者设置为null
      3. 唤醒队列的head节点，然后尝试获取锁，成功则删除head节点

## 16.3 读写锁

ReentrantReadWriteLock的锁策略有两种，分为公平策略和非公平策略，两者有些小区别，为便于理解，本小节将以示例的形式来说明多线程下，使用公平策略的读写锁是如何处理的。

**公平读写锁**

一共会出场几个线程，还有用于实现读写机制的AQS同步器队列。每个线程中的 R(0)W(0)表示当前线程占用了多少读写锁。



接下来，我们一步步来看在公平策略下多线程并发的读写机制是怎样的。

1. 程A请求一个读锁，此时无人竞争锁，A获取读锁1，即线程A重入次数为1，如下所示：



1. 程B请求一个读锁，由于AQS中没有等待节点，当前处于读锁占有状态(线程A占有1个读锁)，所以B成功获取读锁，如下所示：



1. 时候，线程C请求一个写锁，由于当前其他两个线程拥有读锁，写锁获取失败，线程C入队列，如下所示：



AQS初始化会创建一个空的头节点，C入队列，然后会休眠，等待其他线程释放锁唤醒。

1. 程D也来了，线程D想获取一个读锁，虽然当于处于读锁占有阶段，但是目前D不占有任何数量的读锁，而且同步器队列中已经有等待节点，这时候，由于公平策略，D不得已，一个字，等，如下图所示：



1. 时候，线程A执行完了，释放了读锁，由于B仍然占有读锁，所以释放后读锁仍然没有完全释放，写锁仍然没有机会执行，如下图所示：



1. 次，B也执行完了，执行完后，读锁全部释放，这时候会唤醒排在同步器队头的节点C，C成功获取一个写锁，如下图所示：



1. 旦任何一个线程获取了写锁，除了该线程自己，其它线程都将无法获取读锁和写锁，这时候，线程C再次请求一个读锁，这是允许的，但反过来如果一个线程先获取了读锁，再获取写法则是不行的。这时候的状态如下图所示：



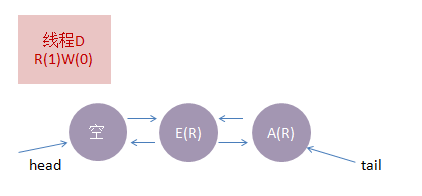
1. 时候假设线程E也来了，E想获取读锁，由于当前处于写锁状态，直接入队，如下所示：



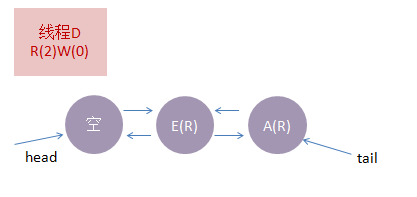
1. 会C终于把活干完了，把读锁和写锁都给释放了，然后线程D被唤醒，获取了读锁，如下图所示：



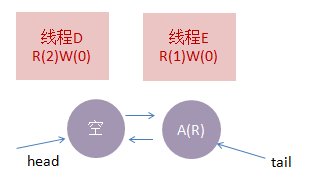
1. 时候，如果再来一个线程，比如A，也想获取读锁，由于节点中还有线程E在等待，而且当前线程A没有获取任何读锁，不是重入状态，所以只能置入队尾，如下图所示：



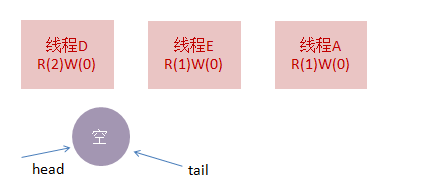
1. 时候，如果D再次调用了一次获取读锁，由于D属于可重入状态，所以直接把读锁+1即可，如下图所示：



1. 于D获取的是读锁，同步队列中的E等待的也是读锁，所以E会被唤醒，获取读锁继续执行，如下图所示：



1. 样的，由于线程A获取的是读锁，在E执行后，会唤醒线程A，A也可以获得读锁，并继续执行，如下图所示：



1. 后大家各自执行，悄然退场。

**非公平读写锁**

接下来我们再来看一下非公平策略读写锁机制又是如何的，为了更好的对比，我们沿用公平锁的流程。

由于获取读锁的逻辑比较复杂，我们在这里先简单进行归纳：

a. 如果当前全局处于无锁状态，则当前线程获取读锁

b. 如果当前全局处于读锁状态，且队列中没有等待线程，则当前线程获取读锁

c. 如果当前全局处于写锁占用状态（并且不是当前线程占有），则当前线程入队尾

d. 如果当前全局处于读锁状态，且等待队列中第一个等待线程想获取写锁，那么当前线程能够获取到读锁的条件为：当前线程获取了写锁，还未释放；当前线程获取了读锁，这一次只是重入读锁而已；其它情况当前线程入队尾。之所以这样处理一方面是为了效率，一方面是为了避免想获取写锁的线程饥饿，老是得不到执行的机会

e. 如果当前全局处于读锁状态，且等待队列中第一个等待线程不是写锁，则当前线程可以抢占读锁

获取写锁相对就比较简单了，规则如下：

h. 如果当前处于无锁状态，则当前线程获取写锁

i. 如果当前全局处于读锁状态，当前线程入队尾

j. 如果当前全局处于写锁状态，除非是重入获取写锁，否则入队尾

接下来我们看一遍流程：

1.线程A请求一个读锁，全局处于无锁状态，根据规则a，线程A获取了锁，如下图所示：



2.线程B请求一个读锁，根据规则b，线程B可以获取到读锁



3.这时候，线程C请求一个写锁，由于当前其他两个线程拥有读锁，写锁获取失败，线程C入队列(根据规则i)，如下所示：



AQS初始化会创建一个空的头节点，C入队列，然后会休眠，等待其他线程释放锁唤醒。

4.线程D也来了，线程D想获取一个读锁，根据读锁规则d，队列中第一个等待线程C请求的是写锁，为避免写锁迟迟获取不到，并且线程D不是重入获取读锁，所以线程D也入队，如下图所示：



