# 1. JUC简介

Java 5.0提供了java.util.concurrent（简称JUC）包，此包中提供了在并发编程时常用的实用工具类，方便用户使用多线程。但是应该注意的是要合理使用多线程，因为如果使用不当，多线程的开销是大于单线程的，如线程的创建、同步和销毁等。

# 2. volatile关键字：内存可见性

下面以一个例子介绍volatile关键字的用处。

我们先创建一个MyThread线程类，其中维护了一个flag属性当做标记。在线程要执行的run()方法中，会将flag属性改为true，并打印提示。

|  |
| --- |
| **package** juc.demo; **public class** MyThread **implements** Runnable {  **private boolean flag** = **false**; // falg属性  @Override  **public void** run() {  **try** {  Thread.*sleep*(50);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *// 以上使用sleep 50毫秒模拟执行场景。* **flag** = **true**;  System.***out***.println(**"改变flag属性为："** + **flag**);  }   *// 对flag的getter和setter方法。* **public boolean** isFlag() {  **return flag**;  }  **public void** setFlag(**boolean** flag) {  **this**.**flag** = flag;  } } |

在主类中，创建上述线程并执行，主线程中用while(true)不断观察上述线程对象中的flag属性（即flag就是线程间共享了），若flag属性为真，就打印提示，然后退出循环结束程序。

|  |
| --- |
| **package** juc.demo;  **public class** Demo {  **public static void** main(String[] args) {  MyThread mt = **new** MyThread(); *// 线程对象* **new** Thread(mt).start(); *// 执行* **while** (**true**) {  **if** (mt.isFlag()) {  System.***out***.println(**"----flag属性已为true----"**);  **break**;  }  }  } } |

但上述程序执行时，只会输出“改变flag属性为：true”，且程序不会停止。这说明，MyThread线程已经将flag置为true，且该线程已结束，但是主线程中得到的flag一直是false，因此不断循环，程序无法退出。这是怎么回事呢？

原来，JVM会为每一个执行的线程分配一个独立的缓存，这用于提高效率。在上例中，一开始，共享数据flag（值为false）在主存中（这里的主存指存放原始数据的地方），两个线程都要先读到flag数据，并将flag放在各自线程的缓存中。所以一开始两个线程读到的flag都为false。MyThread读到数据并完成对flag的修改后，还要将此修改操作也保存到主存上，这样flag才最终变为false。那么为啥主线程一直拿到的flag是false呢，难道不会刷新缓存吗？原来是因为底层的while循环效率非常高，主线程根本来不及更新本缓存的数据，因此导致每次都是false，不停循环。

当然，以上前提是主线程在MyThread线程修改flag前读到数据，由于我们在MyThread中设置了sleep 50毫秒，因此一般都是满足该前提的。

实际上产生上述问题就是两个线程间共享数据“不可见”导致的，他们两个线程之间对于共享数据的操作彼此是不可见的。

那么目前我们可使用同步锁来解决这个问题。使用同步锁能够保证每次都刷新缓存。即改变代码为：

|  |
| --- |
| **package** juc.demo;  **public class** Demo {  **public static void** main(String[] args) {  MyThread mt = **new** MyThread(); *// 线程对象* **new** Thread(mt).start(); *// 执行* **while** (**true**) {  **synchronized** (mt) {  **if** (mt.isFlag()) {  System.***out***.println(**"----flag属性已为true----"**);  **break**;  }  }  }  } } |

但是使用同步锁解决问题会导致程序运行的效率降低。

那么我们既不想效率降低太多，又要解决共享变量可见性的问题，就可以使用volatile关键字。volatile关键字可以保证多个线程访问共享数据时，彼此的数据是可见的。

保证所以我们使用volatile关键字就会使多个线程共享数据时，他们的操作是可见的。可以简单理解为，使用volatile关键字后，多个线程就会直接操作“主存”中的共享数据“flag”。我们只要在flag前面加上volatile关键字修饰即可（不用之前的加锁了）：

|  |
| --- |
| **private volatile boolean flag** = **false**; |

虽然使用volatile也会影响性能，但相比加同步锁，效率还是高的。那么我们是否以后都用volatile就行了呢？这是不行的。因为volatile相较于同步锁，有如下区别：

