

1. HandlerThread ?

2. join() ?

3. 生产者&消费者模式 ? // curQ => split => concept => more

a. `sharedVar[repo]`; // rps MAXCAP = 1;

b. `producer(sync)`: `sync(this){while(rps >= MAXCAP){wait();} cnt++; notifyAll();}` // swwn rps >= maxcap

c. `consumer(sync)`: `sync(this){ while(cnt<= 0) {wait();} cnt--; notifyAll();}`

d. `sync`: 获取对象锁 `objX.mntr`, 锁住的是对象(资源), 当前线程调用`sync`对象方法, 持有对象锁之后, 才能操作对象的成员变量(这就是为啥成员变量一定要`private`, 只能通过对象方法操作);

i. `objX.mntr`

1. `notify()` & `wait()`通过什么关联?

a. 对象的同步锁 // `objX.monitor`

b. 同步锁是对象持有的, 每个对象只有一个

retrieve:

eg.rqs ->apply ->sum

```
sync(this){ // put curThr to objX.mntr.owner / put curThr to objX.mntr.syncSet
```

```
    while(repo >= max){ // consumer
```

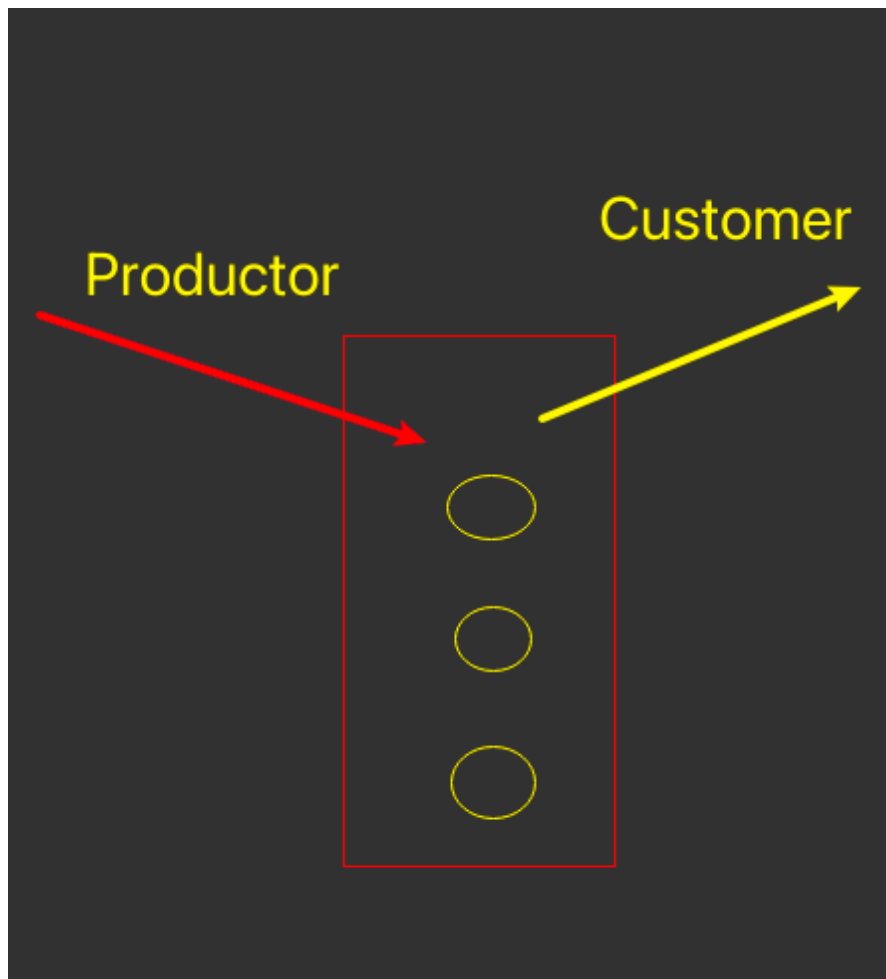
```
        wait(); // put curThr to waitSet, release objX.mntr.owner
```

```
    }
```

```
    repo++;
```

```
    notifyAll(); // put allThr of
```

```
}
```



```

1 // curQ => sample& specific eg => concept => more
2 repo = 0; // rps sharedVar
3 MAXCAP = 1; // 1 customer, 2 producer
4
5 // producer: //swwn=> sync while(repo>=MAXCP){wait();}
6 sync P(this){ // monitor.enter() // curThr locate on objX.monitor.owner[get
objX.monitor] / put curThr to objX.mntr.syncQueue
7     while(rps >= MAXCAP){ // while ? avoid multiple producers[2 producers] be
notified, bf cnt++ judge the condition again
8         wait(); // put curThr to objX[this].mntr.watiQueue -> curThr release
objX.mntr(mntrX.exit)
9     }
10    repo++;
11    notifyAll(); //=> what be notified ? all thrs of objX.mntr.waitQueue //
migrate allThrs from objX.mntr.waitQueue to objX.mntr.syncQueue
12    // mntrX.exit()
13 }
14
15 // consumer:
16 sync C(this){
17     while(rps <= 0){
18         wait();
19     }
20     repo--;
21     notifyAll();
22 }