5.这时候，线程A执行完了，释放了读锁，由于B仍然占有读锁，所以释放后读锁仍然没有完全释放，写锁仍然没有机会执行，如下图所示：



6.这次，B也执行完了，执行完后，读锁全部释放，这时候会唤醒排在同步器队头的节点C，C成功获取一个写锁，如下图所示：



7.一旦任何一个线程获取了写锁，除了该线程自己，其它线程都将无法获取读锁和写锁，这时候，线程C再次请求一个读锁，这是允许的，但反过来如果一个线程先获取了读锁，再获取写锁则是不行的。这时候的状态如下图所示：



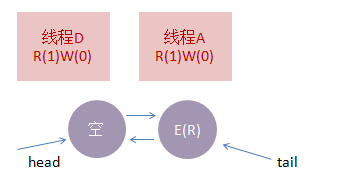
8.这时候假设线程E也来了，E想获取读锁，由于当前处于写锁状态，直接入队，如下所示：



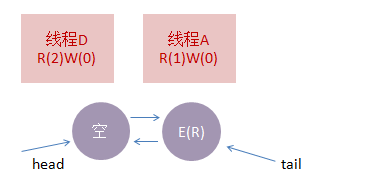
9.这会C终于把活干完了，把读锁和写锁都给释放了，然后线程D被唤醒，获取了读锁，如下图所示：



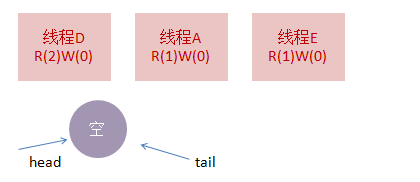
10.这时候，如果再来一个线程，比如A，也想获取读锁，虽然等待队列中，E线程刚好还没被唤醒，但A线程是可以抢占读锁的(这里假设抢占到了)，这个跟公平锁有明显的区别，如下图所示：



11.这时候，如果D再次调用了一次获取读锁，由于D属于可重入状态，所以直接把读锁+1即可，如下图所示：



12.由于当前状态下处于读锁状态，前面的线程D其实醒来后，是会同时唤醒线程E的，所以线程E也醒过来继续干活了，如下图所示：



13.同步队列中没有等待线程了，各个线程执行完后，一切相安无事了。

## 16.4 条件锁

条件锁的原理是基于AQS的队列中，另外增加了一个Condition队列，当

Condition队列的线程被唤醒后，需要进入AQS队列等待获取锁资源执行。

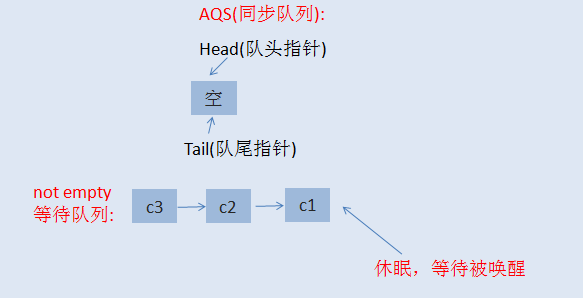
使用公平策略模式演示流程，使用到3个生产线程，3个消费线程，分别表示 p1、p2、p3和c1、c2、c3。

Condition的内部实现是使用节点链来实现的，每个条件实例对应一个节点链，我们有notEmpty 和 notFull 两个条件实例，所以会有两个等待节点链。

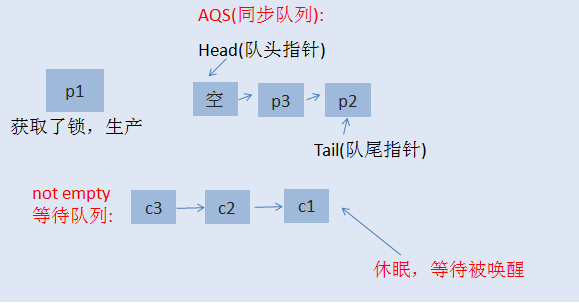
1、线程c3执行，然后发现没有产品可以消费，执行 notEmpty.await，进入等待队列中等候。



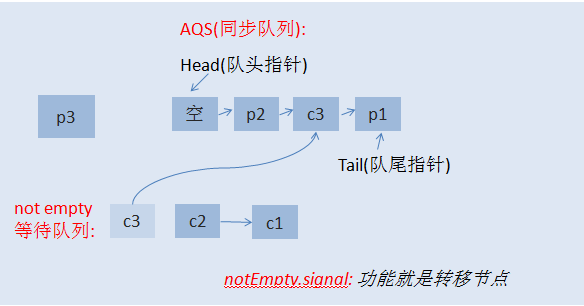
2、线程c2和线程c1执行，然后发现没有产品可以消费，执行 notEmpty.await，进入等待队列中等候。