（1）volatile是一种轻量级的同步策略。

（2）volatile不具备“互斥性”关系，而同步锁有互斥性，即一个线程进入了同步锁中，另一个线程无法进入，这就是互斥性，而volatile无互斥性。

（3）不能保证变量状态的“原子性”操作。下一章继续讨论原子性。

但若是两个多个线程都改变某变量，就会发生线程安全问题，就不能使用volatile了。

总结：

内存可见性（Memory Visibility）是指当某个线程正在使用对象状态而另一个线程在同时修改该状态，需要确保当一个线程修改了对象状态后，其他线程能够看到发生的状态变化。

可见性错误是指当读操作与写操作在不同的线程中执行时，我们无法确保执行读操作的线程能适时地看到其他线程写入的值。我们可以通过同步来保证对象被安全地发布，除此之外我们也可以使用一种更加轻量级的 volatile 变量。

Java 提供了一种稍弱的同步机制，即 volatile 变量，用来确保将变量的更新操作通知到其他线程。可以将 volatile 看做一个轻量级的锁，但有不同。

# 3. CAS算法与原子变量

我们先看下面的例子：

MyThread线程中维护了一个number变量，并提供了getNumber()方法，该方法返回“number++”的值。线程输出getNumber()方法修改后的值：

|  |
| --- |
| **package** juc.demo;  **public class** MyThread **implements** Runnable {  **private int number** = 0;   @Override  **public void** run() {  **try** {  Thread.*sleep*(50);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *// 以上使用sleep 50毫秒模拟执行场景。* System.***out***.println(Thread.*currentThread*().getName() + **"："** + **this**.getNumber());  }   **public int** getNumber() {  **return this**.**number**++;  } } |

下面的主类中用10个线程运行上述的MyThread代码，相当于共享了number变量：

|  |
| --- |
| **package** juc.demo;  **public class** Demo {  **public static void** main(String[] args) {  MyThread mt = **new** MyThread(); *// 线程对象* **for** (**int** i = 0; i < 10; i++) {  **new** Thread(mt).start(); *// 执行* }  } } |

运行程序很容易出现安全问题，例如输出如下：

|  |
| --- |
| Thread-1：1  Thread-0：4  Thread-2：3  Thread-4：0  Thread-3：2  Thread-5：5  Thread-6：6  Thread-8：7  Thread-7：8  Thread-9：7 |

产生线程安全的根本原因是因为对number的“number++”操作不是“原子性”的。

为什么说“i++”操作不是原子性的呢？那是因为“i++”操作底层分为“读、改和写”三个步骤。比如“int x = i++”，因为“i++”是“先赋值，后自增”，“i++”的底层必然要保存“原值”和“新值”，然后将“原值”再赋给x。实际上“i++”底层使用了一个临时变量，该临时变量用户保存“原值”，下面的代码展示了底层操作是如何进行的：

|  |
| --- |
| int temp = i; // 用temp保存旧值  i = i + 1; // 将i自增1  x = temp; // 将旧值赋给x |

当然，如果是“i = i++”，那么最后就将temp赋给i，就是：

|  |
| --- |
| int temp = i;  i = i + 1;  i = temp; |

这也就解释了为什么执行“int i = 10; i = i++;”后，结果还是10。当然，如果直接是“i++”，就会只有上面两个执行步骤，显然他们也不是原子性的。

既然不是原子性的，使用volatile也不能保证原子性，所以由于操作的非原子性（原子性就是不可分割的，即我们想要i++是不可分割的，但是并不满足，因为i++会导致读、改和写分割开来），多个线程操作时，也能会操作同一个i值，从而导致上面的相同的输出。那么我们只能使用同步来解决。如果我们给getNumber()方法加上synchronized关键字，就不会有线程安全问题了。

JDK1.5以后，我们可以使用“原子变量”。java.util.concurrent.atomic包下提供了常用的原子变量，例如AtomicBoolean、AtomicLong等类。和之前的包装类使用类似，只是他们保证所进行的操作都是“原子性”的。比如AtomicArray就保证数组中的每个元素都是原子性的和可见性的。

那么原子变量是如何保证原子性的呢？

（1）类中使用的变量都是volatile类型的，保证内存可见性

（2）使用CAS算法保证数据的原子性。

CAS即“Compare And Swap”的缩写，CAS算法是硬件对于并发操作共享数据的支持。CAS的原理：

CAS维护了三个操作数，一个是内存值V，即当前内存中真实存储的值；一个值是预估值A，即一开始时保存的内存中的值，即“旧值”；还有一个值是更新值B，即待更新到内存中的值。

