## Java中共享资源的竞争（互斥）

**未完待续。。。。。**

共享资源竞争需要解决的问题是，采取某种方式防止同一个共享资源在同一时刻被多个任务访问

### 1、同步控制（synchronized）和显示锁（Lock对象）

解决资源竞争的**第一种方式**是采用锁的形式，即在一个线程访问该资源时会在其上加上锁，使其他任务在该资源解锁前无法访问，这种机制采用的就是互斥量（见并行总结第一部分），在java中给一个资源上锁有两种形式，一个是使用synchronized关键字，一个是使用显示的Lock

**synchronized：**如果某个任务处于一个标记为synchronized的方法调用中，那么在这个线程从该方法返回之前，其他所有要调用类中任何标记为synchronized方法的线程都会被**阻塞**，如下面的Demo中Number类中的next（）方法会不停的产生偶数，这时候有两个线程调用该方法，如果不加synchronized，总会产生非偶数，这是由于Number的next（）方法在执行第二次i++的时候恰好该方法被线程B调用而导致出错，如果加上synchronized关键字则不会出现奇数的情况

**几点说明：**

1、使用并发时，域要设置为private，否则synchronized不能防止其他任务之间访问域

2、synchronized static方法可以在类的范围内防止对static数据的并发访问

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53099102" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53099102" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** Number {
2. **private** **static** **int** i;//域要设置为private
3. //产生偶数
4. **public**  **static** **int** next() {
5. i++;
6. i++;
7. **return** i;
8. }
10. /\*\*
11. \* 加上 synchronized关键字 则当一个线程调用该方法时另一个线程被阻塞
12. \*/
13. //  public  synchronized static int next() {
14. //      i++;
15. //      i++;
16. //      return i;
17. //  }


21. }
23. **class** A **implements** Runnable {
24. **private** **boolean** ok = **false**;
26. @Override
27. **public** **void** run() {
28. **int** t = 0;
29. **while** (**true**) {
30. t = Number.next();
31. //如果获取的不是偶数就停止
32. **if** (t % 2 != 0) {
33. System.out.println(t + "  error");
34. ok = **true**;
35. System.exit(0);
36. }
37. }
38. }
40. **public** **boolean** isOk() {
41. **return** ok;
42. }
44. }
46. **class** B **implements** Runnable {
47. **private** A a = **new** A();
49. @Override
50. **public** **void** run() {
51. **while** (**true**) {
52. Number.next();
53. **if** (a.isOk())
54. System.exit(0);
55. }
56. }
58. }
60. **public** **class** Test {
61. **public** **static** **void** main(String[] args) {
62. ExecutorService service = Executors.newCachedThreadPool();
63. service.execute(**new** A());
64. service.execute(**new** B());
65. service.shutdown();
67. /\*\*
68. \* 结果： xxxx(一个奇数)  error
69. \*/
70. }
71. }

有时只希望防止多个线程同时访问方法内部的部分代码而不是防止访问整个方法，则可以使用同步控制块（或者称为临界区），使用synchronized关键字建立，这样可以提高时间性能，在进入同步控制块代码之前必须得到syncObject对象的锁，如果其他线程已经得到这个锁，那么就得等到锁被释放以后才能进入临界区

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53099102" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53099102" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. //synObject  指定对象
2. **synchronized**(synObject){
3. //code
4. }

比如上面例子中的 next（）可以改为

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53099102" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53099102" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. /\*\*
2. \* 同步控制块,注意因为是静态方法所以使用  类名.class   如果是非静态方法使用this
3. \* @return
4. \*/
5. **public**  **static** **int** next() {
6. **synchronized**(Number.**class**){
7. i++;
8. i++;
9. }
10. **return** i;
11. }

在并发线程中使用一个不是线程安全的类(采用组合的形式)，比如下面的Demo中，Pair类是一个普通的类，现在需要将Pair类应用到多线程中，采用组合的方式，将普通的Pair类加到线程类M1和M2中，通过同步控制方法完成Pair类中x和y的增加