```

```

1 Util: retrieve, repo,max; swwn
2
3 MAX = 1; // 2 producer, 1 customer
4 prod: thr_0
5 synchronized(Util.class){ // java.lang.Class
6     while( repo>= MAX){

```

```

7      wait(); // put curThr to Util.class.monitor.waitQueue, curThr release
      objX.monitor
8  }
9      i++;
10     notifyAll(); //=> what be notified ? all thrs of objX.mntr.waitQueue
11 }
12
13 consumer: thr_1
14 synchronized(Util.class)
15     while(repo <= 0){
16         wait(); //=> curThr put on objX.mntr.waitQueue, release objX.monitor
17     }
18     i--
19     notifyAll(); //=> what be notified ? all thrs of objX.mntr.waitQueue
20 }
21

```

```

1 retrieve:
2 private static volatile INS; s
3 public static getINS(){ // insin
4     if(null == INS){
5         sync(Singleton.class){
6             if(null == INS){
7                 INS = new Instance()
8             }
9         }
10    }
11    return INS;
12 }
13

```

```

1 retrieve: apply firstly // handler loop

```

```

2 // handle
3 handler = new Handler(Looper.getMainLooper()){
4     handleMessage(){
5         ...
6     }
7 }
8
9 // sendMsg
10 Msg msg = obtainMsg()
11 msg.what
12 msg.obj
13 handler.sendMessage(msg);

```

```

1 retrieve:
2     thr: person
3     mtd: tools
4     data: src
5
6     waitSet, syncSet, owner // thr
7     waitSet -> syncSet -> owner[thr activity]
8
9     sync P(){ // acquire/release
10         while(rpsCnt >= maxcap){
11             wait(); // waitSet
12         }
13         rpsCnt++;
14         notifyAll(); // waitSet to syncSet
15     }

```

apply:

1. mainThr new Handler of subThread ? // handler hold subThr's looper ?
 - a. Handler handlerX = new Handler(handlerThread.getLooper()); // loop() run on subThread;
 - b. handlerX hold the looperX of subThread;

```
void tst(){
    // 1. run loop() on thrX
    HandlerThread handlerThread = new HandlerThread( name: "handlerThreadX");
    handlerThread.start();

    // 2. define handleMsg & sendMsg
    HandlerX handlerX = new HandlerX(handlerThread.getLooper());
    Message message = Message.obtain();
    message.what = 1;
    handlerX.sendMessage(message);

    // 3. quit loop() terminate thrX
    handlerThread.quitSafely();
}

static class HandlerX extends Handler {

    HandlerX(Looper looper){
        super(looper);
    }

    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        // do sth...
    }
}
```

```

public void run() {
    mTid = Process.myTid();
    Looper.prepare();
    synchronized (this) {
        mLooper = Looper.myLooper();
        notifyAll();
    }
    Process.setThreadPriority(mPriority);
    onLooperPrepared();
    Looper.loop();
    mTid = -1;
}
}

/**
 * This method returns the Looper associated with this thread. If this thread not been started
 * or for any reason isAlive() returns false, this method will return null. If this thread
 * has been started, this method will block until the looper has been initialized.
 * @return The looper.
 */
public Looper getLooper() {
    if (!isAlive()) {
        return null;
    }

    // If the thread has been started, wait until the looper has been created.
    synchronized (this) {
        while (isAlive() && mLooper == null) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
    }
    return mLooper;
}
}

```

HandlerThread

same objX

1. **wait**: curThrX get mntrX.lock, put curThrX to mntrX.waitQueue, release mntrX.lock;
 - a. sync ? // get mntrX, curThr could invoke syncMtd();
 - b. Object.wait, Thread.wait ? // curThr put to mntrX.waitQueue;
2. **notifyAll**: curThrX get monitorX, migrate all thrs of waitQueue to syncQueue -> after wait, waitThrX get monitorX & CPU resource -> execute code(will use original thread ?) // yes, waitThrX

src:

1. https://blog.csdn.net/qq_45731021/article/details/116502429
2. https://blog.csdn.net/weixin_58104242/article/details/123376882
3. <https://blog.csdn.net/LWYYYYYYY/article/details/116155143>
4. https://blog.csdn.net/RuiKe1400360107/article/details/111182406?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-2&spm=1001.2101.3001.4242.1
5. <https://zhuanlan.zhihu.com/p/89710060>
6. <https://www.cnblogs.com/myseries/p/13903051.html>
7. <https://www.cnblogs.com/breakingbrad/p/12711800.html>
8. **锁和监视器的区别**: <https://www.jianshu.com/p/ed9e616bfad3>

sum:

2. notifyAll notify which thrX ?

- a. 等待和唤醒必须是同一个锁 objX.mntr; 操作同步线程时, 必须要标识它们操作线程的锁, 只有同一个锁上的被等待线程, 可以被同一个锁上的notifyAll() 唤醒, 不可以对不同锁中的线程进行唤醒。
- b. thr1: obj1.sync mtd(){while(isLoop) wait();} //thr1 -> wait on obj1.waitQueue
- c. thr2: obj2.sync mtd(){while(isLoop) wait();} //thr2 -> wait on obj2.waitQueue
- d. thr3: obj1.sync mtd(){isLoop = false; notifyAll();} // notify all obj1.thrX // thrX migrate from waitQueue(waiting) to syncQueue(blocked), get obj1.monitor -> exec code

3. notifyAll(), wait() must be invoked on non-static sync mtd

- a. they are belong to objX(all objX extends Object) // couldn't invoke by static mtd
- b. notifyAll(), notify, wait(), should be invoke on same objX(monitorX);

4. wait 能不能在 static 方法中使用? 为什么?

- a. 不能, 因为wait 方法是实例方法(非 static 方法), 因此不能在 static 中使用

1. sync:

- a. 对象的锁将保护整个方法, 要调用该方法, 线程必须获得内部的对象锁 // sync 锁住的是对象方法(代码块)的执行权 => 成员变量必须是private的, 只能通过方法来操作, 保证线程安全;
- b. 锁是对象内存堆中头部的一部分数据。JVM中的每个对象都有一个锁 (或互斥锁), 任何程序都可以使用它来协调对对象的多线程访问。
- c. 监视器是一种同步模型, 锁为实现监视器提供必要的支持

