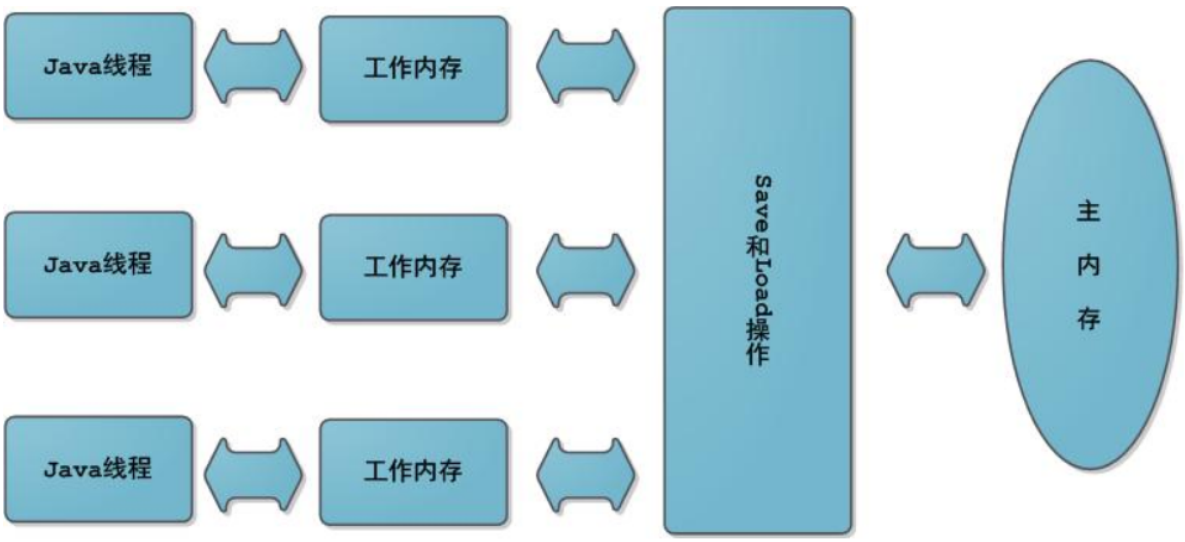
Java并发（锁编程、无锁编程）

2019.03.13 weideng

1、线程工作内存和主内存间数据一致性协议

Java内存模型中规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了该线程使用到的变量到主内存副本拷贝，线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，而不能直接读写主内存中的变量。不同线程之间无法直接访问对方工作内存中的变量，线程间变量值的传递均需要在主内存来完成，线程、主内存和工作内存的交互关系如下图所示



java关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步到主内存之间的实现细节，Java内存模型定义了以下**八种操作**来完成：

1、lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。
2、unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。
3、read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用。把主内存中的变量的值传输到工作内存中的这个过程。
4、load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。将上一步传输过来的值赋给工作内存中变量的副本。
5、use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。比如 i++，会取出i的值。
6、assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。比如 i++，将++之后的值通过assign操作赋值给工作内存中i变量的副本。
7、store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。

8、write（写入）：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

Java内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：

1、不允许read和load、store和write操作之一单独出现

2、不允许一个线程丢弃它的最近assign的操作，即变量在工作内存中改变了之后必须同步到主内存中。

3、不允许一个线程无原因地（没有发生过任何assign操作）把数据从工作内存同步回主内存中。

4、一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量。即就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。

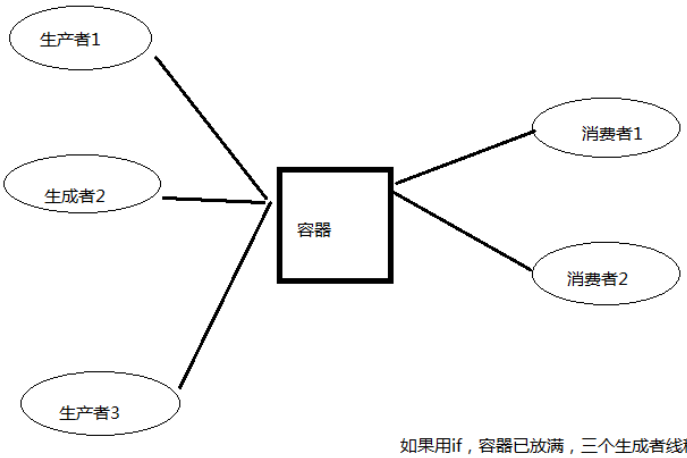
5、一个变量在同一时刻只允许一条线程对其进行lock操作，lock和unlock必须成对出现

6、如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前需要重新执行load或assign操作初始化变量的值

7、如果一个变量事先没有被lock操作锁定，则不允许对它执行unlock操作；也不允许去unlock一个被其他线程锁定的变量。

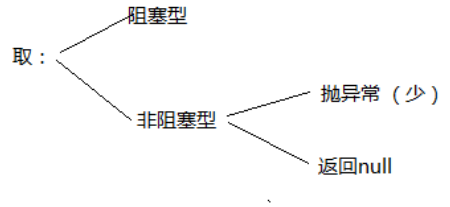
8、对一个变量执行unlock操作之前，必须先把此变量同步到主内存中（执行store和write操作）。

