**1.线程简介**

**1.1 什么是线程?**

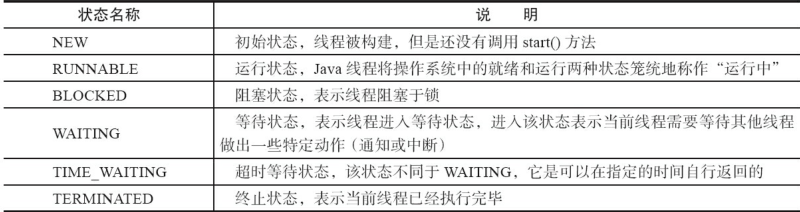
**运行一个程序**,操作系统就会创建一个**进程**,进程是操作系统**资源分配**的最小单元.线程是轻量级进程,一个**进程**可以包含**多个线程**,这些线程都拥有**各自**的计数器、堆栈和局部变量等属性，各线程之间**共享进程的资源**。处理器在线程上高速切换，让我们感觉到线程同时执行。

**1.2为什么要使用多线程**

①更多的处理器核心----单线程只能在一个处理器上使用，如果不采用多线程，即便处理再多也没用，如果是多线程，那么多个线程在多个处理器运行，可以减少处理时间，提高效率

②更快的响应时间----如果一系列复杂操作，要在后台运行，如下订单，包括插入数据，生成订单快照、发邮件通知卖家和记录销售数量等，如果只有一个线程，那么用户的等待时间可能会很长，如果采用多线程进行操作处理，可以减少响应时间。

**1.3线程的状态**



New：线程对象已经创建，但是没有调用start方法

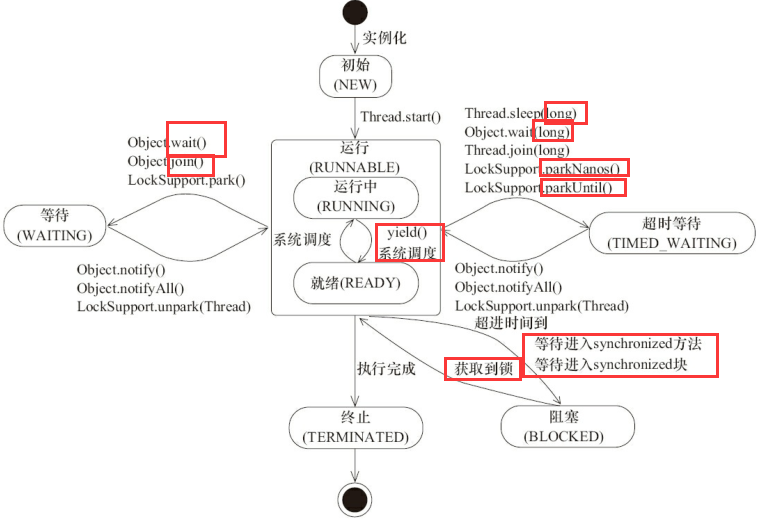
RUNNABLE：运行状态，就绪和运行都称为这个状态

BLOCKED: 锁被占有时的状态，这里指的synchronized，不包括lock

WAITING：调用wait方法后的状态

TIME\_WAITING：调用sleep后的状态，超过时间就会回到运行

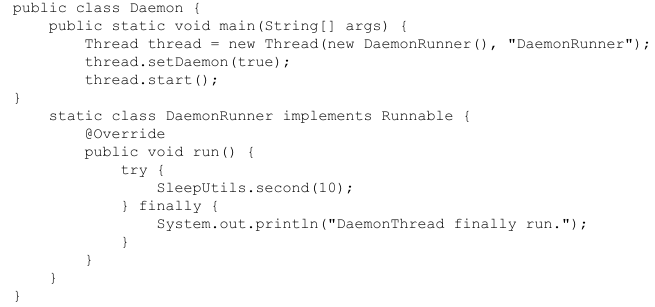
TERMINATED: run方法执行完毕的状态



注意：阻塞状态**指示synchronized关键字获取锁的状态**，但是阻塞在lock接口中的线程状态确实等待状态，Lock接口对于阻塞的实现使用了LockSupport相关方法。Yield方法只会让状态到就绪状态，让同优先级的线程有机会去被系统调度，但不保证，也能yield之后还是本身线程在运行；