5. wait 为什么使用while判断? 被唤醒后重新判断条件是否成立

- a. 如果有(2P1C)两个生产者P1和P2, 一个消费者C, repo最大为1;(多个生产者和消费者)
 - i. 当存储空间满了之后, 生产者P1和P2都被wait, 进入等待队列;
 - ii. 当消费者C取走了一个数据后, 如果调用了notifyAll (), 注意, 此处是调用notifyAll (),
 - iii. 生产者线程P1和P2都将被迁移到同步队列等待锁释放, 如果此时P1和P2中的wait不在while循环中而是在if中, 则P1和P2都不会再次判断是否符合执行条件, 直接执行wait()之后的程序,
 - iv. 如果P1放入了一个数据至存储空间, 则此时存储空间已经满了; 但是P2还是会继续往存储空间里放数据, 错误便产生了;

6. 为啥不能用notify而使用notifyAll ?

- a. notify 只会唤醒调用对象的等待队列中任意一个线程进入到同步队列等待获取锁;

7. notify 使用场景 ?

- a. 当前对象锁只有两线程使用;

8. 对于释放对象监视器, wait()方法和notify()/notifyAll()区别:

- a. 锁对象调用wait()方法之后, 会立即释放对象监视器。
- b. notify()/notifyAll()不会立即释放, 等到线程剩余代码执行完毕之后才会释放;#fefd95

9. wait(long timeout)

- a. 等待timeout的时间, 超过timeout时间, 进入同步队列, 等待途中是可以被notify唤醒的
- b. wait()一直在 objX.mntr.waitSet 中等待;

10. wait和notify为什么要放在synchronized中? //每一个对象都有一个与之对应的监视器Mntr(), 每一个监视器里面都有(一个该对象的锁)一个等待队列和一个同步队列; // 监视器是一种同步模型, 锁[objX.mntr.owner]为实现监视

器提供必要的支持

a. wait & notify 操作都依赖于对象监视器(objX.mntr)相关, 所以要使用在同步中(sync), 因为只有同步才能获取对象锁(mntr); // sync(objX){ objY.wait() } ?

b. wait方法的语义有两个:

- 使得当前线程进入等待队列
- 释放当前的对象锁, 让别的线程继续竞争锁;
- 这些操作都和监视器是相关的, 所以wait必须要获得一个监视器锁。

a. notify

- 唤醒一个线程, 需要知道待唤醒的线程在哪里, 就必须找到这个对象获取这个对象的锁然后去到这个对象的等待队列去唤醒一个线程

i. wait:

- put curThr to objX.mntr.waitQueue // need get objX.mntr;
- release objX.mntr;

ii. notifyAll: migrate allThrs of objX.mntr.waitQueue to objX.mntr.syncQueue; need get objX.mntr;

iii. 洗手间上锁[monitor.enter; put curThr to objX.mntr.ower]-> 处理 -> 开锁(monitor.exit)

c. 防止lost wakeup

i. C: while(cnt<=0){wait();};

ii. P: cnt++; notify();

- while(cnt <= 0) -> cnt++ , notify -> wait() // lost wake up -> wait()

我们知道, wait()的作用是让当前线程由“运行状态”进入“等待(阻塞)状态”的同时, 也会释放同步锁。而sleep()的作用也是让当前线程由“运行状态”进入到“休眠(阻塞)状态”。

