# Oracle并发教程

翻译：clark huang

## 进程以及线程

在当前的编程语言中，执行程序执行有两种最基本的单元：线程以及进程。在java编程语言中，大多数的并发程序考虑都是线程。然而，进程同样也是十分重要的一环。一个计算机系统通畅含有很多活跃的进程以及线程。就算是在单核系统中这个说法也是十分正确的。因为在每一个时刻系统只能执行某个单独的线程，执行时间在单核系统中被分给了不同的线程和进程中，这个在系统中叫做，时间分割。

当今社会，多核操作系统下的多进程或者是多线程的执行变得越来越普遍。这对多线程或者是多进程的编程发展起到了十分重要的推动作用。

### 进程

一个进程就是一个自我归一的可执行环境，它的里面包含了所有的执行所有程序的先天条件。一个进程通常有一个完整的，私有的运行资源。特别指出的是，每个进程需要有他自己的内存空间。

进程对应用程序或者是代码来说通常来说是匿名的。然而一个单系统的应用程序可能实际上是多个进程合作并行的结果。为了协调进程间的交互。大多数的操作系统支持IPC 内部进程资源共享。例如，通过管道，或者是Sockets。 IPC不仅仅是用于本机器的线程间的共享资源。并且还可以应用在不同系统间的进程之间。

### 线程

线程也被称之为轻量级的进程。她们两都提供程序的执行环境。但是创建一个新的线程所需要的资源相对进程来说少很多。

线程是存活在进程内部的，每一个进程至少拥有一个线程。线程分享进程的资源，包括内存和已经开启了的文件。这么做的目的是为了效率，但是也有出问题的潜在威胁。

多线程执行是java平台生存的水分。每一个应用程序至少有一个线程或者多个用于系统内存管理或者是信号量的处理等等。

## 线程对象

每一个线程都是Thread类的一个实例。在java中有两种方式完成并发程序。

* 一种是直接操控Thread类，包括他的创建销毁等等。
* 另一种是抽象出对Thread的管理，通过executor进行任务的并发处理

这个章节我们主要讨论的就是第一种，第二种我们会在高级别的并发对象中进行讨论

### 定义和开启线程

创建一个线程的实例我们必须给予这个线程运行的代码，我们可以通过以下两种方式

* 第一种是通过对Runnable接口的实现完成。
* 另外一种是通过对Thread类的继承然后对run的重写完成。因为Thread本身是对Runnable进行实现了的。只是run方法并没有做任何东西。

这两种方式你喜欢哪一种呢？由于第一种方式，让我们的自定义类可以去实现另外的一个类，而第二种方式则一定死了他是作为Thread类的一个子类，所以相对来说第一种方式更为灵活。并且在高级别的对线程进行管理也都是使用的这种方式。

### 通过睡眠阻断执行

Thread.Sleep能够让当前线程在一段时间内被悬挂起来。这是一个十分有效的方式，是的其他的线程或者其他的应用程序获得时间片段。Sleep也可以被用于调频在接下来的例子中会有所涉及。