2、ReentrantLock.Demo5中使用while而不能使用if的原因

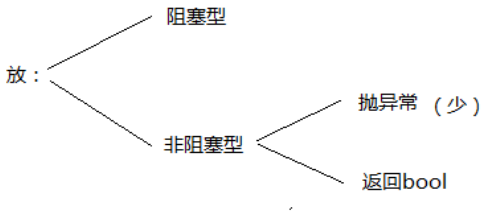


如果用if，容器已放满，三个生成者线程都因为调用put方法而被阻塞。当一个消费者取走一个元素后，notifyAll唤醒三个生产者线程，线程2得到了锁，执行if（满足），放进去一个元素（此时容器又已经满了），线程2放进去元素之后释放锁。线程1得到锁，就不会执行if判断语句了。导致容器放11个元素。使用while，任何一个生产者线程得到锁之后都会重新判断容器是否被放满。

3、并发容器中的存、取操作



take、put



poll、offer

peek、add

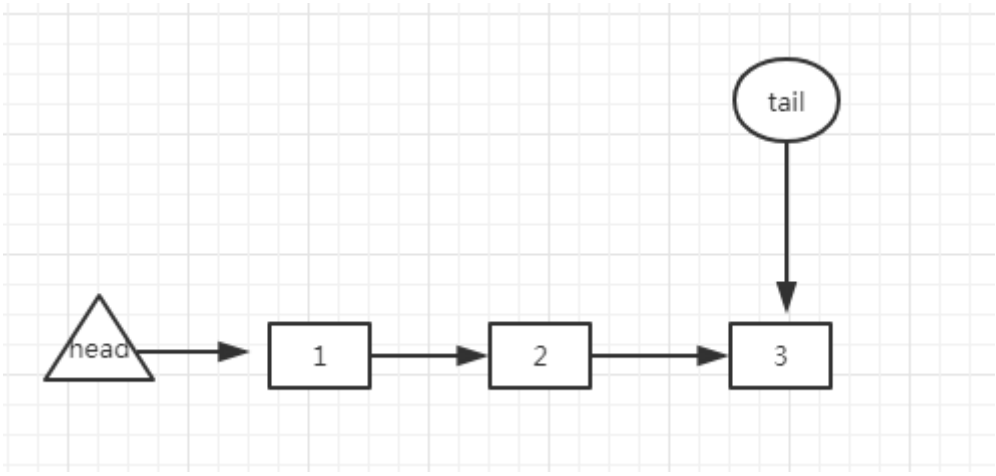
4、解决多线程并发环境下并发冲突的策略

悲观策略：认为出错是容易发生的（锁机制，试图避免一切可能产生并发冲突的情况）。锁机制相关的demo见share313项目中的22个demo实例
乐观策略：认为出错是不容易发生的（CAS，compare and swap比较交换技术，基于冲突检查，意思就是说等真的发生了并发冲突再说）。无锁并发程序可以使性能飞速提升，同时无锁并发编程比有锁并发编程难度也大了几个数量级，需要保证每一行代码之间，甚至一行代码内部，一个操作符在编译成多条指令各指令之间，就算有其他线程挤进来改变了值，也没有关系，不会造成危害，因为这种改变被编程者事先预知了，预料到任何时刻都可能有其他线程挤进来。