CAS算法就是：当然仅当V==A时，即当前内存中的值等于以前保存的旧值时，才允许将B的值赋给V，即才允许更新，否则什么也不做。

即如果你要替换将内存中的值替换为新值，就需要保证旧值和内存中的现在的值一致。这样保证多线程操作的原子性。因为某个线程想要修改内存中的值，那么只有别的线程没有操作过内存中的值才能更新，因为只要修改内存中的值，由于CAS算法，该线程就需要重新读取内存中的值，只有值和旧值一样才能改变。

CAS的效率还是比同步锁高的。我们现在改用原子变量改进MyThread类即可：

|  |
| --- |
| **package** juc.demo;  **import** java.util.concurrent.atomic.AtomicInteger;  **public class** MyThread **implements** Runnable {  **private** AtomicInteger **number** = **new** AtomicInteger(0);   @Override  **public void** run() {  **try** {  Thread.*sleep*(50);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *// 以上使用sleep 50毫秒模拟执行场景。* System.***out***.println(Thread.*currentThread*().getName() + **"："** + **this**.getNumber());  }   **public int** getNumber() {  **return this**.**number**.getAndIncrement();  } } |

这样就搞定了。上述调用的方法的getAndIncrement()，就是先获取再自增的意思，当然也可使用自减、设置值等。其中还提供了一个核心的方法boolean compareAndSet(int expect, int update)，即先比较再设置值。返回是否设置成功。

# 4. ConcurrentHashMap

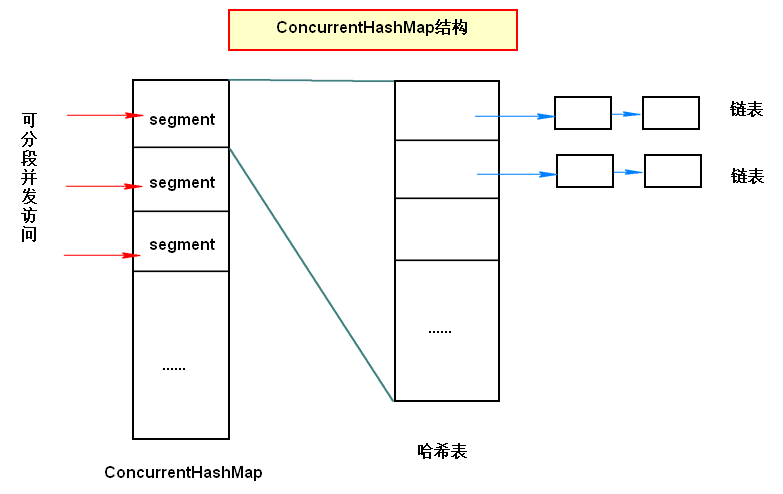
java.util.concurrent包中提供了多种并发容器类来改进同步容器的性能。

ConcurrentHashMap 同步容器类是Java 5 增加的一个线程安全的哈希表。对与多线程的操作，性能介于 HashMap 与 Hashtable 之间。ConcurrentHashMap内部采用“锁分段”机制替代 Hashtable 的独占锁。进而提高性能。

HashMap是线程不安全的，Hashtable是线程安全的，效率低。HashMap和Hashtable的底层都是哈希表，区别在于Hashtable的哈希表是有锁的，锁的方式是锁整个表，这样多线程时就会一个线程访问完后才允许另一个线程访问。这样导致Hashtable效率非常低，因此即使在需要线程安全的地方，我们也不用Hashtable。而且Hashtable在进行一些复合操作时，也会有线程安全问题，例如：如果Hashtable中无A元素，就将A元素添加到Hashtable中。这样的符合操作中间有可能让别的线程抢占从而产生线程安全问题。

ConcurrentHashMap采用锁分段机制：

ConcurrentHashMap中有一个并发级别concurrentLevel，默认为16，即ConcurrentHashMap对象中默认分有16个段（segment），每个段中有默认长度为16的哈希表，然后每个哈希表元素中又存储了一个个的链表。这样ConcurrentHashMap的结构如下图所示：



正如图所示，将哈希表进行了分段，默认有16段哈希表，这样，多线程访问时，允许多线程分别进入多个段进行访问（每个段都是独立的锁），这样提高了并发访问的效率，并且保证了线程安全。并且ConcurrentHashMap提供了一些复合操作的方法。