3、 线程 p1 启动，得到了锁，p1开始生产产品，这时候p3抢在p2之前，执行了lock操作，结果p2和p3都处于等待状态，入同步队列等待，只是P3在P2前

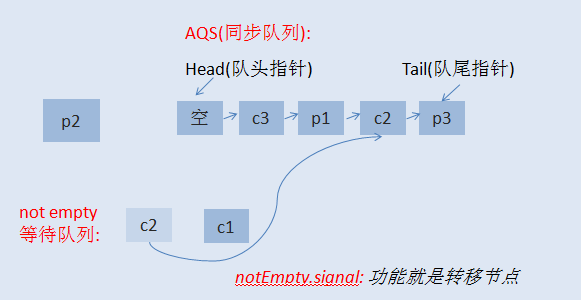


4、这会，p1生产完毕，通知 not empty等待队列，可以唤醒一个等待线程节点了，然后释放了锁，释放锁会导致p3被唤醒，然后p1进入下一个循环，进入同步队列。

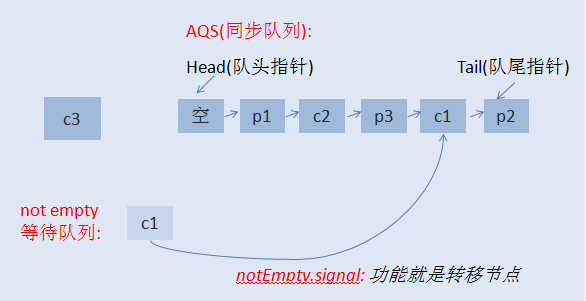


事情开始变得有趣了，p1执行一次生产后，执行了 notEmpty.signal，其效果就是把 not empty等待列表中的头节点，即c3节点移到同步等待列队中，重新参与抢占锁。

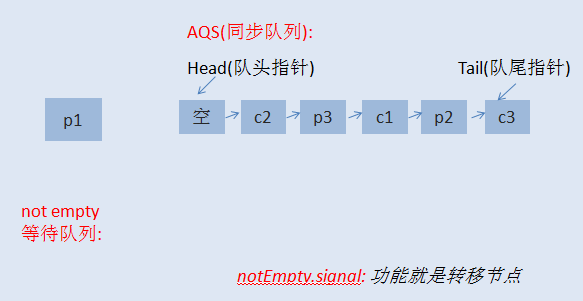
5、p3生产完了产品后，继续notEmpty.signal，同时释放锁，释放锁后会唤醒p2线程，然后p3在下一轮尝试获取锁的时候，再次入队。



6、接着，p2继续生产，生产后执行 notEmpty.signal，同时释放锁，释放锁后唤醒c3线程，然后p2在下一轮尝试取锁的时候，入列。



7、c3进行消费，你可以看到，现在 not empty等待列队中已经没有等待节点了，由于我们使用的是公平策略排它锁，这就会导致同步队列中的节点一个接着一个执行，而目前同步队列中的节点排列为一生产，一消费，这不难可以知道，接下来代码已经不会进入 wait条件了，所以一个一个轮流执行就是，比如c3，执行完了，继续notFull.signal(); 然后释放锁，入队，这里要明白，notFull.signal();这句代码其实没有作用了，因为 not full等待队列中没有任何等待线程节点。 c3执行后，状态如下图所示：