但是, wait()会释放对象的同步锁, 而sleep()则不会释放锁。

```
public void run() {  
    // 获取obj对象的同步锁  
    synchronized (obj) {  
        try {  
            for(int i=0; i <10; i++){  
                System.out.printf("%s: %d\n", this.getName(), i);  
                // i能被4整除时, 休眠100毫秒  
                if (i%4 == 0)  
                    Thread.sleep(100);  
            }  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

will not release lock of obj

Object中的wait(), notify()等函数, 和synchronized一样 会对“对象的同步锁”进行操作。

wait()会使“当前线程”等待 因为线程进入等待状态, 所以线程应该释放它锁持有的“同步锁” 否则其它线程获取不到该“同步锁”而无法运行!

OK, 线程调用wait()之后, 会释放它锁持有的“同步锁”; 而且, 根据前面的介绍, 我们知道: 等待线程可以被notify()或notifyAll()唤醒。现在, 请思考一个问题: notify()是依据什么唤醒等待线程的? 或者说, wait()等待线程和notify()之间是通过什么关联起来的? 答案是: 依据“对象的同步锁”。

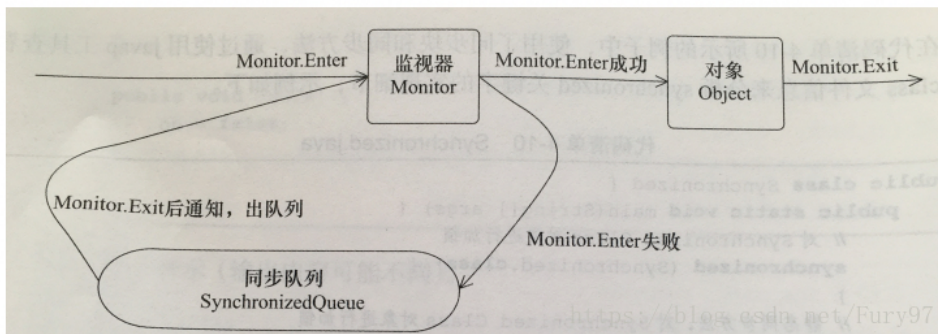
负责唤醒等待线程的那个线程 我们称为“唤醒线程”, 它只有在获取“该对象的同步锁”(这里的同步锁必须和等待线程的同步锁是同一个), 并且调用notify()或notifyAll()方法之后, 才能唤醒等待线程。虽然, 等待线程被唤醒; 但是, 它不能立刻执行, 因为唤醒线程还持有“该对象的同步锁”。必须等到唤醒线程释放了“对象的同步锁”之后, 等待线程才能获取到“对象的同步锁”进而继续运行。