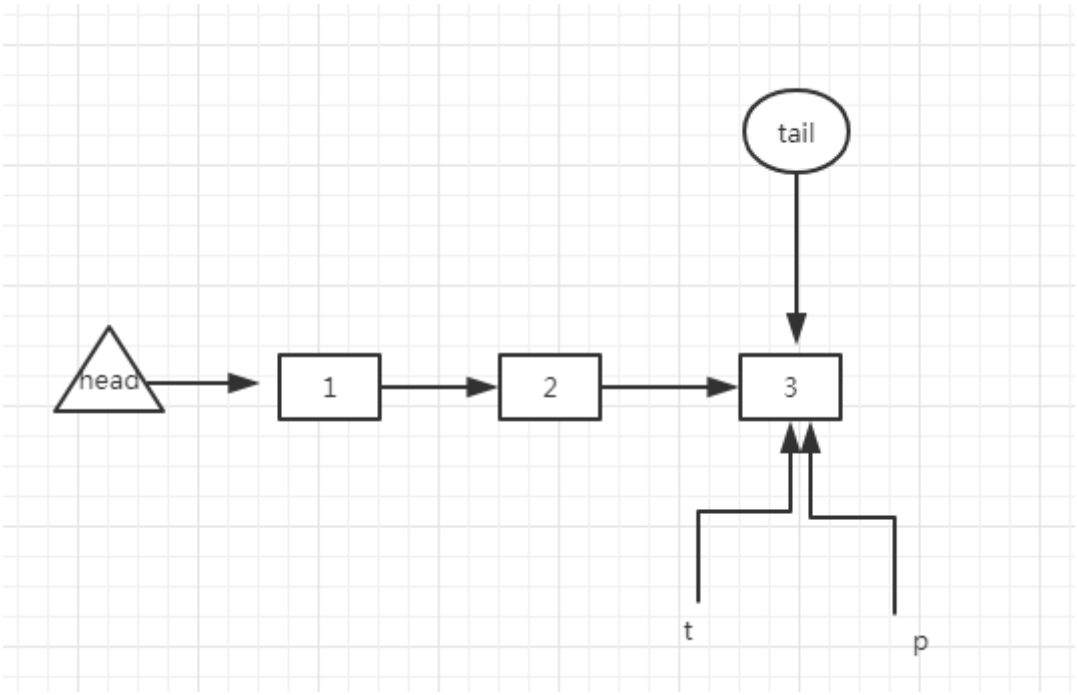
cas ==> 非阻塞型，天然免疫死锁，节省线程之间切换调度开销，节省锁竞争开销，减少线程之间相互影响。
CAS(V, E, N)
参数：V：要更新的变量；E：期望值；N：新值。当要更新的变量的值等于期望值时，才会更新为N。
直接操作内存地址和地址中的数据。比较、更新过程是原子的（由处理器硬件保证其原子性），由封装在UNSAFE类中的一系列方法实现。期望值是线程的工作内存中缓存的值，比较时比较的是工作内存缓存值和主内存中的值。（线程假设只有自己一个线程在运行，自己知道上一次对该变量进行操作后的值是多少，如果比较发现不是这个值了，说明有其他线程存在并且其他线程对该变量进行了修改使得主内存中的变量值和自己预期的不一致了，就不会将V更新成N，而是执行一些重新去主内存读取新值的操作后再次执行CAS操作）。