JDK1.8对ConcurrentHashMap进行了升级，采用了CAS算法，逐步取消了锁分段机制。可以将CAS理解为“无锁算法”，相比使用锁，提高了效率，并且无需分段，直接并行访问且保证线程安全。

之前我们使用线程安全的集合类都是通过Collections工具类中的synchronizedXXX()方法将对应的类转化为线程安全的类，实际上就是把每个方法转化为带有synchronized的同步方法。现在，我们可使用ConcurrentHashMap，基本用法同HashMap。

JUC还提供了其他的容器类：

ConcurrentSkipListMap、 ConcurrentSkipListSet、CopyOnWriteArrayList 和 CopyOnWriteArraySet。ConcurrentSkipListMap 通常优于同步的 TreeMap。当期望的读数和遍历远远大于列表的更新数时， CopyOnWriteArrayList 优于同步的 ArrayList，因为CopyOnWriteArrayList在更新数据时会复制整个列表数据，然后添加数据，造成效率低，但是正是由于这个特点，他不会发生并发修改异常，即我们可用迭代器遍历，遍历的同时再用该集合对象向其中添加数据，但不推荐。

# 5. CountDownLatch闭锁

闭锁，就是允许让一个（或多个）线程等待，直到其他线程全部执行完毕后，再执行这个（或这些）等待的线程。什么意思呢，就是利用闭锁，可让其他线程早于某些线程先执行。

例如想要实现这样的功能：统计10个线程的总执行时间。那么实际上程序总共需要11个线程，其中主线程（即第11个线程）用于统计时间。线程的开始时间是好得到的，因为主线程先运行，直接拿到即可，但是开启了另外10个线程后，连同主线程共11个线程就不好得到其中10个线程的结束时间了，因为无法保证这10个线程在第主线程之前执行完。

但有了CountDownLatch，就能保证这10个线程先执行（即允许主线程等待）。我们需要在线程中维护CountDownLatch变量。实际CountDownLatch对象中就维护了一个变量，用来计数，该计数的总量由等会的主方法创建对象时传递过来。因此我们必须保证每次线程结束后就让计数减1，调用countDown()方法，并且在finally中保证执行。只有计数减到0了，主线程才会被唤醒执行。我们还是用了同步锁，防止run()的线程安全问题。

|  |
| --- |
| **package** juc.demo;  **import** java.util.concurrent.CountDownLatch;  **public class** MyThread **implements** Runnable {  **private** CountDownLatch **countDownLatch**; *// 维护countDownLatch变量   // 构造器传入countDownLatch对象* **public** MyThread(CountDownLatch countDownLatch) {  **this**.**countDownLatch** = countDownLatch;  }   @Override  **public void** run() {  **synchronized** (**this**) {  **try** {  Thread.*sleep*(500);  *// 使用sleep 50毫秒模拟执行场景。* System.***out***.println(Thread.*currentThread*().getName() + **"结束"**);  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  **countDownLatch**.countDown(); *// 将计数减1* }  }  } } |

主线程代码如下：

|  |
| --- |
| **package** juc.demo;  **import** java.time.Duration; **import** java.time.Instant; **import** java.util.concurrent.CountDownLatch;  **public class** Demo {  **public static void** main(String[] args) {  *// 创建countDownLatch对象，给定“计数器”* CountDownLatch countDownLatch = **new** CountDownLatch(10);  MyThread mt = **new** MyThread(countDownLatch);  Instant start = Instant.*now*(); *// 开始时间* **for** (**int** i = 0; i < 10; i++) {  **new** Thread(mt).start();  }  **try** {  *// 用await方法让其他线程先执行* countDownLatch.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  Instant end = Instant.*now*(); *//结束时间* System.***out***.println(**"耗费时间："** + Duration.*between*(start, end).getSeconds());  } } |

可以看到，我们在线程中共用了countDownLatch对象，这是必须的，并且初始的“计数”就是10。在循环开启完线程后，必须调用countDownLatch.await()方法，这样才能使那些线程先执行，只有当countDownLatch里维护的“计数”减到0时，await后面的线程代码才会执行。这就是闭锁的原理和使用。