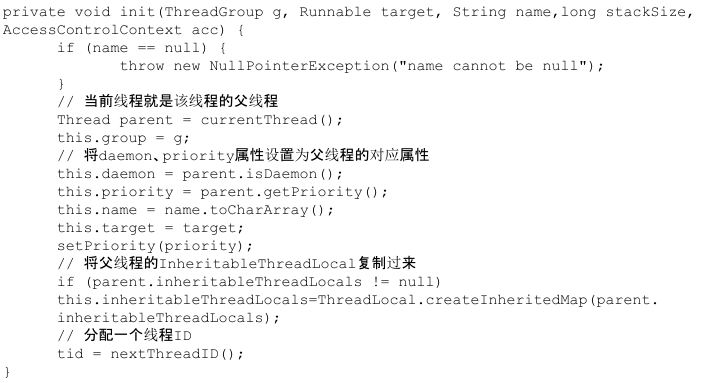
**1.4 Daemon线程**

支持型线程，因为它主要被用作程序中后台调度以及支持性工作。当虚拟机中，非Daemon线程都运行结束之后，jvm会退出。可以通过Thread.setDaemon（true）设置守护线程。但是需要在线程start方法之前，设置。



在main方法执行完后，JVM已经没有非Daemon线程，JVM退出，此时Daemon线程立即终止，不会执行finally块，也就没有输出了。因此，构**建Daemon线程时，不能依靠finally块来确保关闭资源。**

**2.启动和终止线程**



CurrentThread就是你的Thread对象在哪个线程new的，就是哪个线程，即parent，child线程继承了parent的是否Daemon，优先级和加载资源的contextClassLoader以及可继承的TheadLocal，分配一个唯一id标识这个child线程。

**2.1 启动线程**

线程start()方法的含义：当前线程parent 同步告知java虚拟机，只要线程规划器空闲，应立即启动调用start方法的线程。

**2.2理解中断**

中断可以理解为线程的一个**标识位属性**，中断好比其他线程对该线程打了个招呼，其他线程通过调用该**线程的interrupt()**方法对其进行中断操作。

线程通过检查**自身是否被中断**来进行响应，线程通过方法**isInterrupted()**来进行判断是否被中断，也可以调用静态方法**Thread.interrupted()对当前线程的中断标识位进行复位**。如果该线程已经处于**终止状态**，即使该线程被中断过，在调用该线程对象的**isInterrupted()时依旧会返回false。**

一些声明InterruptedException的方法，例如sleep方法，这些方法在抛出InterruptedException之前，Java虚拟机会先将该**线程的中断标识位清除**，然后抛出InterruptedException，此时调用**isInterrupted()方法将会返回false**。

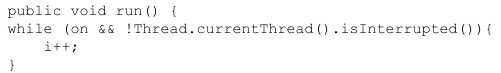
**2.3 过期的suspend()、resume()和stop()；**

暂停、恢复、停止

不建议使用的原因：以suspend()方法为例，在调用后，**线程不会释放已经占有的资源**（比如锁），而是**占有着资源进入睡眠状态**，这样容易**引发死锁问题**。同样，stop()方法在终结一个线程时**不会保证线程的资源正常释放**。

**2.4安全的终止线程**

**中断操作即**调用interrupt方法是一种简便的线程间交互方式，而这种交互方式最适合用来取消或停止任务。除了中断以外，还可以利用一个boolean变量来控制是否需要停止任务并终止该线程。



当on为false时，循环结束，然后run方法结束，线程终止。这种通过**标识位或者中断操作**的方式能够使**线程在终止时有机会去清理资源**，而不是武断地将线程停止，因此这种终止线程的做法显得更加安全和优雅。