5、以ConcurrentLinkedQueue为Demo分析CAS技术的使用

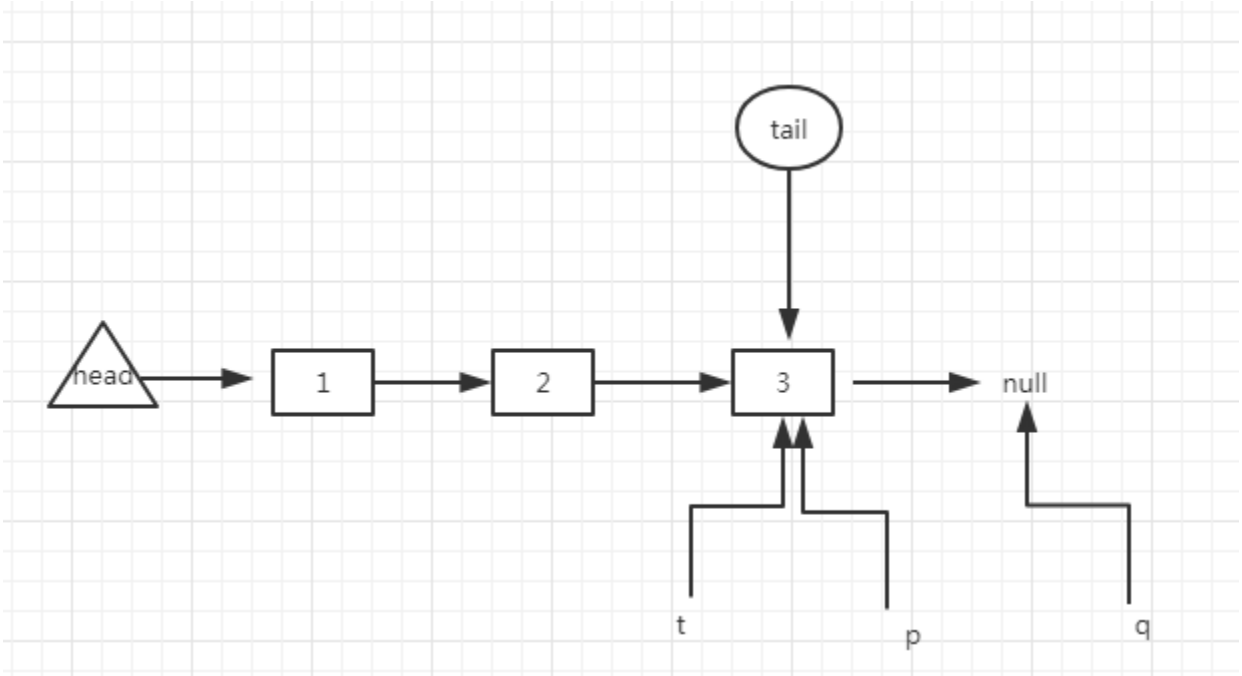
（1）、存过程：offer方法（结合源码）
图1：



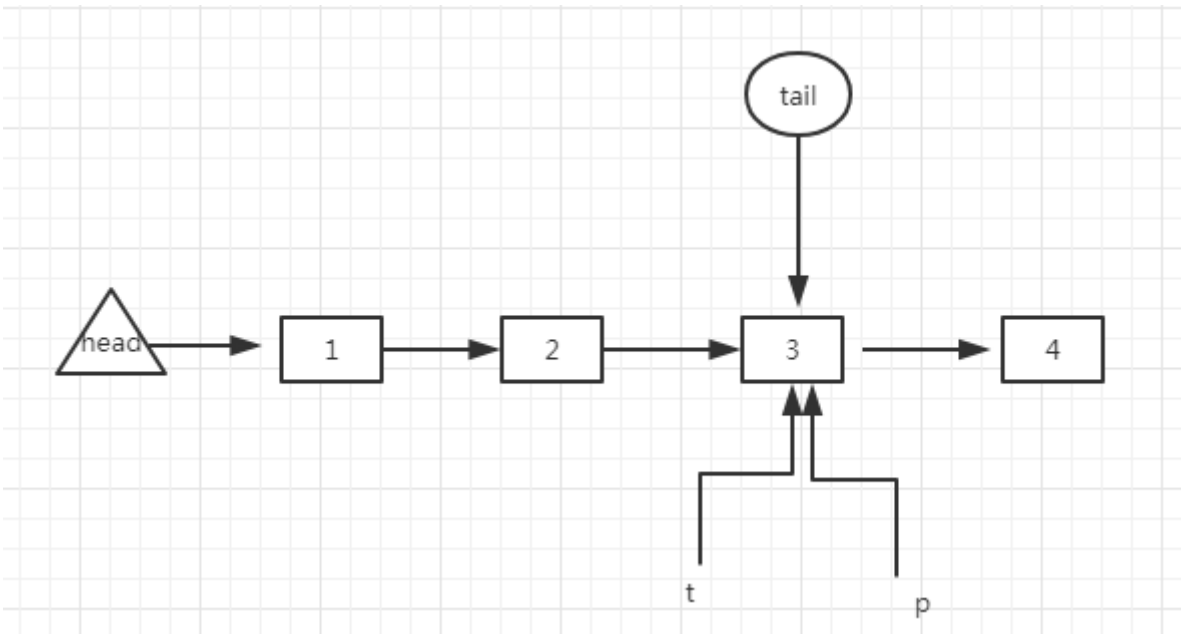
for的初始，t和p都指向tail所指节点
图2：



每次循环开始时，q都指向p的下一个节点，此时q指向null。
图3：

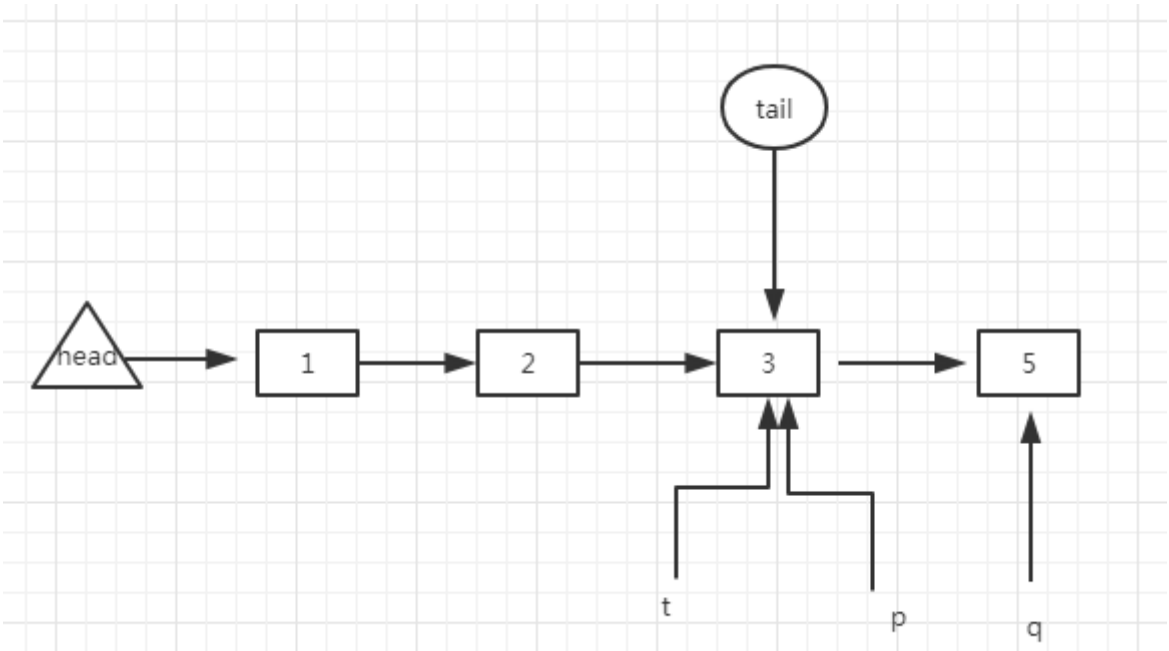


如果q指向的是null，说明tail执行for初始时tail指向的是尾节点。准备执行cas操作，p.casNext(null, newNode)，在cpu硬件层保证原子性执行该条语句。其含义是，如果p所指节点的next域如果为null，就让p所指节点的next域为newNode，即指向newNode。完成插入，函数调用结束。(p 和 t指向同一个节点。所以不会执行casTail（t， newNode），不会更新tail，tail还是指向3号节点）
图4：