有两个重载的Sleep方法。分别接收毫秒数和纳米数。然而这些时间并不能保证完全的正确性，因为她们是由程序运行的底层OS系统决定的。

这个方法可能会发生的异常是InteruptedException 。 当当前线程Sleep的时候，如果被别的线程打扰当前休息的线程的时候。

### 打断

一个中断标识表示当前线程应该停止他正在做的事情，开始做点其他的事情。当然这个完全取决于程序员，通常情况下的处理方式，如果当前线程被中断，那么就会退出当前线程。

通常可以通过Thread.interrupted() 方法进行判断当前线程是否被中断。

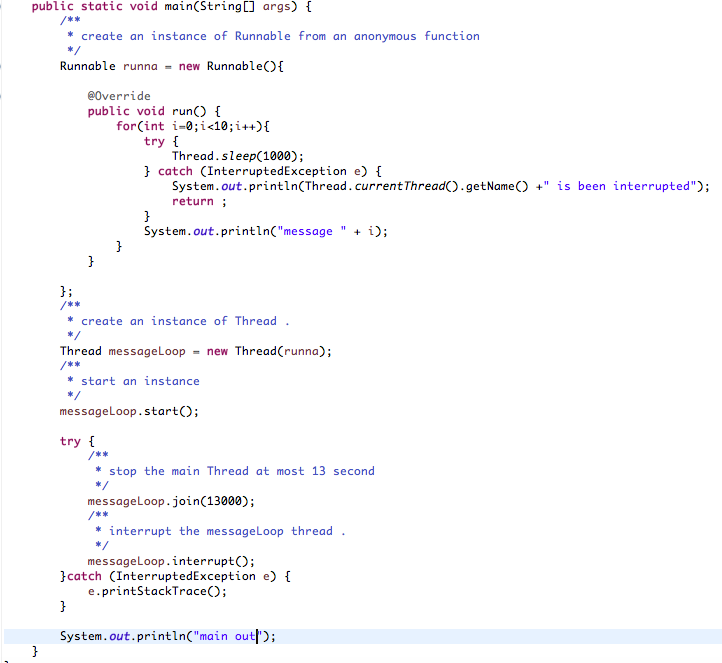
### 加入

join方法允许一个线程等待着另一个线程的结束然后再开始执行。例如如下，当前线程会进入暂停状态直到t线程完成了所有的执行为止。

t.join()

t的重载方法可以设置等待的具体时间。和sleep一样这个时间也不是准确的。由OS决定。并且也会抛出InterruptedException

### 简单的线程用例



## 同步

线程之间的交流通常通过共享数据来实现的，这种沟通方式效率十分的高，但是可能产生两种错误。一种叫做线程干扰，一种叫做内存一致性错误。阻止这两种错误的方式叫做synchronization

然而，同步却可能造成“线程争论”，这种情况的发生当多个线程试图同时抢夺一个资源的时候，导致java虚拟机运行其他线程的时候，更加的缓慢，甚至有可能造成她们的执行暂定。饥饿和活锁都是这种形式的“线程争论”。

### 线程干扰

i++ 当多个线程对进行i++操作的时候然后使用i的值，这个时候i的值可能由于线程干扰导致出现非预期的情况。比如当两个线程同时执行i++的操作，i从0开始计数。这个时候这两个线程可能得到的只都是1而产生不一致的情形发生。

### 内存一致性冲突

内存一致性错误发生在对同一个数据的当前数据值的不一致性观念导致的。导致内存不一致性问题的原因十分的复杂，也超出了这个教程的范围。幸运的是，编程者不需要了解其中的细节，而只需要知道如何规避这类问题即可。

避免内存不一致性的关键在于理解Happen-before 关系。这个关系简单的保证了单线程内存写入对另外一个线程的可见性即可。例如上面的哪个例子。如果两个线程同时操作i++，并且打出print（i）的值。当这两个语句通知执行在同一个线程的时候print语句打出的值几本可以确定是1，然而如果这两个语句，分别被不同的线程执行呢。这个时候print的值可能会是0因为我们不能保证其中执行i++的线程对执行print的线程是可见的。除非编程者建立了一个happens-before 关系在两个程序之间。

有几种不同的方式创建happen-before 关系。其中一个就是synchronzied 。 我们已经见到过两种创建happends-before的关系：

* 当一个线程调用了Thread.start 方法，每一个执行语句对执行了这个方法的执行语句有一个happen-before 的关系，每个语句也有一个happen-before 的关系和所有被新线程执行的语句。
* 当被终止一个线程调用了Thread.join在另外一个线程中的时候，这个时候所有的可执行语句在被终止的线程中的，对其他所有正确执行的线程有一个happen-before的关系。效果就是join双方的代码对对方都是可见的。

### 同步方法

java 提供了两种常见的同步的方式，一种是同步方法，一种是同步语句。其中同步语句相对来说更为复杂。会在下一章节进行描述。

进行方法同步有以下几个效果：

* 首先，对同一个方法的调用不会产生交叉的效果。当一个线程通过一个对象调用这个方法的时候，那么其他所有的线程通过这个对象调用这个方法，都会被阻塞在方法外面
* 第二，当一个方法同步后，它就自动建立起了happens-before 原则对其他的调用这个方法线程，这样就保证了对象的状态对其他的所有的线程都是可见的。