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53099102" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53099102" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. /\*\*
2. \*
3. \* 未实现同步控制的不安全类
4. \*
5. \* @author he
6. \*
7. \*/
8. **class** Pair {
9. **private**  **int** x, y;
11. **public** Pair() {
12. **this**(0, 0);
13. }
15. **public** Pair(**int** x, **int** y) {
16. **this**.x = x;
17. **this**.y = y;
18. }
20. **public** **int** getX() {
21. **return** x;
22. }
24. **public** **int** getY() {
25. **return** y;
26. }
28. **public** **void** incrementX() {
29. x++;
30. }
32. **public** **void** incrementY() {
33. y++;
34. }
36. // 内部异常类，当x!=y时抛出该异常
37. **class** PairValuesNotEqualException **extends** RuntimeException {
38. **public** PairValuesNotEqualException() {
39. **super**("Pair values not equals" + Pair.**this**);
40. }
42. }
44. //x和y不等抛出异常
45. **public** **void** checkPair() {
46. **if** (x != y)
47. **throw** **new** PairValuesNotEqualException();
48. }
50. }
52. **abstract** **class** PariManager {
53. Pair p = **new** Pair();//组合
55. **public** **synchronized** Pair getPair() {
56. **return** **new** Pair(p.getX(), p.getY());
57. }
59. // 增加Pair的x和y
60. **public** **abstract** **void** increment();
61. }
63. /\*\*
64. \* 使用同步控制方法
65. \* @author he
66. \*
67. \*/
68. **class** PariManager1 **extends** PariManager **implements** Runnable {
70. @Override
71. **public** **synchronized** **void** increment() {
72. p.incrementX();
73. p.incrementY();
74. }
76. @Override
77. **public** **void** run() {
78. **while** (**true**) {
79. increment();
80. }
81. }
82. }
84. /\*\*
85. \* 使用同步控制块
86. \* @author he
87. \*
88. \*/
89. **class** PairManager2 **extends** PariManager **implements** Runnable {
91. @Override
92. **public** **void** increment() {
93. **synchronized** (**this**) {
94. p.incrementX();
95. p.incrementY();
96. }
97. }
99. @Override
100. **public** **void** run() {
101. **while** (**true**) {
102. increment();
103. }
104. }
106. }
108. /\*\*
109. \* 检测Pair中的x和y是否会出现不等情况的类
110. \*
111. \* @author he
112. \*
113. \*/
114. **class** Check **implements** Runnable {
115. **private** PariManager manager;
117. **public** Check(PariManager manager) {
118. **this**.manager = manager;
119. }
121. @Override
122. **public** **void** run() {
123. **while**(**true**){
124. //检测Pair中的x和y是否相等，不等则抛出异常
125. manager.getPair().checkPair();
126. }
128. }
129. }
131. **public** **class** Test {
132. **public** **static** **void** main(String[] args) {
133. ExecutorService service=Executors.newCachedThreadPool();
134. PariManager1 p1=**new** PariManager1();
135. PairManager2 p2=**new** PairManager2();
136. Check check1=**new** Check(p1);
137. Check check2=**new** Check(p2);
138. service.execute(p1);
139. service.execute(p2);
140. service.execute(check1);
141. service.execute(check2);
143. **try** {
144. TimeUnit.MILLISECONDS.sleep(2000);//2s后退出程序
145. } **catch** (InterruptedException e) {
146. // TODO Auto-generated catch block
147. e.printStackTrace();
148. }
149. System.exit(0);
151. }
153. }

**显示Lock：**Java.util.concurrent.locks是Java中显式的互斥机制，Lock对象必须显式地创建、锁定和释放，更加灵活

对于第一个Demo，只需将Number类改为下面这样即可

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53099102" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53099102" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** Number {
2. **private** **static** **int** i;
3. **private** **static** Lock lock = **new** ReentrantLock();//互斥锁
5. // 产生偶数
6. **public** **static** **int** next() {
7. lock.lock();// 加锁
9. **try** {
10. i++;
11. i++;
12. **return** i;
13. } **finally** {
14. lock.unlock();// 解锁
15. }
17. }
18. }

解决资源竞争的第二种方式是根除对变量的共享，线程本地储存是一种自动化机制，可以为使用相同变量的每个不同的线程都创建不同的储存，比如第一个Demo可以进行如下修改，不需要使用同步控制，因为ThreadLocal保证不会出现竞争条件