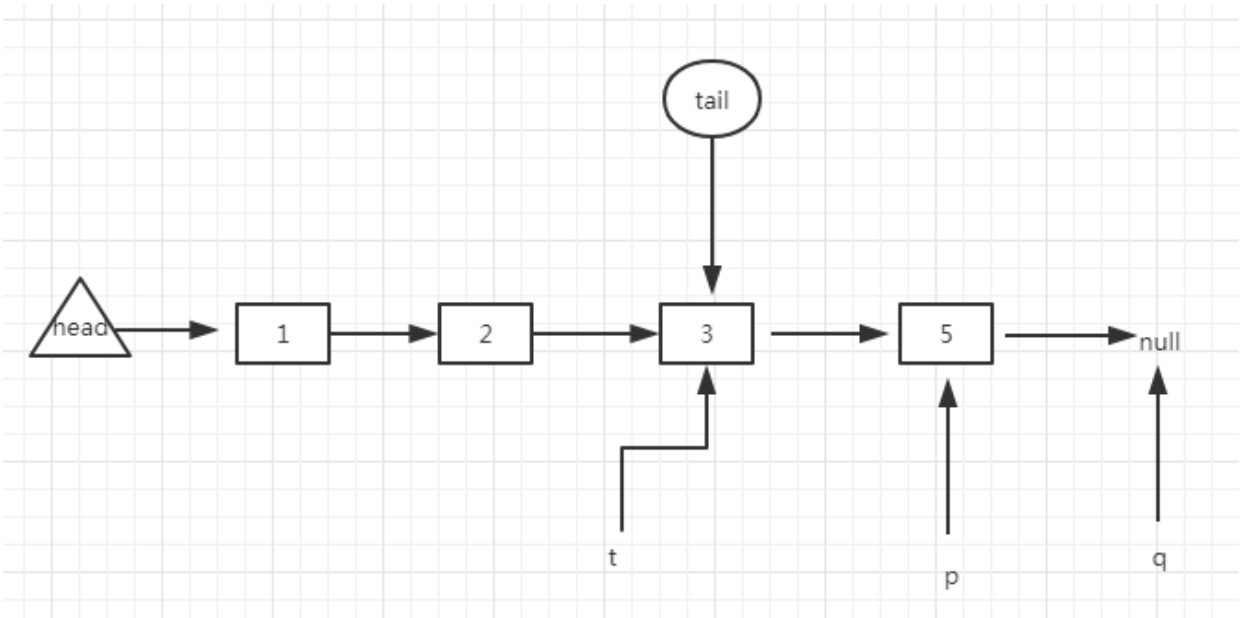


如果两个if之间有其他线程挤进来先插入了一个5号节点，就会导致3号节点（即p所指节点）的next域不为null。p.casNext(null, newNode)就不会把新节点放在p所指节点的后面（如果还放在后面就会导致其他线程刚才放入的那个节点不可达）。转而执行下一次for循环（类似于自旋操作），q明