总之, notify(), wait()依赖于“同步锁”, 而“同步锁”是对象锁持有, 并且每个对象有且仅有一个! 这就是为什么notify(), wait()等函数定义在Object类, 而不是Thread类中的原因。

在Object.java中, 定义了wait(), notify()和notifyAll()等接口。wait()的作用是让当前线程进入等待状态, 同时, wait()也会让当前线程释放它所持有的锁。而notify()和notifyAll()的作用, 则是唤醒当前对象上的等待线程; notify()是唤醒单个线程, 而notifyAll()是唤醒所有的线程。

在继续介绍之前, 我们先了解一下使用synchronized关键字时, 对象, 监视器, 同步队列和执行线程之间的关系。

同步队列



synchronized关键字对对象的获取锁与释放锁, 实质上是使用了monitorenter和monitorexit指令, 本质是对于一个对象的监视器(monitor)的获取与释放, 这个获取是排他的, 也就意味着同一时刻只能有一个线程获取到对象的监视器。

由图我们可以看到这样的过程:

1. 线程访问对象, 首先使用monitorenter来获取对象监视器。
2. 如果获取成功, 则线程继续执行。
3. 如果获取失败, 说明已有其他线程获得了该对象的锁, 则该线程进入同步队列, 线程状态变为BLOCKED (阻塞)。
4. 当之前已获得该对象的锁的线程释放锁时, 该释放操作会唤醒阻塞在同步队列中的线程, 使其继续尝试获取该对象的监视器。

retrieve:

1. executor: thr = persons; A, B, C persons
2. TaskSet(mtd&data): objX = TaskSet
sync(this){ // objX's waitQueue, syncQueue;
waitQueue,
}
sync(this) => syncQueue: store thr, could get monitor
objX.wait() => waitQueue; store thr, waitNotify

retrieve:

1. wait & notifyAll

a. why under sync ? lost wake up

i. C: while(cnt<=0){wait();};

ii. P: cnt++; notify();

1. while(cnt <= 0) -> cnt++ , notify; // lost wake up -> wait()