注意：构造器是不能被同步的，使用synchronized关键字在构造器上会包编译错误。同步构造器本身就说不通，因为创造对象的时候，紧紧只允许有一个线程有权限进入构造方法。

警告：当创建一个对象可能被多个线程共享的时候，千万不要提早的把对象暴露出来。例如：

在构造器中写入这样的语句。

list.add(this)

这样的语句过早的暴露了当前对象，可能当前对象根本就没有创建完成，就已经被暴露了。

同步方法对线程干扰和内存一致性错误：如果一个对象对不止一个线程可见。那么所有的读和写对那个实例的变量都应该是同步的。（一个例外：final 的一些变量。由于final变量在申明的同时已经被构建了，所以对他的读操作即使不是用关键字synchronized也是没有问题的。）这个策略是有效的，但是能够引起活跃锁的问题，后面的张杰会进行讨论。

### 内部锁和同步

当同步构建在实体内部的时候这个时候称为内部锁或者是监控锁。（API对这些实体的定义简单问监控器）内部锁在对一个对象的状态的强制祛除方面，以及建立happen－before管理起着很为重要的作用。

每一个对象都有一个内部锁与之联系着。为了方便起见，一个线程要对一个对象的属性进行访问的时候，必须获取这个对象的内部锁。并且在用完属性之后释放内部锁。一个线程在进入和退出对象的这段时间拥有对象的内部锁。只要一个线程拥有内部锁，其他的所有的线程则不能拥有这个锁。其他的线程会被阻塞住，当它们常识去获取锁的时候。

当一个线程去调用一个同步了的方法时候。它会自动获取这个对象方法的内部锁并在方法运行完后释放这个内部锁。当这个方法调用异常的时候这个锁同样会被释放掉。

第二种同步的方式是使用同步语句。相比于同步方法。同步语句必须指定对象提供对象的内部锁。

public void addName(){

synchronized(this){

lastName = name;

nameCount++;

}

list.add(name);

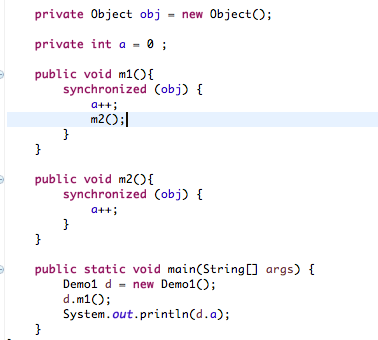
}

在这个例子中addname方法需要为lastname和namecount提供同步保障。也就是这两个方法需要同时被一个线程调用。在同步语句中调用其他对象的方法容易产生活跃度问题（Liveness）

同步语句对于提高线程同步效率很有效。

重用同步

我们知道一个线程不能获取一个呗其他线程占用的线程。但是一个线程能够获取一个他自己已经拥有的锁。允许一个线程多次获取同一个锁的行为我们称之为可重用同步。比如说这样的一个情形，当一个同步语句中直接或者间接的调用了一个方法然后这个方法同样也包含同步语句，并且这两个同步语句使用的是同样的锁。如果没有可重用锁的话，我们程序猿就必须做很多预防措施来避免我们在同步语句中调用了同样的锁锁住的方法。



### 原子化权限

在编程语言中，原子性操作指的是所有的一次性发生。一个原子性操作不能中间被打断。他要么就是完全执行，要么就是完全不执行。原子性操作对所有的其他操作都是可见的。

我们先前已经看到过的一个自增表达式例如，c++，并不是原子性操作。即使是非常简单的表达式也可以被拆分成很多的行为。然而下面的一些特定的行为都是原子性的：

* 对引用变量和大多数基础类型（出了long类型和double类型之外的）的操作是原子性的
* 当变量申明加上了volatile的时候读写是具有原子性的。（包括long 和 double）

原子性的操作是不能被打断的。所以在多线程中可以安心使用。然而这个并没有包含同步的原子性操作。因为内存一致性错误还是有可能的。使用vialite变量降低了内存一致性错误的风险，因为任何对这种个变量的读写都建立了happen-before 关系对其他的线程对当前变量的读取。对这种变量的修改对其他线程总是可见的。更重要的是，这意味着当一个线程读取vialite变量，它不仅可以看到最后的一次更改后的值，并且能够看到代码对这个变量的改变所带来的副作用。因为这个变量可以随时被其他线程改变，所以，我们对他的连续使用性的时候的问题需要考虑下。