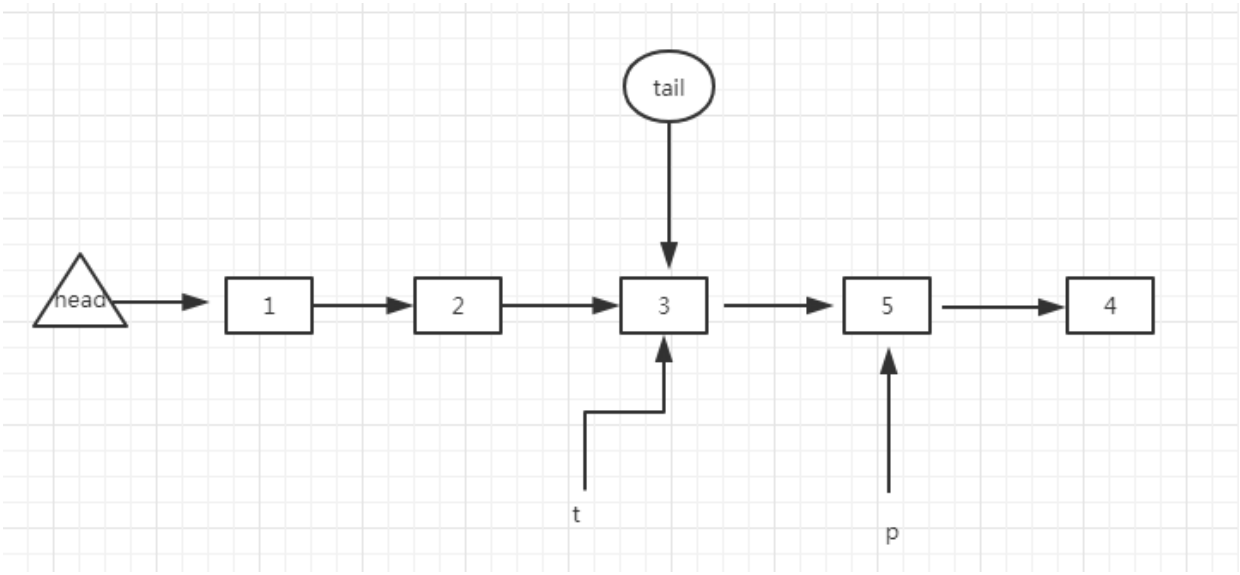
显不为null了，如下图所示：
图5：



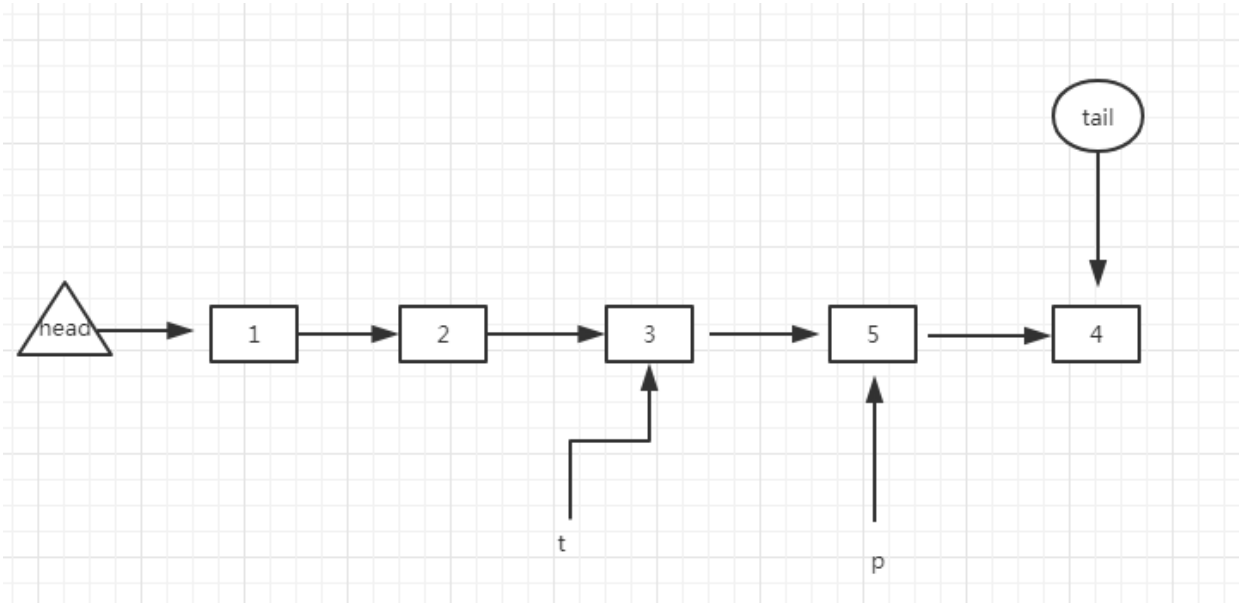
执行p = (p != t && t != (t = tail)) ? t : q; 更新p为q所指节点。进去下一次for循环第一行后q再次指向null。
图6：



进而执行两个if。若两个if中间没有其他线程挤进来（若有线程挤进来，重复以上操作），使得：
图7：



已经将4号节点放到链表中了，此时p和t已经不再是指向同一个节点了。执行casTail（t，newNode），t为期待值，newNode为新值，要更新的变量V为tail。如果tail的值，还是预期值t，就让tail指向新值（即4号节点）。插入4号之后的样子：



t和p是函数栈帧中的变量，函数调用结束后就丢掉了。

疑问：为什么其他线程在插入5号节点的时候没有将tial指向其新插入的5号节点呢
因为其他线程插入5号节点后发现其p 和 t指向的同一个节点，如上图4所示。为什么会出现图4所示的情况呢？因为其他线程在插入5号节点的时候，tail是指向尾部的。指向尾部，插入后就故意不更新tail，而要每2次插入才更新一次tail，为什么要这样呢？因为为了减少使用cas操作更新tail的次数，进一步提高程序效率。