8、后面的事情我想大家都可以想得出来是怎样一步一步交替执行的了

## 16.5 分布式系统中使用内置锁是否有效

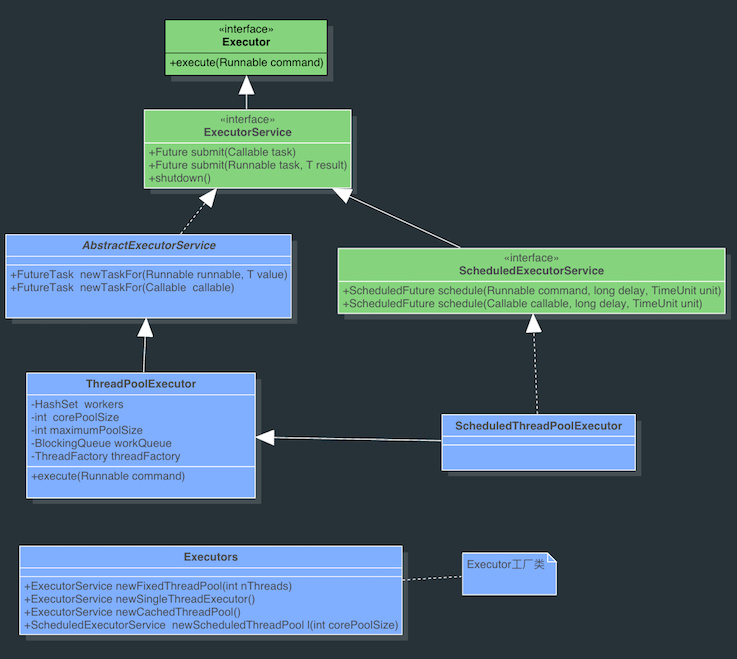
1. synchronized作用范围是单个jvm实例, 如果做了集群,分布式等,就没用了
2. synchronized是作用在对象实例上的,如果不是单例,则多个实例间不会同步(这个一般用spring管理bean,默认就是单例)
3. 单个jvm时,synchronized也不能保证数据库事务的隔离性. 这与代码中的事务传播级别,数据库的事务隔离级别,加锁时机等相关
   1. RR(Repeatable Read)级别.mysql默认的是RR,事务开启后,不会读取到其他事务提交的数据
      1. 执行业务操作时会开启事务
      2. 假设现在有线程T1,T2同时执行事务buy（先查询库存，后扣减库存）方法假设T1先执行,T2等待
      3. spring的事务开启和提交等是通过aop(代理)实现的,所以执行buy方法前,就会开启事务.
      4. 这时候T1,T2是两个事务,当T1执行完后,T2执行,读取不到T1提交的数据,所以会出问题.
   2. RC(Read Committed)级别.事务开启后,可以读取到其他事务提交的数据
      1. 事务流程
         1. 开启事务(aop)
         2. 加锁(进入synchronized方法)
         3. 释放锁(退出synchronized方法)
         4. 提交事务(aop)
      2. 可以看出是先释放锁,再提交事务.所以T2执行查询,可能还是未读到T1提交的数据,还会出问题
4. 引申场景，多线程下，库存扣减如何操作
   1. 简单的加synchronized不能跨jvm，而且也有很小的概率发生错误（T1释放了锁，但是没有提交，T2读取了老的数据）
   2. 颠倒处理逻辑，先扣减库存，再查询，如果库存小于1，则抛异常。（不具备通用性，如果是add操作，就不适用，而且也不能记录扣减前后的库存数量）
   3. CAS，update product set num = 10 where id = 1 and num = old\_num;
      1. 适用CAS后，根据update返回的影响行数，判断有没有执行成功
      2. 如果返回0，则while循环再次查询数据库（隔离级别要是RC，否则会死循环）
      3. 使用该方式可能出现ABA问题
         1. T1，T2读取数据为100,
         2. T1：update num = 90 where id = 1 and num = 100;
         3. T3：update num = 100 where id = 1 and num = 90;
         4. 这个时候T2读取的100已经不是最开始的100了，但是不影响业务
         5. 解决ABA问题，一般需要增加版本字段，每次版本号加一
      4. CAS重试次数需要限制，不然写操作多的时候，会造成数据库大量的重试，影响性能
   4. 使用悲观锁，select for update(性能会有一定影响)
      1. 入口要统一，所有的库存扣减都要使用select for update，不能有普通的select
      2. 加锁顺序，避免死锁
         1. T1锁1,2两条记录
         2. T2锁2,1两条记录
   5. 使用分布式锁（zk，redis），原理与synchronized相同，只是支持跨jvm

# 线程池

## 线程池参数含义

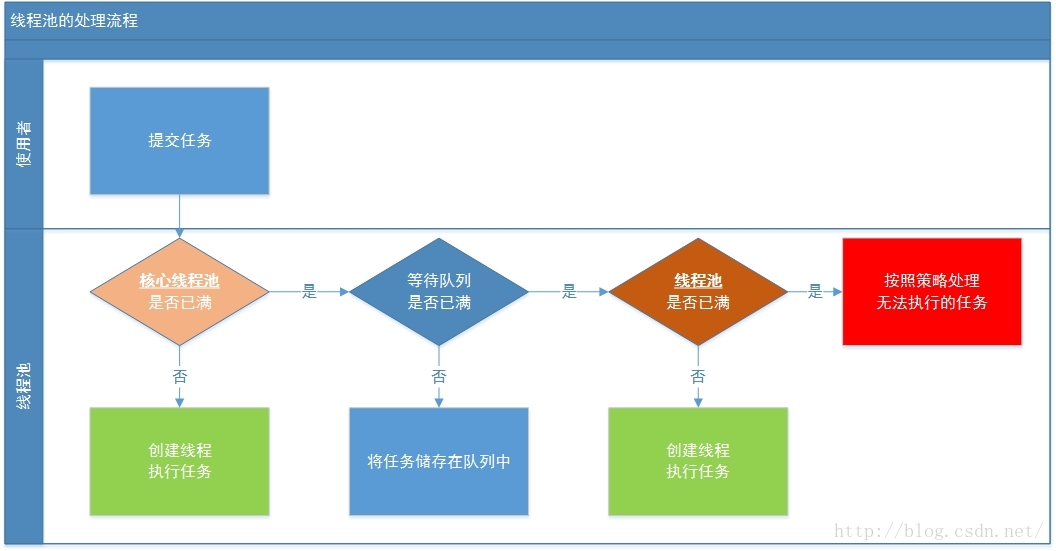
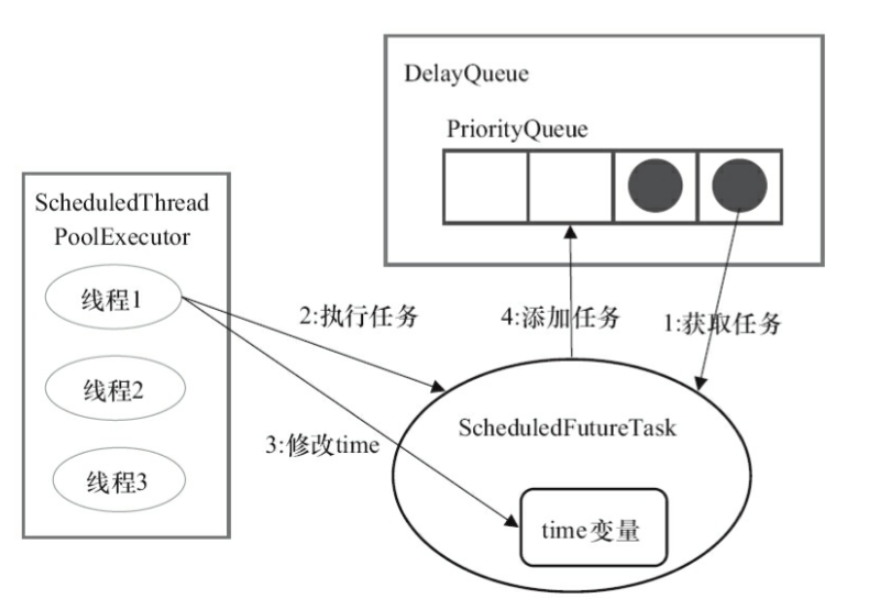
1. corePoolSize：核心池大小，初始化时候，池中没有线程，除非调用prestartAllCoreThreads()或者prestartCoreThread()方法，前者预先创建corePoolSize个线程，后者预先创建一个线程。当线程池中线程数量等于corePoolSize时候，其他的任务就会放入缓存队列。
2. maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程
3. keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。
   1. 默认情况下，只有线程数大于corePoolSize，才会生效，知道线程池中线程数等于corePoolSize。
   2. 但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0。
4. unit：keepAliveTime的时间单位，有天，时，分，秒，毫秒，微秒，纳秒。
5. workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择
   1. ArrayBlockingQueue：有界队列
   2. LinkedBlockingQueue：无界队列
   3. SynchronousQueue：同步队列，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务
6. threadFactory：线程工厂，主要用来创建线程。
7. handler：表示当拒绝处理任务时的策略，有以下四种取值：
   1. ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。
   2. ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。
   3. ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
   4. ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

