## Java线程之间的协作（同步操作）

线程之间协同工作共同解决一个问题，在互斥的基础上

### wait()和 notifyAll()

如前面所说，当线程调用wait()时线程会释放锁，知道对该线程调用notify（）或notifyAll()才能将其唤醒，因此wait（）提供了一种在任务之间对活动同步的方式。

wait()、notify()、notifyAll()只能在同步控制方法或同步控制块中调用，如果在一般方法中调用虽然能编译但是在运行时会抛出异常

因为当线程调用wait()的时候会释放锁，那么在wait（）期间该线程对象中的其他synchronized可能进行调用，比如下面的Demo:[点击打开链接](https://github.com/xiaoyuzdy/learnJava/blob/master/Java Thinking002/src/Number_2103/P704.java" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

能被notifyAll()唤醒的线程与底层锁有关：[点击打开链接](http://blog.csdn.net/fuckluy/article/details/50825765" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

**注意：**始终应该使用while循环模式调用wait方法，永远不要在循环之外调用wait方法，循环会在等待之前和之后测试条件，一般情况下应该优先使用notifyAll（）方法而不是notify（），因为notifyAll可能也会唤醒其他一些线程，但这并不会影响程序的正确性

### 生产者与消费者问题

现在假设有一个餐馆，里面有一个厨师，有一个服务员，和一个容器，该容器最大只能存放10块肉的存量，每次服务员从容器中拿走一些肉，如果容器没中没有肉了，则服务员线程挂起，当容器中有足够的肉是唤醒服务员线程，如果容器中有足够的肉厨师线程挂起，当容器中肉量不足时厨师现场唤醒

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53170538" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53170538" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. //厨师类
2. **class** Chef **implements** Runnable {
3. **private** Restaurant restaurant;
5. **public** Chef(Restaurant restaurant) {
6. **this**.restaurant = restaurant;
7. }
9. @Override
10. **public** **void** run() {
11. **try** {
12. **while** (!Thread.interrupted()) {
14. **synchronized** (**this**) {
15. //如果容器中肉过多 则挂起厨师线程
16. **while** (restaurant.count >= restaurant.N - 1) {
17. wait();
18. }
19. }
21. System.out.println("厨师制作了一块肉，容器中还有 " + restaurant.count++ + " 块肉");
23. //获取restaurant.person的锁进行同步控制
24. **synchronized** (restaurant.person) {
25. restaurant.person.notifyAll();//唤醒服务员
26. }
28. }
29. } **catch** (InterruptedException e) {
30. System.out.println("厨师线程中断");
31. }
33. }
35. }
37. // 服务员类
38. **class** WaitPerson **implements** Runnable {
39. **private** Restaurant restaurant;
41. **public** WaitPerson(Restaurant restaurant) {
42. **this**.restaurant = restaurant;
43. }
45. @Override
46. **public** **void** run() {
47. **try** {
48. **while** (!Thread.interrupted()) {
49. //如果容器中肉的数量少于1则挂起服务员线程
50. **synchronized** (**this**) {
51. **while** (restaurant.count <= 1)
52. wait();
53. }
55. System.out.println("服务员拿走了一块肉，容器中还有 " + restaurant.count-- + " 块");
56. //获取restaurant.chef的锁进行同步控制
57. **synchronized** (restaurant.chef) {
58. restaurant.chef.notifyAll();//唤醒厨师类
59. }
61. }
63. } **catch** (InterruptedException e) {
64. System.out.println("服务员线程被中断");
65. }
66. }
67. }
69. **public** **class** Restaurant {
70. **final** **static** **int** N = 10;// 容器的容量
71. **int** count = 0;// 计数器
72. // 厨师
73. Chef chef = **new** Chef(**this**);
74. // 服务员
75. WaitPerson person = **new** WaitPerson(**this**);
76. ExecutorService service = Executors.newCachedThreadPool();
77. **public** Restaurant() {
78. service.execute(chef);
79. service.execute(person);
80. }
82. **public** **static** **void** main(String[] args) **throws** InterruptedException {
83. **new** Restaurant();
84. //3s后退出
85. TimeUnit.SECONDS.sleep(3);
86. System.exit(0);
87. }
89. }