b. why Object.wait notifyAll ?

i.

c. why use while ?

Object

.wait();

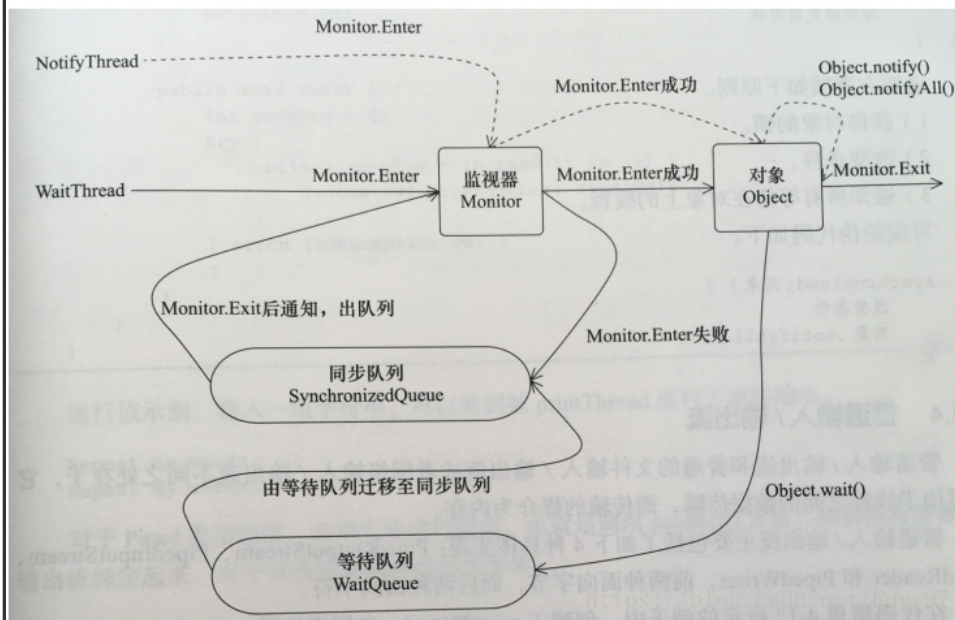
.notifyAll();

objX.mntr.owner = curThr;

objX.wait();

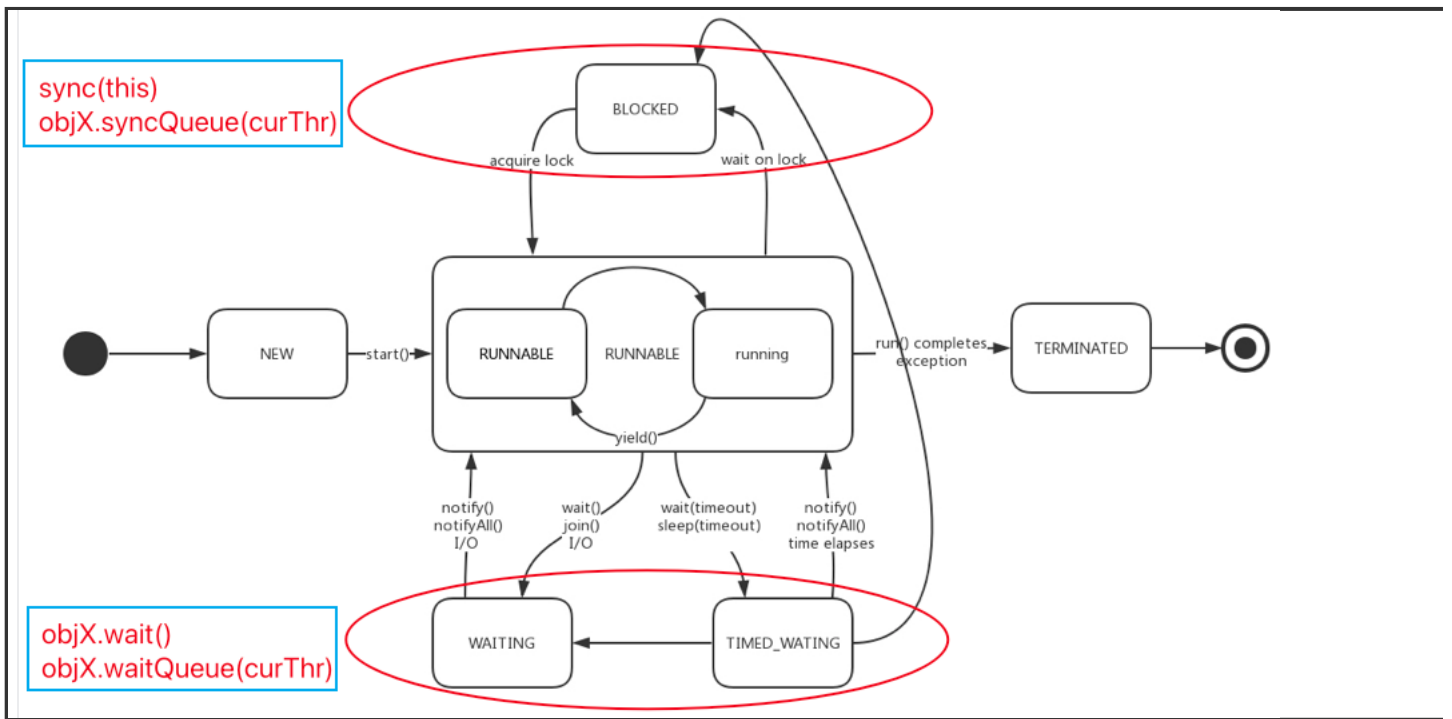
了解了同步队列之后，我们进入正题，看一个等待/通知的示例。

示例



由图我们可以看到这样的过程：

- `WaitThread`首先`Monitor.Enter`获取对象的监视器（获取锁），获取成功后，调用对象的`wait()`方法，进入了等待队列`WaitQueue`，进入等待状态，同时释放了锁（这个很关键，调用`wait`的同时会释放锁）。
- `NotifyThread`之前因为获取锁失败，进入了同步队列阻塞中，由于`WaitThread`释放了锁，`NotifyThread`自然获得了对象的锁，并继续执行，调用了对象的`notify()`方法。
- `WaitQueue`等待队列中的`WaitThread`收到了`notify`信号，从`WaitQueue`等待队列迁移至同步队列，`WaitThread`从等待状态进入阻塞状态。
- `NotifyThread`执行完毕，释放了锁，由于`WaitThread`在同步队列中，自然就获得了锁，然后继续执行。



sum

1. wait & notifyAll ? thrX, objX

- wait():** sync(objX) -> **curThrX** wait on waitQueue of **objX**; // thrX, objX
- notifyAll():** sync(objX) -> migrate thrXs of ObjX from waitQueue to syncQueue(block, acquire lock)

等待/通知机制的经典范式：

该范式分为两部分，分别针对等待方（消费者）和通知方（生产者）。

等待方遵循如下原则

1. 获取对象的锁。
2. 如果条件不满足，那么调用对象的wait()方法，被通知后仍要检查条件。
3. 条件满足则执行对应的逻辑。

对应的伪代码如下：