**3.线程间的通信**

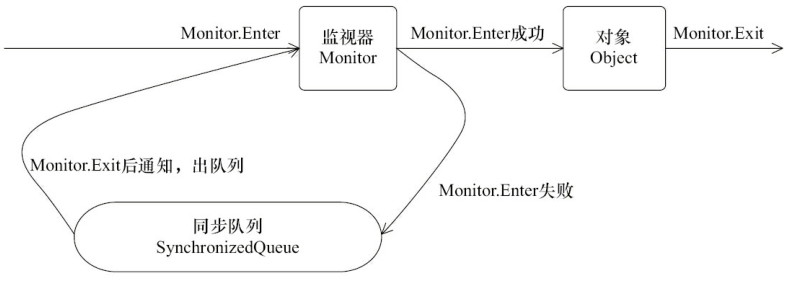
3.1 volatile和synchronized关键字

虽然对象以及成员变量分配的内存是在共享内存中的，但是每个执行的线程还是可以拥有一份拷贝，最终修改的值还是得写入内存中，才会被其他线程可见，因此，程序执行过程中，一个线程看到的变量不一定是最新的。

**Volatile修饰变量**，就是告知程序任何对变量的访问都需要**从共享内存中**获取，而对它的改变必须同步**刷新回内存**，能够保证所有线程对变量的访问的可见性。

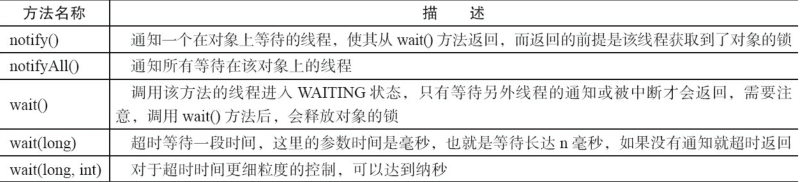
**关键字synchronized**可以修饰方法或者以同步块的形式来进行使用，它主要确保多个线程在同一个时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性。

对于**同步块的实现**使用了**monitorenter和monitorexi**t指令，而**同步方法**则是依靠方法修饰符上的**ACC\_SYNCHRONIZED**来完成的。无论采用哪种方式，其本质是对一个对象的**监视器（monitor）**进行获取，而这个获取过程是**排他的**，也就是同一时刻只能有一个线程获取到由synchronized所保护对象的监视器。



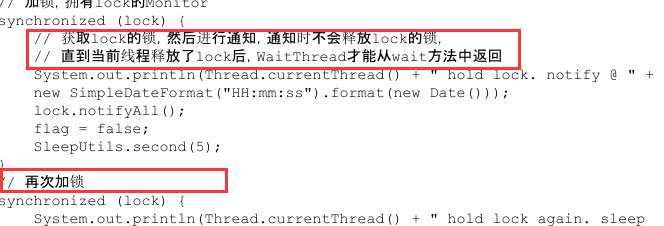
任意线程对**Object（Object由synchronized保护）**的访问，首先要获得**Object的监视器**。如果获取失败，线程进入**同步队列**，线程状态变为**BLOCKED**。当访问Object的前驱（**获得了锁的线程**）释放了锁，则该**释放操作**唤醒阻塞在**同步队列中的线程**，使其重新**尝试对监视器的获取。**

**3.2 等待/通知机制**



**等待/通知机制**，是指一个**线程A**调用了**对象O**的**wait()**方法进入**等待状态**，而另一个**线程B**调用了**对象O**的**notify()**或者**notifyAll()**方法，**线程A**收到**通知后**从**对象O的wait()方法**返回，进而**执行后续操作**。上述两个线程通过对象O来完成交互，而对象上的wait()和notify/notifyAll()的关系就如同**开关信号**一样，用来完成**等待方和通知方**之间的交互工作。