## 线程池类层次结构



1. Executor顶层接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable/Callable类型，从字面意思可以理解，就是用来执行传进去的任务的
2. Executor顶层接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable/Callable类型，从字面意思可以理解，就是用来执行传进去的任务的；
3. Executor顶层接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable/Callable类型，从字面意思可以理解，就是用来执行传进去的任务的
4. ExecutorService接口继承了Executor接口，并声明了一些方法：submit（返回一个futureTask，可以实现线程的取消，阻塞方式的获取线程返回值）、invokeAll、invokeAny以及shutDown等；
5. 抽象类AbstractExecutorService实现了ExecutorService接口，基本实现了ExecutorService中声明的所有方法；
6. ThreadPoolExecutor继承了类AbstractExecutorService
   1. shutDown
      1. 停止接受外部submit的任务
      2. 内部正在跑的，已经队列中等待的任务，会执行完成
      3. 第二步处理完后，线程池才真正关闭
   2. shutDownNow
      1. 停止接受外部submit的任务
      2. 忽略队列中等待的任务
      3. 使用interrupt，尝试中断正在跑的线程（如果线程中没有，sleep，wait，Condition，定时锁等，interrupt是不生效的）
      4. 返回未执行的任务列表
   3. awaitTermination(timeout,unit)
      1. 等待所有的任务跑完（正在跑的，还有队列中等待的任务）
      2. 在没有到达超时时间，还能继续接受新任务
      3. 等待超时，或者线程被中断
      4. 正常执行完所有任务，然后结束，返回true
      5. 超时返回false（部分任务不能执行完）
      6. 被中断则抛出interruptException（部分任务不能执行完）
7. ScheduledThreadPoolExecutor继承ThreadPoolExecutor，实现ScheduledExecutorService
   1. scheduleAtFixedRate：固定的频率调度一个任务
   2. scheduleWithFixedDealy：相对于任务来说，不管任务执行多长时间，最后都会延迟一定的时间再调度
8. 如何配置线程池大小
   1. CPU密集型：线程数=CPU+1；尽量避免上下文切换
   2. I/O密集型：线程数=2\*CPU；尽量提高CPU使用率
9. 自定义线程池，池大小与阻塞队列选择
   1. 小池大队列：小池容易满，需要大队列存放阻塞线程，例如FixedThreadPool
   2. 大池小队列：大池不容易满，可以使用小队列粗放阻塞线程，例如CachedThreadPool

## 线程池处理流程

1. ThreadPoolExecutor
   1. newFixedThreadPool：定长的线程池，核心线程与最大线程相同，使用无界阻塞队列；适合需要限制线程数量的场景，比如服务器压力比较大，可以通过该方式限制线程数量，线程空闲时间为0，即线程一旦空闲就立马回收
   2. newSingleThreadExecutor：核心线程与最大线程都为1，使用无界阻塞队列，适合串行的保证先后顺序的场景，线程空闲时间为0，即线程一旦空闲就立马回收
   3. newCachedThreadPool：无界线程池，核心线程数0，最大线程数为int的最大值，使用同步阻塞队列，线程空闲1分钟回收；适合并发量大，但是耗时很短的场景，如web简单的查询请求
   4. 
2. ScheduledThreadPoolExecutor
   1. 线程从DelayedWorkQueue中获取一个到期的ScheduledFutureTask(主要继承Delayed接口，实现了getDelay,compareTo,继承Future接口，实现cancel,get等方法)，即time大于等于当前时间
      1. ScheduledFutureTask组成
         1. long型成员变量time：任务开始的具体时间
         2. long型sequenceNumber：加入队列的序列号
         3. long型period：任务执行的间隔周期
         4. 备注：如果任务开始时间相同，则根据入队序列判断哪个任务先执行
   2. 执行该任务
   3. 修改task对象的time为下次开始时间
   4. 把该task加入队列中继续排队，等待下次调度（添加任务时候需要加锁）
   5. 