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53099102" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53099102" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** Number {
2. /\*\*
3. \* ThreadLocal 对象通常当作静态域储存
4. \*/
5. **private** **static** ThreadLocal<Integer> local = **new** ThreadLocal<Integer>() {
6. //设置初始值，防止未调用set直接调用get时出空指针异常
7. **protected** Integer initialValue() {
8. **return** 0;
9. };
10. };
12. /\*\*
13. \* local.set() 会将参数插入到为其线程储存的对象中
14. \*/
15. **public** **static** **void** set() {
16. //储存偶数
17. local.set(local.get()+1);
18. local.set(local.get()+1);
19. }

22. /\*\*
23. \* local.get() 会返回其线程相关联的对象的副本
24. \* @return
25. \*/
26. **public** **static** **int** get() {
27. set();
28. **return** local.get();
29. }
31. }
33. **class** A **implements** Runnable {
34. @Override
35. **public** **void** run() {
36. **int** t = 0;
37. **while** (**true**) {
38. t = Number.get();
39. // 如果获取的不是偶数就停止
40. **if** (t % 2 != 0) {
41. System.out.println(t+" error");
42. System.exit(0);
43. }
44. }
45. }
46. }
48. **class** B **implements** Runnable {
49. @Override
50. **public** **void** run() {
51. **while** (**true**) {
52. Number.get();
53. }
54. }
55. }
57. **public** **class** Test {
58. **public** **static** **void** main(String[] args) **throws** InterruptedException {
59. ExecutorService service = Executors.newCachedThreadPool();
60. service.execute(**new** A());
61. service.execute(**new** B());
62. service.shutdown();
63. //5s 秒后自动退出
64. TimeUnit.MILLISECONDS.sleep(5000);
65. System.exit(0);
67. }
69. }

可以简单的看一下set和get方法的源码

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53099102" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53099102" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **public** **void** set(T value) {
2. Thread t = Thread.currentThread();
3. ThreadLocalMap map = getMap(t);
4. **if** (map != **null**)
5. map.set(**this**, value);
6. **else**
7. createMap(t, value);
8. }
9. **public** T get() {
10. Thread t = Thread.currentThread();
11. ThreadLocalMap map = getMap(t);
12. **if** (map != **null**) {
13. ThreadLocalMap.Entry e = map.getEntry(**this**);
14. **if** (e != **null**) {
15. @SuppressWarnings("unchecked")
16. T result = (T)e.value;
17. **return** result;
18. }
19. }
20. **return** setInitialValue();
21. }

可以看到其实set的内部使用的是一个以线程为键的Map，每次调用的时候会先获得当前线程，然后以当前线程为Key将值保存到Map中，get方法也是同理，因此在Demo中一个线程对值的修改对另一个线程是没有影响的，是两个不同的副本

### ****2、Java中的原子性，可视性（原子操作需要进行同步控制）****

**原子操作：**不能被线程调度机制中断的操作，一旦操作开始，那么他**一定可以**在可能发生的上下文切换之前（这里可以理解为线程切换）执行完毕，在Java中除long和double之外的所有基本类型的**简单操作**（诸如赋值和返回操作，递增不是原子性操作，如i++）**具有原子性**，long和double不具备原子性的原因是long和double是八字节，64位，但是JVM可以将64位的读取和写入当作两个分离的32位操作执行，因此在两个32位操作之间可能发生上下文切换，导致不正确性，long和double要想获得**简单的原子性**（赋值和返回操作）需要使用**volatile**关键字，原子操作是由线程机制保证其不可中断的，专家级程序员可以利用此点编写无锁的代码，但是不建议这么做

**可视性**：在多批处理器系统中，对一个任务作出修改，是原子性的，但是对其他任务可能是不可视的（比如，修改只是暂时性的储存在本地处理器的缓存中，因此其他任务读取的值还是旧的值，这样就产生了多个任务读取值具有不同视图的问题）

**同步机制（synchronized）保证了可视性**，即保证了当一个任务被修改后其他任务获取的视图是相同的视图

**volatile关键字保证了可视性，**使用volatile修饰的域会被立即写入到主存中，读取操作都发生在主存中如果一个任务所做的操作要对所有任务可视则需要用volatile关键字，否则不需要加一个具有原子操作的未被volatile修饰的域不会刷新到主存中去，因此其他读取该域的任务也不会看到新值，如果要该域被多个任务同时访问，要加上volatile关键字保证可视性，或使用同步控（synchronized）多批处理器系统（详细了解可看《现代操作系统》的第八章），简单了解一下可以看我的笔记：[点击打开链接](http://note.youdao.com/noteshare?id=1ee5bab7d719dd0176445d986955ef74" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