再次加锁前，已经释放了lock的锁，因此此时可以从wait方法中返回。

wait()、notify()以及notifyAll()时需要注意的细节，如下：

①使用上述方法前，需要先对调用对象进行加锁，synchronized

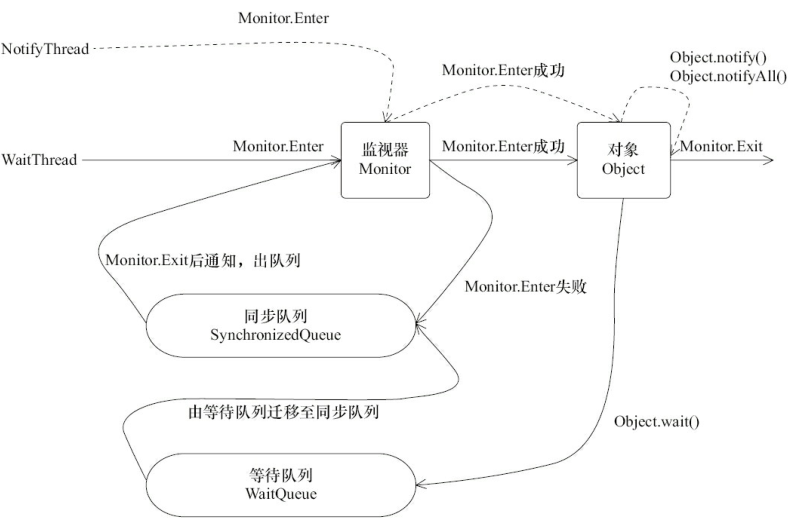
②调用wait方法后，状态由running到waiting，当前线程放入对象的等待队列；

③notify/all方法后，等待线程不会立即从wait返回，需要调用notify的线程释放锁之后，才**有机会**从wait返回。可能不是想要的线程返回，与队列有关。

④notify将**等待队列**中的一个等待线程，从**等待队列**移到**同步队列**，notifyAll是将全部线程移动到**同步队列**，线程状态从**waiting变为blocked**

⑤从wait方法返回的前提是获得了调用对象的锁。

等待/通知机制依托于**同步机制**，其目的就是确保**等待线程从wait()方法返回**时能够感知到**通知线程**对变量**做出的修改**。



WaitThread首先获取了**对象的锁**，然后调用对象的**wait()方法**，从而**放弃了锁**并进入了对象的**等待队列WaitQueue**中，进入等待状态。由于**WaitThread释放了对象的锁**，NotifyThread随后**获取了对象的锁**，并调用对象的notify()方法，将WaitThread从**WaitQueue**移到**SynchronizedQueue**中，此时WaitThread的状态变为**阻塞状态**。**NotifyThread释放了锁**之后，WaitThread再次获取到**锁并从wait()方法返回**继续执行。**再次获取到锁之后，是从wait方法开始执行，不会执行wait前面的语句。**

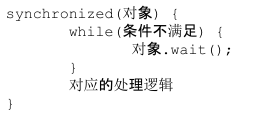
**4.3等待/通知的经典结构**

**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**。



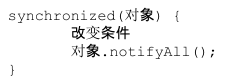
**通知方**遵循如下原则。

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。



**3.4管道输入输出流**

管道输入/输出流，它主要用于**线程之间的数据传输**，而传输的媒介为**内存**。

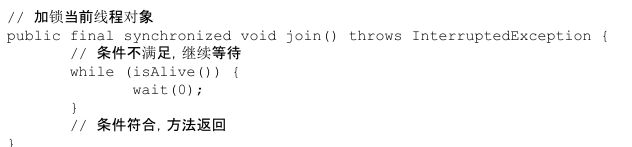
管道输入/输出流主要包括了如下4种具体实现：PipedOutputStream、PipedInputStream、PipedReader和PipedWriter，前两种面向字节，而后两种面向字符。

|  |
| --- |
| **public** **class** Piped {  **public** **static** **void** main(String[] args) **throws** Exception {  PipedWriter out = **new** PipedWriter();  PipedReader in = **new** PipedReader();  // 将输出流和输入流进行连接，否则在使用时会抛出IOException  out.connect(in);  Thread printThread = **new** Thread(**new** Print(in), "PrintThread");  printThread.start();  **int** receive = 0;  **try** {  **while** ((receive = System.***in***.read()) != -1) {  out.write(receive);  }  } **finally** {  out.close();  }  }  **static** **class** Print **implements** Runnable {  **private** PipedReader in;  **public** Print(PipedReader in) {  **this**.in = in;  }  **public** **void** run() {  **int** receive = 0;  **try** {  **while** ((receive = in.read()) != -1) {  System.***out***.print((**char**) receive);  }  } **catch** (IOException ex) {  }  }  } |