# 多线程工具类

## CountDownLatch

1. 使用场景
   1. 主线程等待多个子线程执行完后， 才能往下执行
2. 使用方式
   1. CountDownLatch构造函数接受一个int类型参数作为计数器，然后主线程调用await（）会阻塞
   2. 完成一个子线程，计数器减一
   3. 计数器为0后，主线程唤醒，开始执行
3. 注意事项
   1. 初始化后不能修改计数器
   2. 如果构造函数一开始就是传0，主线程调用也不会阻塞
   3. 可以使用await（time，timeUtil），设置指定时间

## CyclicBarrier

1. 使用场景
   1. 需要一组线程全部到达屏障后，再开启屏障执行下一步操作
2. 使用方式
   1. 默认构造方法CyclicBarrier（int parties），定义屏障拦截几个线程
   2. 每个线程调用await（）方法告诉CyclicBarrier，我已经到达屏障，然后当前线程被阻塞
   3. 当所有线程都达到后，开始执行线程各自的逻辑
3. 补充
   1. 提供了一个高级函数CyclicBarrier（int parties，Runnable barrierAction），当线程都达到屏障后，执行action动作

## Semaphore

1. 使用场景
   1. 多线程下访问数量固定的公共资源
2. 使用方式
   1. 构造方法Semaphor（int permits），permits定义公共资源的数量
   2. 线程访问公共资源，首先调用acquire（）方法获取资源，没有获取到，则阻塞
   3. 执行完后，调用release（）释放资源
3. 补充
   1. tryAcquire（）尝试获取，不会阻塞
   2. intavaliablePermits()：返回信号量中当前公共资源的个数
   3. intgetQueueLength()：返回正在等待获取许公共资源的线程个数
   4. booleanhasQueueThreads()：是否有线程正在等待公共资源
   5. reducePermits(int reduction)：减少reduction个公共资源
   6. getQueuedThreads()：返回所有等待获取公共资源的线程集合

## Exchange

1. 使用场景
   1. 用于两个线程间互相交换数据
2. 使用方法
   1. 如果一个线程执行exchange（），会一直等待第二个线程也执行exchange（）
   2. 当两个方法都到达同步点后，就可以交换数据，将本线程产生的数据传递给对方

## Phaser

1. 使用场景
   1. java多线程技术提供了Phaser工具类，Phaser表示“阶段器”，用来解决控制多个线程分阶段共同完成任务的情景问题。其作用相比CountDownLatch和CyclicBarrier更加灵活，例如有这样的一个题目：5个学生一起参加考试，一共有三道题，要求所有学生到齐才能开始考试，全部同学都做完第一题，学生才能继续做第二题，全部学生做完了第二题，才能做第三题，所有学生都做完的第三题，考试才结束。分析这个题目：这是一个多线程（5个学生）分阶段问题（考试考试、第一题做完、第二题做完、第三题做完），所以很适合用Phaser解决这个问题
2. 使用方法
   1. 继承Phaser类，覆盖onAdvance（int phase，int registedParties）方法，表示每个阶段处理的事情，里面可以获取第几个阶段，以及注册的线程数
   2. 每个线程获取phaser对象，然后在指定地方调用phaser.arriveAndAwaitAdvance()
   3. 当所有线程都调用后，开始执行phaser中定义的方法
3. 补充
   1. arriveAndDeregister（）到达屏障后，当前线程注销
   2. arrive（）到达屏障后，不等待其他线程一起，自己接着往下处理
   3. arriveAndAwaitAdvance（）到达屏障后，等待其他线程一起，然后处理公共逻辑
   4. register（）动态的注册一个线程，加入phaser

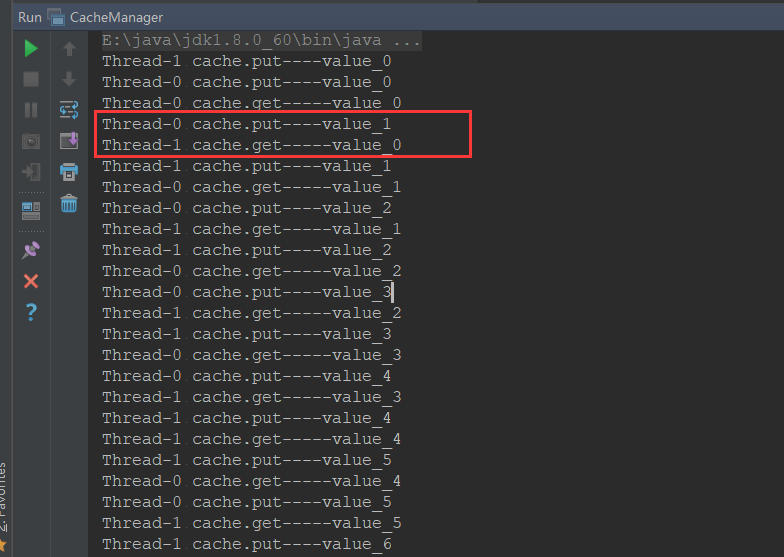
## Fork/Join

1. 使用场景
   1. 将一个大任务拆分成多个子任务，然后并行处理，提高效率
   2. fork将大任务拆分
   3. join将每个子任务的结果统计出来
2. 使用方法
   1. RecursiveAction：继承ForkJoinTask，没有返回
   2. RecursiveTask：继承ForkJoinTask，有返回，返回一个Future
3. 补充
   1. public ForkJoinPool(int parallelism)：创建一个包含指定个数的并行线程的ForkJoinPool
   2. public ForkJoinPool() ：以Runtime.getRuntime().availableProcessors()的返回值作为parallelism来创建ForkJoinPool