Java中的**同步队列**---->BlockingQueue，可用于解决任务协作问题，**同步队列在任何时刻都只允许一个任务插入或移除元素**，如果队列为空则挂起消费者线程，其方式与wait( )和notifyAll( )相比更简单且更可靠，使用BlockingQueue改写上面的代码：[点击打开链接](https://github.com/xiaoyuzdy/learnJava/blob/master/Java Thinking002/src/Number_2103/P709.java" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

### 死锁（详细见总结（一））

死锁的四个必要条件：

1、互斥条件，资源只能被一个进程使用

2、占有和等待条件，得到某个资源的进程可以再请求新的资源且不释放已经占有的资源

3、不可抢占，已经分配给某个进程的资源不能被其他进程抢占，只能自己释放

4、**循环等待**（这个条件易被破坏而不产生死锁），有两个或两个以上进程形成环路

**经典的死锁问题，**哲学家就餐问题，问题描述：一群哲学家围着一张桌子（**循环等待**）吃面，他们将花费一定的时间思考，一定的时间吃面，他们中间放着一根筷子，每当他们就餐时就必须同时得到两边的筷子（**互斥**），且别的人不能抢他人的筷子（**不可抢占**），如果一个哲学家左边或右边已经有人在使用筷子，那么这个哲学家就必须等待且不释放手中的筷子（**占有和等待**），**Demo:**

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53170538" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53170538" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. /\*\*
2. \* 筷子类
3. \*
4. \* @author he
5. \*
6. \*/
7. **class** Chopstick {
8. **private** **boolean** taken = **false**;
10. // 如果被筷子占用则挂起线程
11. **public** **synchronized** **void** take() **throws** InterruptedException {
12. **while** (taken)
13. wait();
14. taken = **true**;
15. }
17. // 释放筷子，唤醒线程
18. **public** **synchronized** **void** drop() **throws** InterruptedException {
19. taken = **false**;
20. notifyAll();
21. }
23. }
25. **class** Philosopher **implements** Runnable {
27. **private** Chopstick left;
28. **private** Chopstick right;
29. **private** **final** **int** id;
30. **private** **final** **int** ponderFactor;// 模拟思考需要时间的时间因子，用于产生不同范围的随机时间
31. **private** Random random = **new** Random(47);
33. // 该方法用于模拟哲学家思考的时间、吃面的时间
34. **private** **void** pause() **throws** InterruptedException {
35. **if** (ponderFactor == 0)
36. **return**;
37. TimeUnit.MILLISECONDS.sleep(random.nextInt(ponderFactor \* 300));
38. }
40. **public** Philosopher(Chopstick left, Chopstick right, **int** id, **int** ponderFactor) {
41. **this**.left = left;
42. **this**.right = right;
43. **this**.id = id;
44. **this**.ponderFactor = ponderFactor;
45. }
47. @Override
48. **public** **void** run() {
49. **try** {
50. **while** (!Thread.interrupted()) {
51. System.out.println(**this** + " " + "thinking");
52. pause();
53. System.out.println(**this** + " 获取右边的筷子");
54. right.take();
55. System.out.println(**this** + " 获取左边的筷子");
56. left.take();
57. System.out.println(**this** + " 正在吃面");
58. pause();
59. right.drop();
60. left.drop();
61. }
63. } **catch** (InterruptedException e) {
64. // TODO: handle exception
65. }
66. }
68. @Override
69. **public** String toString() {
71. **return** "Philosopher  " + id;
72. }
74. }
76. **public** **class** Test {
77. **public** **static** **void** main(String[] args) **throws** Exception {
78. **int** size = 7;
79. **int** ponder = 0;// 设置为0用于尽快的产生死锁
81. ExecutorService service = Executors.newCachedThreadPool();
82. Chopstick chopstick[] = **new** Chopstick[size];
83. **for** (**int** i = 0; i < size; i++)
84. chopstick[i] = **new** Chopstick();
86. /\*\*
87. \* 因为每个哲学家都是先拿右边的筷子，所以可能会发生每个人都拿着右边的筷子并等待左边的筷子导致死锁的发生
88. \*/
89. **for** (**int** i = 0; i < size; i++)
90. service.execute(**new** Philosopher(chopstick[i], chopstick[(i + 1) % size], i, ponder));
92. /\*\*
93. \* 如果最后一个人先拿左边的筷子后拿右边的筷子则可以避免死锁的发生（破环了循环等待条件）
94. \*/
95. //      for (int i = 0; i < size; i++) {
96. //          if (i < size - 1)
97. //              service.execute(new Philosopher(chopstick[i], chopstick[i + 1], i, ponder));
98. //          else
99. //              service.execute(new Philosopher(chopstick[0], chopstick[i], i, ponder));
100. //      }
102. service.shutdown();
104. }
106. }