**\*\*\*\***在并发中应该优先选择synchronized（更安全）而不是volatile和原子性，比如下面的Demo中原子操作并没有保证程序的正确性

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53099102" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53099102" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **public** **class** Test **implements** Runnable {
3. **private** **volatile** **int** i=0;
5. **public**  **int** getValue(){
6. **return** i;             //原子操作，但是缺少同步使得其数值可以在处于不稳定的中间状态时被读取
7. }
9. //使用同步控制程序的正确性得到保障
11. //  public  synchronized int getValue(){
12. //      return i;             //原子操作
13. //  }
15. //同步控制，产生偶数
16. **public** **synchronized** **void** evenIncrement(){
17. i++;
18. i++;
19. }
21. @Override
22. **public** **void** run() {
23. **while**(**true**)
24. evenIncrement();
25. }
27. **public** **static** **void** main(String[] args) {
28. ExecutorService service = Executors.newCachedThreadPool();
29. Test t=**new** Test();
30. service.execute(t);
31. **while**(**true**){
32. **int** val=t.getValue();
33. //如果得到奇数就停止
34. **if**(val%2!=0){
35. System.out.println(val);
36. System.exit(0);
37. }
38. }
40. /\*\*
41. \* 结果： xxxx(一个奇数)
42. \*/
43. }
44. }

### ****3、线程的阻塞（进入阻塞的原因，见总结（一））****

有的时候需要终止处于阻塞状态的任务--------->只能中断可中断阻塞

当在线程上调用interrupt（）时，**中断发生的唯一时刻**是在任务要进入到阻塞操作中，或者已经在阻塞操作内部时。

1、Thread.interrupted( )，提供了离开run( )循环而**不抛出异常**的方式。

2、如果通过Executor启动线程，那么Executor上的shutdownNow()会终止**所有的**线程，而如果想终止**单一的线程**则需要使用submit（）方法启动单个线程而不是executor()，同时在返回的Future对象上调用cancel（）方法终止单个线程

**[java]** [view plain](http://blog.csdn.net/fuckluy/article/details/53099102" \o "view plain" \t "http://blog.csdn.net/fuckluy/article/details/_blank) [copy](http://blog.csdn.net/fuckluy/article/details/53099102" \o "copy" \t "http://blog.csdn.net/fuckluy/article/details/_blank)

1. **class** A **implements** Runnable{
2. @Override
3. **public** **void** run() {
4. **try** {
5. TimeUnit.MILLISECONDS.sleep(1000);
6. } **catch** (InterruptedException e) {
7. System.out.println("线程A被终止");
8. }
9. }
10. }

13. **class** B **implements** Runnable{
14. @Override
15. **public** **void** run() {
16. **try** {
17. TimeUnit.MILLISECONDS.sleep(1000);
18. } **catch** (InterruptedException e) {
19. System.out.println("线程B被终止");
20. }
21. }
22. }


26. **public** **class** Test {
27. **public** **static** **void** main(String[] args) **throws** InterruptedException {

30. /\*\*
31. \* 使用Thread.interrupt()终止线程A
32. \*/
33. Thread t=**new** Thread(**new** A());
34. t.start();
35. t.interrupt();//终止线程，抛出异常
36. //      t.interrupted();//终止线程,且不抛出异常

39. /\*\*
40. \* 使用execute.shutdownNow()终止所有启动的线程
41. \*/
42. ExecutorService service = Executors.newCachedThreadPool();
43. //      service.execute(new A());
44. //      service.execute(new B());
45. //      service.shutdownNow();//终止线程，会抛出异常，因为Executor发送的是interrupt()调用给它启动的所有线程

48. /\*\*
49. \* 通过submit()启动线程，并通过Future中的cancel终止单个线程
50. \*
51. \*/
53. Future<?> future=service.submit(**new** B());
54. future.cancel(**true**);//终止线程，试了很多次有时候会抛出异常
55. System.out.println(future.isCancelled());//如果在任务正常完成前将其取消，则返回 true。
56. TimeUnit.SECONDS.sleep(2);
57. System.exit(0);
59. }
61. }