# 并发编程陷阱

## 同步不完全(CacheManager)

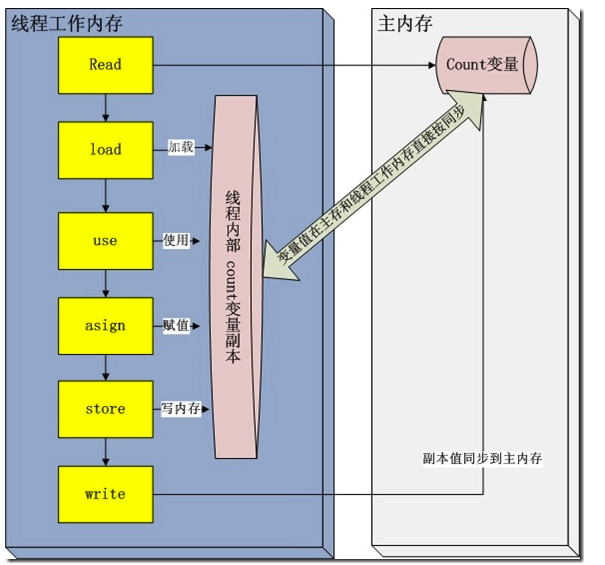
一个简单的缓存类，put操作使用同步，get操作没有使用同步，在多个线程同时put/get同一个key的时候可能导致读取老的数据【将get操作也设置为同步就可以解决该问题，可以使用读写锁优化】。



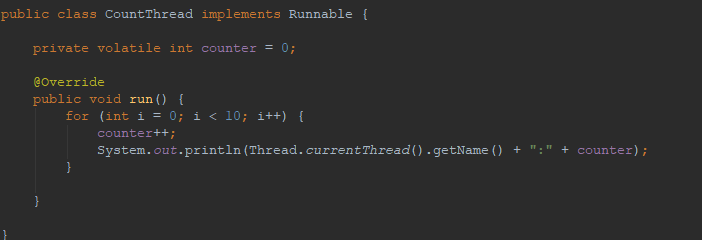
如图，线程0已经put值为“value\_1”，但是线程1还是读取的老的值“value\_0”。

总结：可以并发读，不可以并发写，读时不能写，写时不能读。

## Volatile与变量脏读(CountThread)

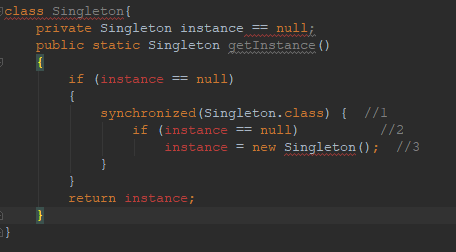


上图是线程直接共享变量的内存模型图，每个线程都会有一个主存的变量拷贝，所以会存在脏读，通过添加volatile关键字，可以保证线程每次都去主存读取变量，保证读取最新数据。



上述计数器，在多线程下是否安全？

## Double check(Singleton)



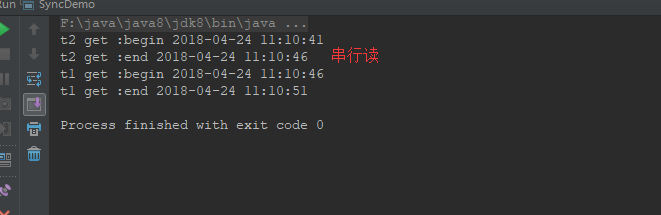
上述单例模式在多线程下是否安全？

解释：

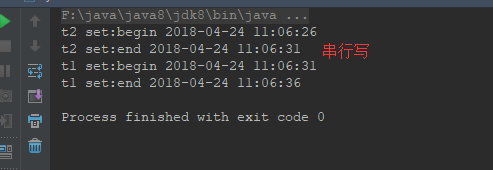
1. 步骤1，加锁，保证只有一个线程进来创建对象
2. 步骤2，再次判空检查，因为可能线程A,B都进入了第一次判空，A获取了锁，B等待锁；A创建实例后释放锁，B获取锁，再次判空，避免B再次创建实例
3. 但是实例的修饰词没有volatile，多线程下，B不能立刻获取到instance的最新值，所以可能B在进行二次判空时候还是判断为null

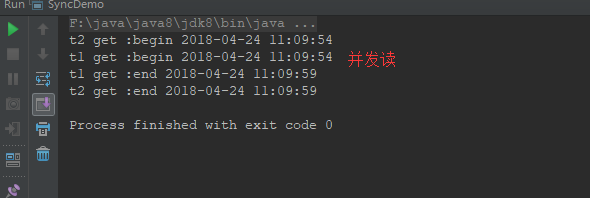
## 读多写少使用Sync锁导致性能下降(SyncDemo)