```
1 synchronized(对象){  
2     while(条件不足){  
3         对象.wait();  
4     }  
5     对应的处理逻辑  
6 }
```

通知方遵循如下原则

1. 获得对象的锁。
2. 改变条件。
3. 通知所有等待在对象上的线程。

对应的伪代码如下：

```
1 synchronized(对象){  
2     改变条件  
3     对象.notifyAll();  
4 }
```

Lost Wake-Up Problem

事情得从一个多线程编程里面臭名昭著的问题"Lost wake-up problem"说起。

这个问题并不是说只在Java语言中会出现，而是会在所有的多线程环境下出现。

假如有两个线程，一个消费者线程，一个生产者线程。生产者线程的任务可以简化成将count加一，而后唤醒消费者；消费者则是将count减一，而后在减到0的时候陷入睡眠：

生产者伪代码：

```
count+1;

notify();
```

消费者伪代码：

```
while(count<=0)

    wait()

count--
```

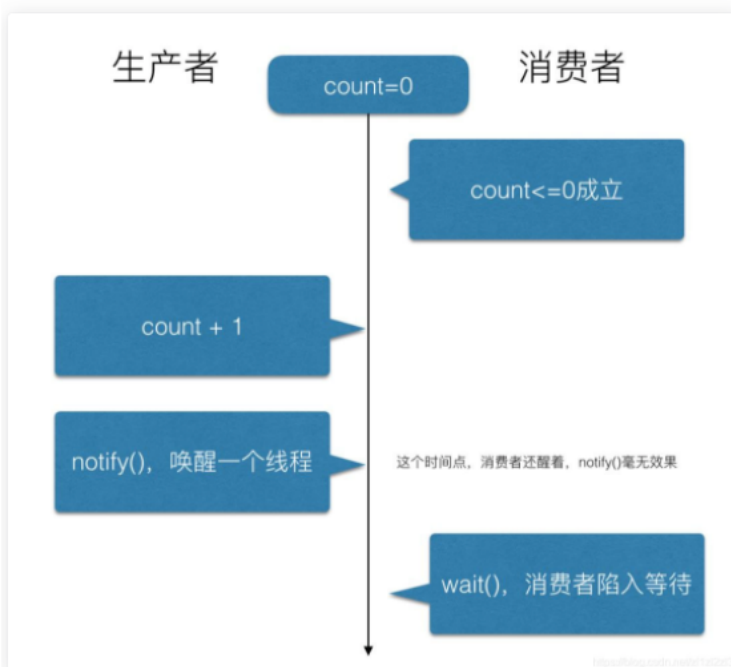
生产者是两个步骤：

1. count+1;
2. notify();

消费者也是两个步骤：

1. 检查count值；
2. 睡眠或者减一；

万一这些步骤混杂在一起呢？比如说，初始的时候count等于0，这个时候消费者检查count的值，发现count小于等于0的条件成立；就在这个时候，发生了上下文切换，生产者进来了，噼里啪啦一顿操作，把两个步骤都执行完了，也就是发出了通知，准备唤醒一个线程。这个时候消费者刚决定睡觉，还没睡呢，所以这个通知就会被丢掉。紧接着，消费者就睡过去了.....



这就是所谓的lost wake up问题。

现在我们应该就能够看到，问题的根源在于，消费者在检查count到调用wait()之间，count就可能被改掉了。



这就是一种很常见的竞态条件。

很自然的想法是，让消费者和生产者竞争一把锁，竞争到了的，才能够修改count的值。

于是生产者的代码是：

```
tryLock()  
count+1  
  
notify()  
releaseLock()
```

消费者的代码是：

```
  
tryLock()  
while(count <= 0)  
    wait()  
  
count-1  
releaseLock()  

```

Java强制我们的wait()/notify()调用必须要在一个同步块中，就是不想让我们在不经意间出现这种lost wake up问题。

```
1 public class RunA implements Runnable {
2
3     private Object lock;
4
5     public RunA(Object lock) {
6         this.lock = lock;
7     }
8
9     @Override
10    public void run() {
11
12        synchronized (lock){
13            try {
14                System.out.println("A begin");
15                // lock.wait(); // 永远等待着，不会执行下去
16                lock.wait(2000); // 等待了 2 秒之后，继续执行下去
17                System.out.println("A end");
18            } catch (InterruptedException e) {
19                e.printStackTrace();
20            }
21        }
22    }
23 }
```