使用闭锁的操作还能把各个分线程计算的结果汇总成一个最终结果，因为最终结果必须要等待分线程全部计算完毕。

# 6. Callable

Java 5.0 在 java.util.concurrent 提供了一个新的创建执行线程的方式： Callable 接口。

Callable 接口类似于 Runnable，两者都是为那些其实例可能被另一个线程执行的类设计的。但是Callable接口有泛型，实现的call方法有返回结果，返回结果的类型就是反省，并且会抛出异常。

Callable 的执行FutureTask类的支持，以便接收返回的结果。FutureTask是Future接口的实现类（也是Runnable接口的实现类）。

其使用代码如下：

|  |
| --- |
| **package** juc.demo;  **import** java.util.concurrent.Callable;  **public class** MyThread **implements** Callable<Integer> {  @Override  **public** Integer call() **throws** Exception {  *// 计算1 - 50000的总和* **int** sum = 0;  **for** (**int** i = 1; i <= 50000; i++){  sum += i;  System.***out***.println(**"当前sum: "** + sum);  }  **return** sum;  } } |

主类：

|  |
| --- |
| **package** juc.demo;  **import** java.time.Duration; **import** java.time.Instant; **import** java.util.concurrent.CountDownLatch; **import** java.util.concurrent.FutureTask;  **public class** Demo {  **public static void** main(String[] args) **throws** Exception {  *// 创建callable实现类对象* MyThread mt = **new** MyThread();  *// 需要使用FutureTask，传入上述对象* FutureTask future = **new** FutureTask(mt);  *// 再在Thread中传入future，并执行线程* **new** Thread(future).start();  *// 线程执行完后，通过future能拿到执行结果：* System.***out***.println(future.get());  System.***out***.println(**"执行完毕"**);  } } |

可以看出， FutureTask 也可以用作闭锁。在future.get()之后的代码都会等待future拿到结果之后才会执行，但是future.get()之前的代码还是会“我行我素”的与其他线程并行的，因此无法保证。但能保证future.get()之后的代码是后执行的。所以可用于闭锁。

# 7. Lock显式锁（多线程中讲过了）

虽然之前的多线程已经讲过了，但这里再说明一下！

说明：

（1）在Java 5.0之前，协调共享对象的访问时，可以使用的机制只有synchronized和volatile。Java 5.0 后增加了一些新的机制，但并不是一种替代内置锁的方法，而是当内置锁不适用时，作为一种可选择的高级功能。

（2）ReentrantLock 实现了Lock接口，并提供了与synchronized相同的互斥性和内存可见性。但相较于synchronized提供了更高的处理锁的灵活性。

总结：

解决多线程安全问题的方式有：

一、同步代码块；

二、同步方法；

三、同步锁。

# 8. 线程的通信

见13.多线程文档中的“线程的通信”一章。

# 9. ReadWriteLock读写锁

以前，当多线程操作共享数据时，我们都是用给资源上锁的方式（独占锁）来解决线程安全问题。不可避免地，加锁必然会导致程序执行效率的降低，执行时每个相关的线程会独占锁资源。

但有时完全地使用独占锁是没有必要的。出现线程安全问题无非就是不同的线程操作共享数据（读取或者设置，即读取）。例如，多个只读线程获取共享数据就可以共同保持一把锁，没有必要每个读线程都独占一把锁，只要保证“读/写线程”和“写/写线程”之间是互斥的即可，“读/读线程”之前无需互斥，因为不会出错。这样使用锁会提高效率，尤其是在读线程较多时，而完全使用独占锁不利于提高效率。

而读写锁ReadWriteLock就能实现这样的效果：ReadWriteLock维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有写入操作，读取锁可以由多个只读线程同时保持，而写入锁是独占的。因此，对于读取操作占多数的数据结构，使用ReadWriteLock 能提供比独占锁更高的并发性；而对于只读的数据结构，其中包含的不变性可以完全不需要考虑加锁操作。

我们也可以查看API文档，其中这样介绍ReadWriteLock：

|  |
| --- |
| A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive. |

意思和我们讲的一致。在ReadWriteLock中提供了两个方法：readLock()和writeLock()，分别用于获得读锁和写锁，返回的类型都是Lock。ReadWriteLock是接口，我们使用的是它的一个实现类