多线程并发读写某个数据，如果使用同步锁，会导致所有的读操作都必须串行，影响性能。



可以使用读写锁，实现并发读，串行写。





## HashMap多线程下导致CPU运行100%

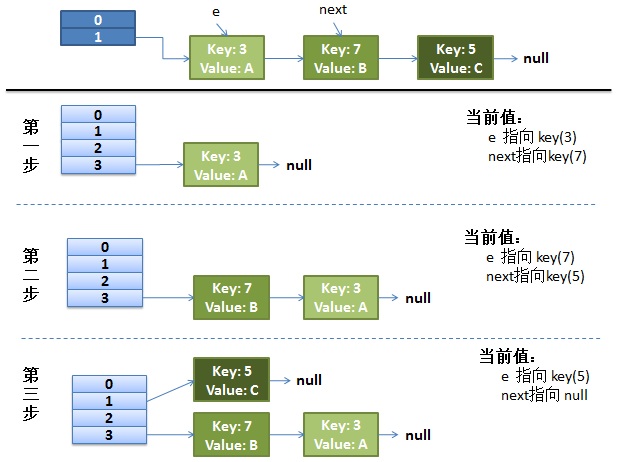
1. HashMap扩容后，元素转移主要代码逻辑

e.next = newTable[i];

newTable[i] = e;

e = next;

1. 单线程下，安全的扩容逻辑

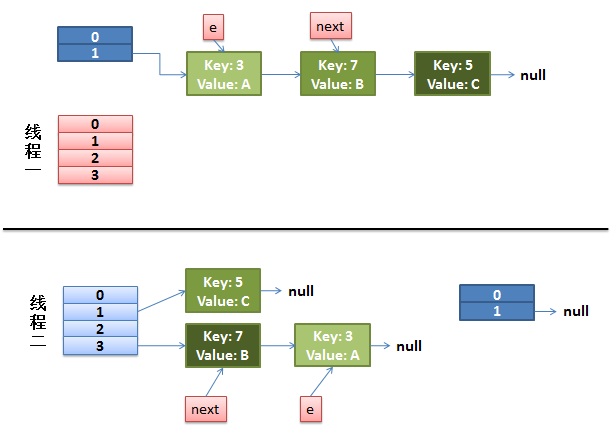


1. 多线程下，非安全扩容

线程一状态：刚生成了新table，还没开始进行数据转移

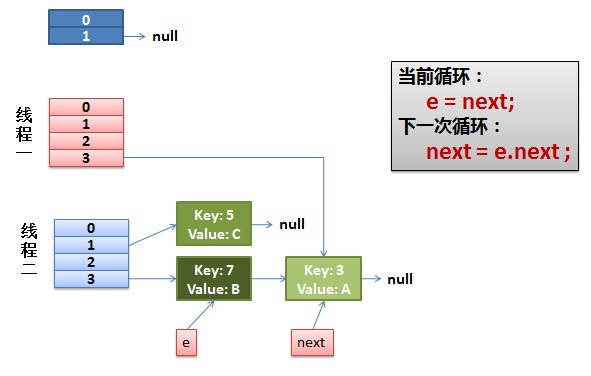
线程二状态：已经将数据转移完成

**此时，e为key3，next为key7**



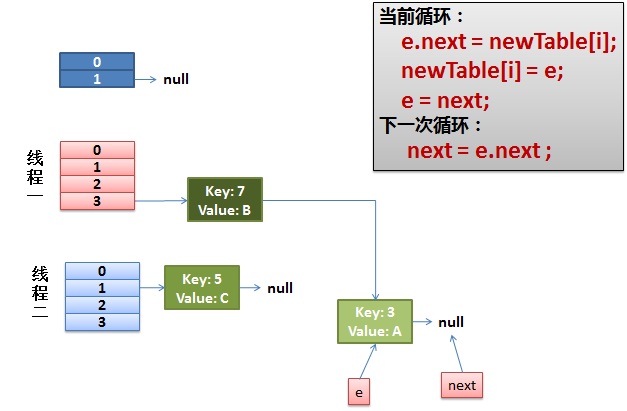
开始第一次数据转移，将key3赋值到table[3]

* 1. e.next = newTable[i]; 将key3的next指针设置为null
  2. newTable[i] = e; 将key3赋值到table[3]
  3. e = next; 将key7设置为e，然后开始新的一轮循环
  4. **此时e为key7，next为key3**【因为线程二已经分配完成，然后线程二中的链表结构为key7指向key3】



开始第二次数据转移，将key7赋值到table[3]

1. e.next = newTable[i]; 将key7的next指针设置为key3【此时table[3]=key3】
2. newTable[i] = e; 将key7赋值到table[3]
3. e = next; 将key3设置为e，然后开始新的一轮循环
4. **此时e为key3，next为null**



开始第三次数据转移，再次将key3赋值到table[3]

1. e.next = newTable[i]; 将key3的next指针设置为key7【此时table[3]=key7】
2. newTable[i] = e; 将key3赋值到table[3]
3. e = next; 此时e=null
4. 环形链出现，table[3]指向key3，key3指向key7，key7又指向key3【**当下次get元素为table[3]时，就会死循环遍历链表**】
5. **此时e为null，next为null**