从system.in即控制台中读入一个字符，转为int，将int值写入到管道输出流中，既然输入流与输出流已经连接，那么在输入流就会读取到int值，转为char就能输出。对于Piped类型的流，必须先进行绑定，也就是调用connect（）方法，如果没有将输入/输出流绑定起来，对于该流的访问将会抛出异常。

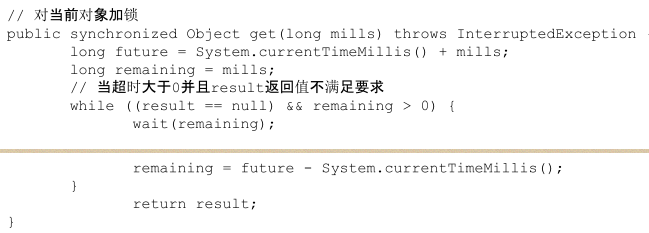
**3.5 Thread.join的使用**

如果一个**线程A**执行了**thread.join()语句**，其含义是：当前**线程A**等待**thread线程终止**之后才从**thread.join()返回**。线程Thread除了提供join()方法之外，还提供了join(long millis)和join(longmillis,int nanos)两个具备超时特性的方法。这两个超时方法表示，如果**线程thread**在给定的**超时时间**里**没有终止**，那么将会从该**超时方法中返回**。



当**线程终止**时，会调用线程自身的**notifyAll()方法**，会通知所有等待在该**线程对象上的线程**。**wait(0);** //如果 timeout 为零，则**不考虑实际时间，在获得通知前该线程将一直等待**。

**4．线程的应用实例**



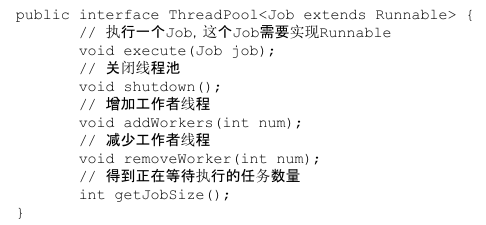
等待超时模式就是在等待/通知范式基础上增加了超时控制，这使得该模式相比原有范式更具有灵活性，因为即使方法执行时间过长，也不会“永久”阻塞调用者，而是会按照调用者的要求“按时”返回。

**4.3线程池技术及其实例**

如果服务端每次接受到一个任务，创建一个线程，然后进行执行，这在原型阶段是个不错的选择，但是面对**成千上万的任务递交**进服务器时，如果还是采用**一个任务一个线程**的方式，那么将会创建**数以万记的线程**，这不是一个好的选择。因为这会使操作系统频繁的**进行线程上下文切换**，无故增加系统的负载，而**线程的创建和消亡都是需要耗费系统资源**的，也无疑浪费了系统资源。

线程池技术能够很好地解决这个问题，它预先创建了**若干数量的线程**，并且**不能由用户直接对线程的创建**进行控制，在这个前提下重复使用固定或较为固定数目的线程来完成任务的执行。这样做的好处是，一方面，**消除了频繁创建和消亡线程的系统资源开销**，另一方面，

面对**过量任务的提交能够平缓的劣化**。



客户端可以通过**execute(Job)方法**将Job提交入**线程池**执行，而客户端自身**不用等待Job的执行完成。**除了execute(Job)方法以外，线程池接口提供了**增大/减少工作者线程以及关闭线程池的方法**。这里**工作者线程**代表着**一个重复执行Job**的线程，而每个由**客户端提交的Job**都将进入到**一个工作队列中等待工作者线程**的处理。