案例如下：

（1）ReadWriteLockDemo类。其中有两个操作方法，分别是读操作和写操作，操作的是该类的共享数据，且我们使用读写锁解决线程安全问题。注意读操作方法中获得的是读锁，写操作方法中获得的是写锁，且要记得释放锁！

|  |
| --- |
| **public class** ReadWriteLockDemo {  **private int sharedNum** = 0; *// 共享数据   // 使用读写锁。* **private** ReadWriteLock **lock** = **new** ReentrantReadWriteLock();   *// 获得共享数据的操作。采用读锁解决多线程操作的安全问题。* **public int** getSharedNum() {  **lock**.readLock().lock(); *// 上锁(读锁)* **try** {  **return this**.**sharedNum**;  } **finally** {  **lock**.readLock().unlock(); *// 释放锁(读锁)。要匹配，不要释放成写锁* }  }   *// 设置共享数据的操作。采用写锁解决多线程操作的安全问题。* **public void** setSharedNum(**int** sharedNum) {  **lock**.writeLock().lock(); *// 上锁(写锁)* **try** {  **this**.**sharedNum** = sharedNum;  } **finally** {  **lock**.writeLock().unlock(); *// 释放锁(写锁)。同样要匹配！* }  } } |

（2）主类中开启多个线程进行读写操作。例如这里有1个写线程，100个读线程，作为使用读写锁的案例是很合适的。

|  |
| --- |
| **public class** TestReadWriteLock {  **public static void** main(String[] args) {  *// 新建ReadWriteLockDemo对象* ReadWriteLockDemo rwld = **new** ReadWriteLockDemo();  *// 创建线程  // 一个写线程* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **int** num = 50;  rwld.setSharedNum(num);  System.***out***.println(Thread.*currentThread*().getName() + **"设置了共享数据为"** + num);  }  }, **"写线程："**).start();  *// 100个读线程* **for** (**int** i = 1; i <= 100; i++) {  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **int** num = rwld.getSharedNum();  System.***out***.println(Thread.*currentThread*().getName() + **"读取到共享数据为"** + num);  }  }, **"读线程"** + i).start();  }  } } |

上述代码中读线程何时读到50这个数据，取决于写线程何时执行。

# 10. 同步方法案例分析

案例1：两个实例同步方法。

|  |
| --- |
| **package** com.zhang.test;  **public class** ThreadDemo {   **public static void** main(String[] args) {  Number number = **new** Number();  *// 创建两个线程分别调用实例同步方法* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  number.getOne();  }  }).start();   **new** Thread(**new** Runnable() {  @Override  **public void** run() {  number.getTwo();  }  }).start();  } }  **class** Number {  **public synchronized void** getOne() {  System.***out***.println(**"one"**);  }   **public synchronized void** getTwo() {  System.***out***.println(**"two"**);  } } |

上述输出的结果可能输出one two，也可能输出two one，就是看哪个线程先执行。

案例2：在getOne方法中将线程睡眠1秒，其余不变，即：

|  |
| --- |
| **public synchronized void** getOne() {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"one"**); } |

运行结果是程序等待1秒后才会输出one和two。这个应该也有可能第二个线程先执行，那么就先输出two，等1秒输出one。

案例3：在Number中添加一个普通的实例方法（不是同步方法），且新增线程调用该方法：

|  |
| --- |
| **package** com.zhang.test;  **public class** ThreadDemo {   **public static void** main(String[] args) {  Number number = **new** Number();  *// 新建线程* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  number.getOne();  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  number.getTwo();  }  }).start();  *// 新增第三个线程* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  number.getThree();  }  }).start();  } }  **class** Number {  **public synchronized void** getOne() {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"one"**);  }   **public synchronized void** getTwo() {  System.***out***.println(**"two"**);  }   **public void** getThree() {  System.***out***.println(**"three"**);  } } |

执行的结果是先输出three，1秒后输出one和two。应该也有可能是别的情况，因为这个新加实例方法和同步无关。

案例4：把案例三新加的线程去掉，此时新建两个Number对象，分别调用getOne和getTwo对象。

|  |
| --- |
| **package** com.zhang.test;  **public class** ThreadDemo {   **public static void** main(String[] args) {  Number number1 = **new** Number();  Number number2 = **new** Number();  *// 新建线程* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  number1.getOne();  }  }).start();  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  number2.getTwo();  }  }).start();  } }  **class** Number {  **public synchronized void** getOne() {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"one"**);  }   **public synchronized void** getTwo() {  System.***out***.println(**"two"**);  }   **public void** getThree() {  System.***out***.println(**"three"**);  } } |