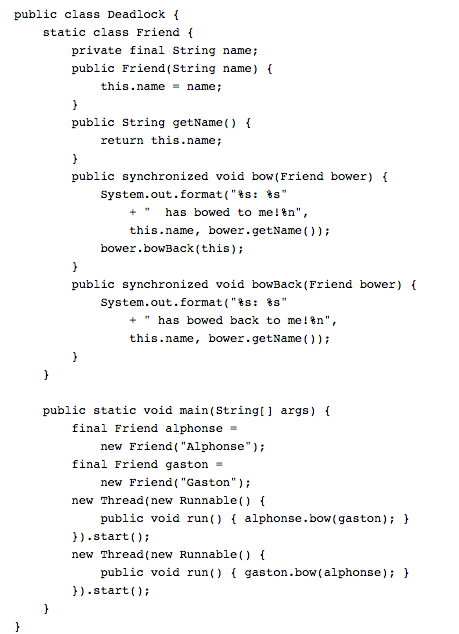
使用这些简单的原子性变量效率更高相比较于同步语句来说的话，但是需要更多的对内存一致性问题的考虑。

## 活跃性

一个并发程序能够在世间间隔中执行我们称之为活跃度。

### 死锁

双方都拿着对方需要的锁这个时候就形成了死锁。



### 饥饿和活锁

饥饿和活锁相对死锁来说不是那么常见，但是对于每个并发程序设计者来说也是很容易犯的错误。

饥饿指的是一个线程对资源的获取没有能力获取，并且没有能力增加获取的机会。这个通常发生在一些很贪婪的线程对共享资源的长期占用，而导致这个的部分线程长时间不能够获取资源。例如。如果一个同步方法执行需要用很长的时间。并且有一个线程长时间调用这个方法。就会导致其他线程获取这个方法的锁的能力变得很少。

活锁：如果一个线程的反馈是根据另外一个线程，然后另外一个线程的反馈是根据另外一个线程，这个时候就可能产生活锁。活锁线程不能够进一步执行。然而这个线程也并没有阻塞。他们只是太忙于相互反馈。这个过程就相当于 A 和B 试图穿过对方。A 转向左边， B转向右边。 导致他们还是阻塞着对方。然后就一直抱着这这样的动作。就产生了活锁。

死锁于活锁相比较。

共同点： 两者都涉及到至少两个线程。并且他们的相互继续执行都需要对方的支持。

不同点：死锁是不释放手上的锁。而活锁是马上释放手上的锁，去抢另外的一个锁，然后发现哪个锁现在也是不能够获取的马上释放掉手中的动作，去抢另外一个锁。一直在行动，从未找到解决方案。

## 保证锁

线程之间通常需要协调他们之间的行为。最常见的协调机制就是通过保证锁。通常这样的锁的描述是：首先需要满足某种条件，然后再能进行阻塞。完成保证锁，需要进行几步操作才行。