|  |
| --- |
| **public** **class** DefaultThreadPool<Job **extends** Runnable> **implements** ThreadPool<Job> {  // 线程池最大限制数  **private** **static** **final** **int** ***MAX\_WORKER\_NUMBERS*** = 10;  // 线程池默认的数量  **private** **static** **final** **int** ***DEFAULT\_WORKER\_NUMBERS*** = 5;  // 线程池最小的数量  **private** **static** **final** **int** ***MIN\_WORKER\_NUMBERS*** = 1;  // 这是一个工作列表，将会向里面插入工作  **private** **final** LinkedList<Job> jobs = **new** LinkedList<Job>();  // 工作者列表  **private** **final** List<Worker> workers = Collections.*synchronizedList*(**new** ArrayList<Worker>());  // 工作者线程的数量  **private** **int** workerNum = ***DEFAULT\_WORKER\_NUMBERS***;  // 线程编号生成  **private** AtomicLong threadNum = **new** AtomicLong();  **public** DefaultThreadPool() {  initializeWokers(***DEFAULT\_WORKER\_NUMBERS***);  }  **public** DefaultThreadPool(**int** num) {  workerNum = num > MAX\_WORKER\_NUMBERS MAX\_WORKER\_NUMBERS : num < MIN\_WORKER\_  NUMBERS MIN\_WORKER\_NUMBERS : num;  initializeWokers(workerNum);  }  **public** **void** execute(Job job) {  **if** (job != **null**) {  // 添加一个工作，然后进行通知  **synchronized** (jobs) {  jobs.addLast(job);  jobs.notify();  }  }  }  **public** **void** shutdown() {  **for** (Worker worker : workers) {  worker.shutdown();  }  }  **public** **void** addWorkers(**int** num) {  **synchronized** (jobs) {  // 限制新增的Worker数量不能超过最大值  **if** (num + **this**.workerNum > ***MAX\_WORKER\_NUMBERS***) {  num = ***MAX\_WORKER\_NUMBERS*** - **this**.workerNum;  }  initializeWokers(num);  **this**.workerNum += num;  }  }  **public** **void** removeWorker(**int** num) {  **synchronized** (jobs) {  **if** (num >= **this**.workerNum) {  **throw** **new** IllegalArgumentException("beyond workNum");  }  // 按照给定的数量停止Worker  **int** count = 0;  **while** (count < num) {  Worker worker = workers.get(count)  **if** (workers.remove(worker)) {  worker.shutdown();  count++;  }  }  **this**.workerNum -= count;  }  }  **public** **int** getJobSize() {  **return** jobs.size();  }  // 初始化线程工作者  **private** **void** initializeWokers(**int** num) {  **for** (**int** i = 0; i < num; i++) {  Worker worker = **new** Worker();  workers.add(worker);  Thread thread = **new** Thread(worker, "ThreadPool-Worker-" + threadNum.incrementAndGet());  thread.start();  }  }  // 工作者，负责消费任务  **class** Worker **implements** Runnable {  // 是否工作  **private** **volatile** **boolean** running = **true**;  **public** **void** run() {  **while** (running) {  Job job = **null**;  **synchronized** (jobs) {  // 如果工作者列表是空的，那么就wait  **while** (jobs.isEmpty()) {  **try** {  jobs.wait();  } **catch** (InterruptedException ex) {  // 感知到外部对WorkerThread的中断操作，返回  Thread.*currentThread*().interrupt();  **return**;  }  }  // 取出一个Job  job = jobs.removeFirst();  }  **if** (job != **null**) {  **try** {  job.run();  } **catch** (Exception ex) {  // 忽略Job执行中的Exception  }  }  }  }  **public** **void** shutdown() {  running = **false**;  }  }  } |

流程：使用构造器传入指定数目或者默认数目的工作者数目num，调用initialzeWorkers方法，创建num个工作者，添加到workers链表中，并为其创建工作者线程，启动线程；客户端调用execute方法，以jobs工作列表为锁对象，将客户端提交的job加入jobs中，然后job来了，就需要工作线程去处理工作，因此使用jobs的notify方法。工作线程做的工作是，以jobs为锁，如果jobs为空，说明此事没有工作需要做，因此wait等待工作产生。如果收到jobs的notify，说明job过来了，取出第一个job并将其从jobs中移除，执行job的run方法。随着大量的任务被提交，更多的工作者线程会被唤醒。