## Java中的并发工具

这里只是简单的介绍，详细的学习可参考《Java编程思想》（强推，我的启蒙书）并发那一章的新类库的构建那一部分，还有就是《Java并发编程实践》（不看过不知道怎么样）

**并发工具优先于wait和notify**，java.util.concurrent中的高级并发工具分为三类：

1、ExecutorFramework，利用线程池开启任务

2、Concurrent Collection （并发集合），为标准的集合接口List、Queue和Map提供了高性能的并发实现（比如：ConcurrentHashMapm，BlockingQueue），具体实现需要自己管理，大多数ExecutorService实现都是用BlockingQueue

3、Syncgronizer（同步器）是一些使线程能够等待另一个线程的对象，允许他们协调动作，比如：CountDownLatch、CyclicBarrier

### CountDownLatch

CountDownLatch：用来同步一个或多个任务，可在构造器中传如一个数值作为计数器，**强制它们等待**由其他任务执行的一组操作完成，这就意味着，如果**计数器的值不为0**的话，任何在这个对象上调用**a**wait( )方法**都将阻塞**，当然该对象可以通过调用countDown（）方法减少这个计数值，CountDownLatch的**计数器的值是不能被重置**的,只能出发一次

**适用情况：**将一个程序分为n个相互独立的可解决的任务，每当一个任务完成时调用countDown()方法将计数器减1，在通过一个线程不断的检测程序的计数器是否为0，如果为0则**a**wait（）成功(用于证明上面所说的如果计数器值不为0则强制等待)，**Demo:**[点击打开链接](https://github.com/xiaoyuzdy/learnJava/blob/master/Java Thinking002/src/Number_2104/CountDownLatchDemo.java" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

### ****CyclicBarrier****

CyclicBarrier与CountDownLatch很相似，但是CountDownLatch只能被触发一次而**CyclicBarrier可以被触发多次**

**适用情况：**创建一组任务，它们并发地执行工作，然后在进行下一步骤之前等待，直至所有任务都完成，CyclicBarrier构造器中有一个构造器有两个参数，一个参数指定线程数，第二个参数是一个Runnable，它将会在CyclicBarrier的数值到达0时自动运行（即，在一组线程中的最后一个线程到达之后（但在释放所有线程之前）），这个Runnable相当于一个屏障（通过匿名内部类创建）。

比如下面的示例，有一组线程有线程A,B,C每个线程分别代表一名运动员，当CyclicBarrier中的数值为0时则调用Runnable去打印他们跑的距离，当到达终点后则停止所有线程，Demo:[点击打开链接](https://github.com/xiaoyuzdy/learnJava/blob/master/Java Thinking002/src/Number_2104/CyclicBarrierDemo.java" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

## **线程安全性的几种级别：**

1、**不可变的**------这个类的实例是不变的，所以不需要外部的同步，比如String、Long和BigInteger

2、**无条件的线程安全**---这个类的实例是可变的，但是这个类有着足够的内部同步，所以它的实例可以被并发地使用，无需任何外部同步，比如Random和CuncurrentHashMap

3、**有条件的线程安全**------类的部分方法需要外部同步才能进性安全的并发，其他的与无条件线程安全一样，比如Collections.synchronized包装返回的集合，他们的iterator要求外部同步

4、**非线程安全**-----类的实例是可变的，没有什么内部同步，为了并发使用它们必须通过外部同步方法包裹它们实例的方法调用，比如ArrayList和HashMap

5、**线程对立的**----类不能安全地被多个线程并发使用，即使所有的方法调用都被外部同步包围（Java平台的类库中线程对立的类或方法非常少）