其实上面的描述可以通过下面的代码实现

while(condition){}

code

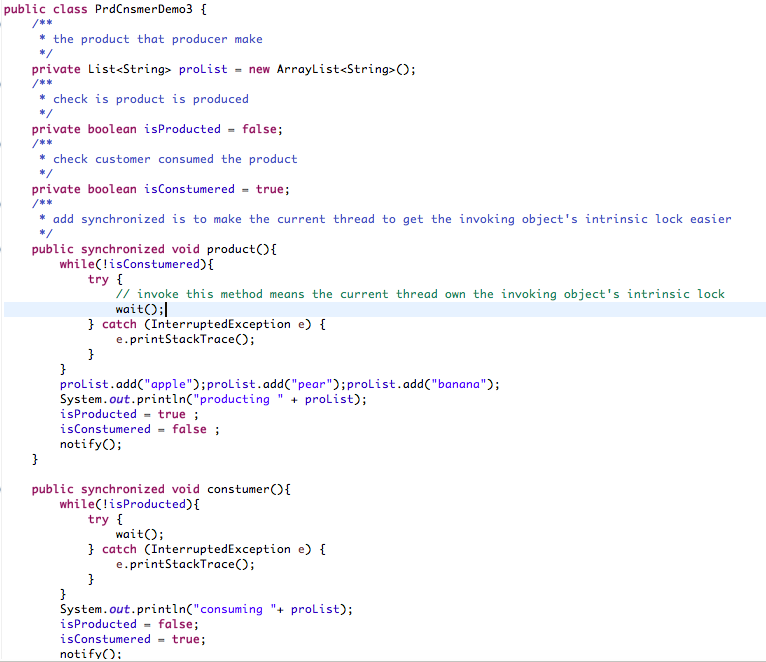
但是这样的无限循环代码，对性能的损耗十分严重，因为他一直在运行。占用时间片段。

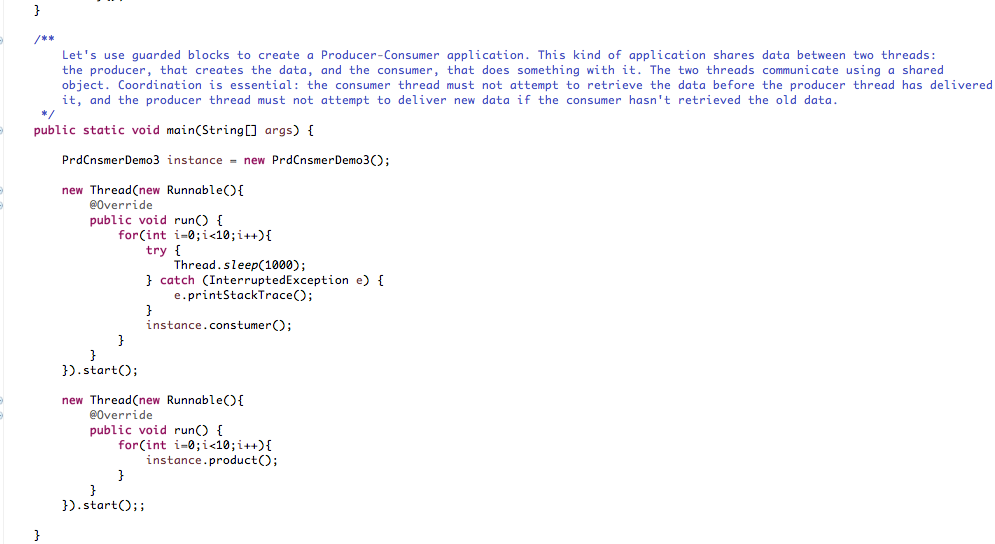
一个更为有效的方式是调用Ojbect.wait方法来将当前线程挂起来。Wait方法会一直阻塞到另外一个线程通知事件的发生。

While(condition){try{wait()}catch(InterruptionException ex){}}

Code

实例：





## 不可改变对象

当一个对象被创建的时候就不可改变了，这个时候我们认为这个对象是一个不可改变对象。对于创建简单可靠的代码来说，使用不可改变对象被人们普遍的接受。

不可改变对象在并发应用中的用途更加的明显。由于他们的状态不可改变，因此，他们不能被其他线程弄到一个不连续的状态。

然而对程序猿来说，却时常不太喜欢使用不可改变对象。因为他们担心创建一个不可改变状态的对象会不会消耗比较大。创建对象的消耗，通常被过高估计了。对于不可改变对象的效率问题，我们甚至能够通过一些方式增加其效率。这些可能因为GC而触发一些瓶颈。而对代码的考虑中，我们也会防止可改变对象的状态维护以及破裂现象

下面的章节中，会创建一个可改变的对象，并且从中分离出一些不可改变的实例。这么做，他们是拥有一些普遍的规则，以及会说明一些不可改变对象使用的优势。

### 一个同步类的小例子

略

### 定义不可改变对象的策略

通过下面的一些方式我们可以简单的设置一些不可改变的对象。当然即使依据这些规则我们也不能说所有的类都是不可改变的对象。这样的方式叙写类，并不是寿命创作者的粗心。他们可能会有很好的理由说明这些类的实例可能永远不会改变，在创建过后。然而，这样的策略要求很相信的分析，并不适用于初学者。