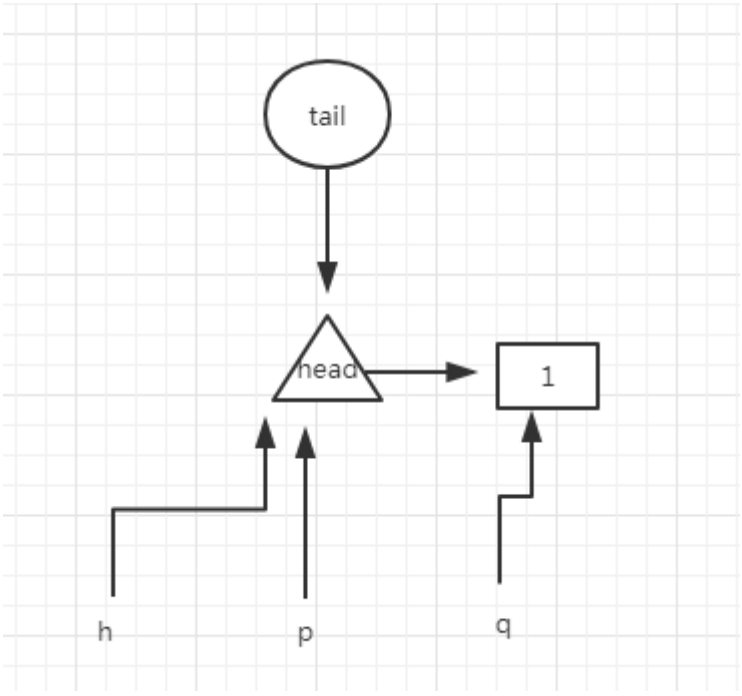
p = (t != (t = tail)) ? t : head;语句
!= 操作不是原子操作，可以被打断。其过程，首先取得t的值，再执行t = tail，t = tail返回t的新值。
（假设tail指向小明），在for（ ）中t被初始化指向tail（即t也指向小明），此处 ！= 左边的t仍指向当时执行for（ ）时的tail所指的那个节点（即小明）。执行完 ！= 的左边后，让t再次指向tail所指的节点。如果在执行 ！= 语句中间没有其他线程挤进来修改tail指针，那么tail

> 就仍然指向小明，t也还是指向小明。t ！= t就不成立。但是执行 ！= 左边两边过程中有其他线程挤进来改变了tail的值，让tail指向了小张，执行完 ！= 右边后，！ = 右边的那个t也指向小张了，于是t ！= t就成立了。那么布尔条件就满足，p就会指向新的tail的值（即小张）。
> 以上是一次赌注，在遇到哨兵节点后，本来应该从head开始从头遍历，但是万一在上次发现t指向的是哨兵节点的时候到执行到这条语句中间的过程有其他线程把tail改变了呢，就不用重头遍历，而只需要让t指向tail改变之后的新节点即可。此时p就指向了最新的tail节点，在此基础上执行下一次for循环。

（2）、取过程：poll方法（结合源码）

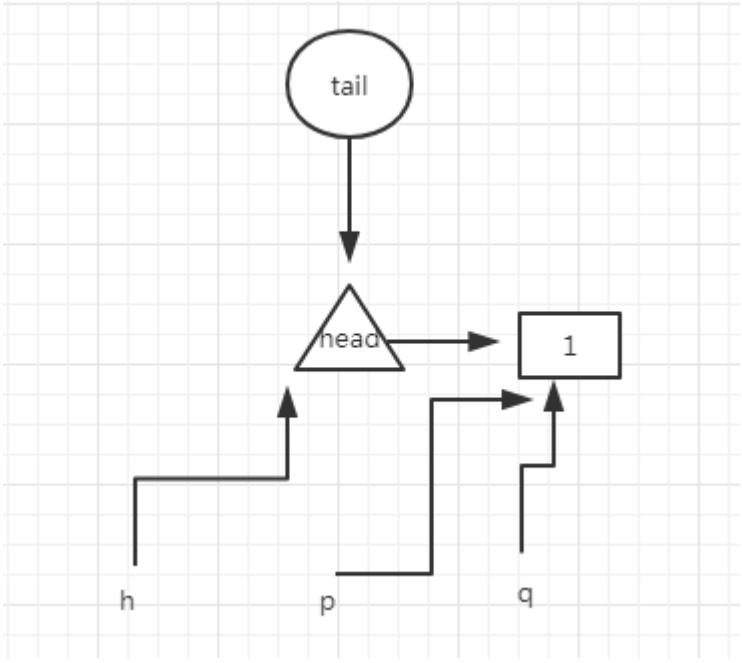
poll，一个for死循环，不是因为容器中没有元素了而死循环等待，而是在取元素的时候其他线程正好也在取，才进行下一次循环取，取到就结束循环。