输出的结果是先输出two，1秒后输出one。这是因为使用两个对象，它们不是使用同一把锁了，因此无需等待另一个线程的锁，直接可以执行。而线程2没有睡眠，因此先输出。

案例5：把getOne改成静态同步方法，且还是使用同一个Number对象。

此时，输出的结果还是先输出two，1秒后输出one。这是因为静态和实例同步方法使用的锁是不一样的，因此还是无需等待另一个线程。

案例6：再把getTwo也改成静态方法，使用一个Number对象，其余不变。

此时输出的结果是等待1秒后输出one和two，也可能先输出two，1秒后输出one。这取决于谁先执行，此时，两个线程使用的又是同一把锁，因为都是静态同步方法。

案例7：改为：getOne是静态同步方法，getTwo是实例同步方法，使用两个Number对象。执行结果是先输出two，1秒后输出one，这时它们使用的锁根本不是同一把锁，无需等待。

案例8：改为：两个都是静态方法，使用两个Number对象。此时输出的结果是等待1秒后输出one和two，也可能先输出two，1秒后输出one。这取决于谁先执行，此时，两个线程使用的又是同一把锁，因为都是静态同步方法。这和是不是使用一个Number对象没有关系，因为是静态的。

上面这些案例都是使用同步方法时常见的情况，总结如下：

（1）分析上述情况的关键点是要弄清同步方法的锁对象是谁，加不加普通方法对同步方法无影响。

（2）所有的实例同步方法用的都是同一把锁——实例对象本身，也就是说如果一个实例对象的实例同步方法获取锁后，该实例对象的其他非静态同步方法必须等待获取锁的方法释放锁后才能获取锁。但是不同实例对象的实例同步方法所采用的锁是不同的（都是各自实例），因此它们之间不会产生等待，这是使用时需要注意的。

（3）所有的静态同步方法用的也是同一把锁——类对象本身（即Class字节码对象）。静态同步方法和实例同步方法使用的锁是不同的，所以静态同步方法与实例同步方法之间是不会有竞态条件的。但是一旦一个静态同步方法获取锁后，该类中的其他的静态同步方法都必须等待该方法释放锁后才能获取锁，这在不同的实例对象间都有效，只要它们是同一个类的实例对象！这是使用静态同步方法需要注意的。

（4）一个对象里面如果有多个synchronized方法，某一个时刻内，只要一个线程去调用其中的一个synchronized方法了，其它的线程都只能等待，换句话说，某一个时刻内，只能有唯一一个线程去访问这些synchronized方法。（前提是使用的锁相同）

# 11. 线程池和线程调度

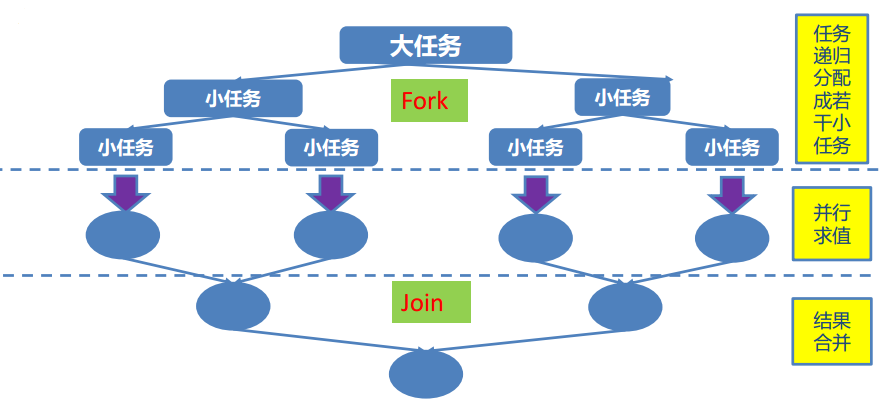
此部分见多线程的文档，已经讲过了，但是这是JDK1.5后JUC新增的内容。

# 12. ForkJoinPool分支/合并框架

ForkJoin框架是在JDK1.7之后出来的。这个在Java8新特性的时候讲过了，但是Java 8又做了改进，因为ForkJoin使用是比较复杂的。

ForkJoin框架就是“分支/合并”框架：就是在必要的情况下，将一个大任务，进行拆分(fork)成若干个小任务（拆到不可再拆时），再将一个个的小任务运算的结果进行 join 汇总。

如图：



Fork/Join框架中也采用了线程池。具体代码可以看看Java8中的讲解。