1. 首先就是去掉所有的set方法。这些方法能够改变field的值，在创建不可改变类的时候不需要使用。
2. 将所有的属性设置成final的活着时private
3. 不要让子类去重写你的方法。最简单的方法做到这种就是定义你的class为final的一个更有效的方式是通过将你的构造器设置成private的，创建实例通过工厂方法创建。
4. 如果实例中包含一些可变对象的引用，不要让这些对象可变

* 不要提供可以修改这个变量的方法
* 不要存储外部的引用，可修改对象最好是通过构造器传递过来，如果可能的话，创建复制对象。存储复制后的对象引用。

## 高级别并发

到目前为止，我们一直关注的都是低级别的API这个是java平台起始时候十分基础的一些功能。这些API能够保证完成一些很基本的功能，但是对于高级别的并发任务可能需要更加高级别的API。这个在当前社会的多核系统以及多核cpu的系统中更是特别的重要。

这个章节我们会查看一些高级别的特性在java平台之后的。大多数的api集中在java.util.concurrent包下面。也有一些特性在conlections中。

### 锁

同步代码依赖的其实是一个简单的可重用锁，这种锁十分 容易使用。但是有很多限制。更多的有效的锁的用法在locks包中。这边并不会对这个包做一个详细的说明。但是会集中对其中一个接口lock进行集中描述。

锁对象启示和同步语句很像。对于隐式的锁每次只能有一个对象可以拥有一个锁。Lock对象也是支持wait/notify这种保证锁的形式的通过的是一个叫做Condition的对象。

对于锁对象相对于隐式锁的最大的优势就在于它可以尝试去获取锁，并且可以退出来。通过tryLock方法去尝试获取，如果没有获取到就会退出来。也可以设置一个获取时间限制。方法lockInterruptibly会返回如果其他线程打断了这个线程在获取锁的过程中。

通过Lock对象我们可以解决我们之前涉及到的死锁的问题。

### 执行器

在之前的章节中，在任务完成和创建线程之间，形成了很强的耦合。这种虽然对于小型的程序来说已经是足够的但是对于大型的系统来说，我们最好能够将对线程的创建和具体的task分开解除耦合。这个类我们称之为executors。

Executor interface

在concurrent包中定义了三个执行接口：

* Executor ， 一个简单的接口支持，fire 新的任务。
* ExecutorService Executor的子接口，添加了新的属性用于管理任务的生命周期和executor本身
* ScheduledExecutorService 第二个接口的子接口支持时间话任务执行。

通常来说，和执行器相关的对象都会声明为其中之一。

Thread Pool

许多concurrent包中的executor方法的实现都使用到了线程池。这些线程独立于Runnable 和Callable存在，并通常被用于对多任务执行中。

使用这种形式的工作线程极小话了线程创建的瓶颈。因为线程对象需要大量的内存空间所以对线程的创建和销毁很容易产生瓶颈问题。

一种常见的线程池就是固定大笑的线程池。这种线程池，拥有一个固定的大小，当线程突然掉了一个后会自动补上。

可以使用Executors方法中的方法完成对线程池的使用。

包括newCachedThreadPool 等等

Forx／join（暂时只是做一个简短的说明具体以后在项目中使用到后进一步描述）

Forx/join框架式ExecutorService的一个实现方法。占用多核处理器的优势。它被设计用于对递归任务时候的算法的情形。目标是使用所有可使用的线程力量去提高程序的表现效率。

Forx/join 分发任务通过线程池。通过work-steal 算法区分。工作线程忙完自己手上的事情的话，可以接手比较忙的线程完成他们的任务。

这个框架的核心类式ForkJoinPool类也是对上面算法的实现。

代码的大致形态应该是：

if (my portion of the work is small enough)

do the work directly

else

split my work into two pieces

invoke the two pieces and wait for the results.

### 同步集合

BlockingQueue ： 先进先出数据结构如果队列为空的时候，取数据，或者队列满了的时候，添加数据都会造成阻塞现象

ConcurrentMap 是用于原子性操作的Map。

### 原子变量

AtomicLong AtomicInteger

### 同步随机数

对于多线程同步随机数的使用时候使用ThreadLocalRandom。它的使用ThreasdLocalRandom.current().nextInt(10)

## 继续深造

## 问答和实践