图1：

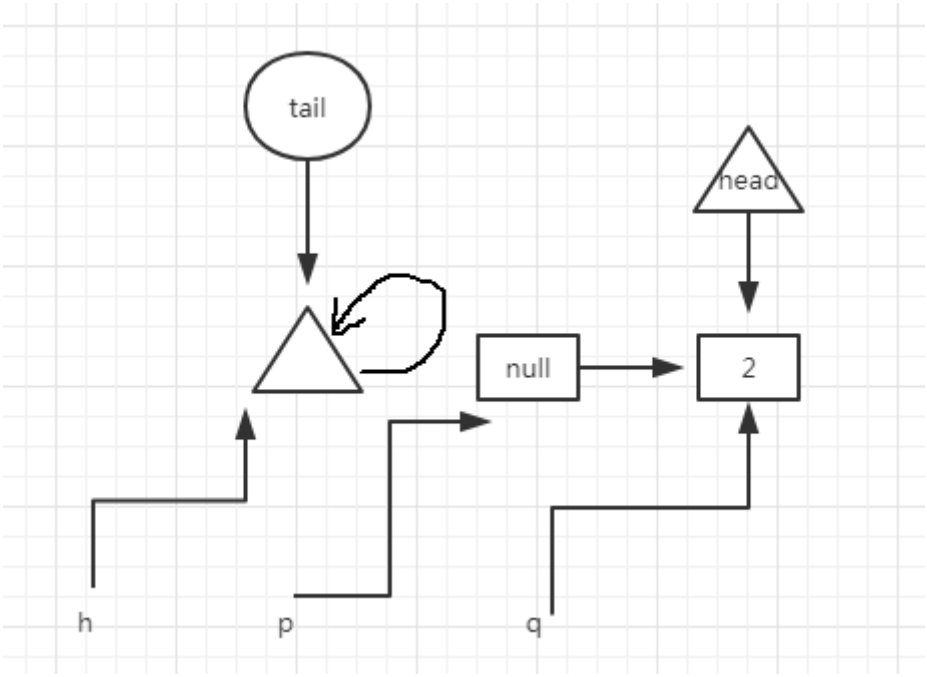


p指向head，其item肯定为null，第一个if不满足，判断else if，使q指向p的next，即1号节点（如图1）。q不为null，else if不满足，直接执行else，使得p和q指向同一个节点（1号节点），如图2

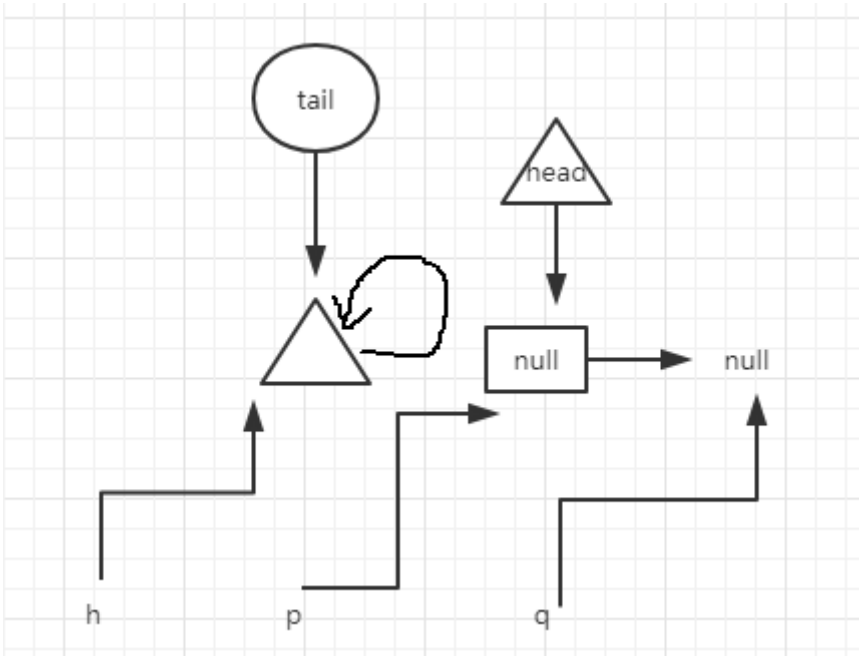
图2：



转入下一次循环，此时p的item不再为null，如果第一个if执行成功（即中间没有其他线程来将1号元素取走，若被其他线程取走，就会执行第一个else if返回null），p ！= h满足，执行updateHead（h，（（q = p.next）！= null）？q ：p）。首先q被更新成p的下一个节点。如果q不为null（即后面还有2号节点，q就指向的2号节点，就让head更新为2号节点），如图3：
图3：



如果后面没有2号节点，就让head指向p（即1号节点，被弹出去的那个节点），如图4：
图4：



综上：有2号节点，2号节点就成为head（因此head节点也是存储数据的节点，for开始时要从head开始），无2号节点1号节点就成为head（此时head节点不存储数据）。不管有没有2号节点，原来那个head节点都成为了哨兵节点（自己指向自己）。由于在poll过程中不改变tail指针，因此tail还是指向那个哨兵节点的。也就是第344行代码存在的意义。