# 第一章 并发编程的挑战

在并发编程时，若希望通过多线程执行任务让程序运行得更快，会面临非常多的挑战： 上下文切换问题，死锁问题，受限于硬件和软件的资源限制问题。

## 1.1 上下文切换

CPU通过时间片分配算法来执行任务，当前任务执行一个时间片后会切换到下一个，切换之前会保存上一个任务的状态。

任务从保存到再加载的过程就是一次上下文切换。上下文切换会影响多线程的执行速度。

**减少上下文切换的方法**：

* 无锁并发编程：多线程处理数据时，可以用一定的方法来避免锁的使用。
* CAS算法：Java的Atomic包使用CAS算法来更新数据，不需要锁
* 使用最少线程：避免创建不需要的线程
* 协程：在单线程中实现多任务的调度，并在单线程里维持多个任务间的切换

减少上下文切换实战：

* 用jstack命令dump线程信息： jstack {pid} > d:/dump12
* 统计所有线程分别处于什么状态：[grep java.lang.Thread.State dump12 | awk ‘{print $2$3$4$5}’ | sort | uniq –c ]
* 打开dump文件查看处于WAITING的线程在做什么
* 减少WAITING状态的线程数目，修改线程池的配置信息，将maxThreads降低

## 1.2 避免死锁的常见方法

* 避免一个线程同时获取多个锁
* 避免一个线程在锁内占用多个资源，尽量保证每个锁只占用一个资源
* 尝试使用定时锁，使用lock.tryLock(timeout)来代替内部锁
* 对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会解锁失败

## 1.3 资源限制

资源限制是指程序并发执行速度受限于计算机硬件资源（带宽、CPU等）或软件资源。

资源限制引发的问题：

受限于资源，并发执行的代码仍然在串行执行，由于上下文的切换，程序不会加快执行，反而会更慢。

解决方法：

对于硬件资源的限制可以使用集群并发执行；对于软件资源的限制可以考虑资源池复用

根据不同的资源限制调整程序的并发度。比如下载文件程序依赖两个资源：带宽和磁盘读写速度；数据库操作涉及数据库连接数，SQL语句执行速度。

# 第二章 并发机制的底层实现原理

Java中所使用的并发机制依赖于JVM 实现和CPU指令

## 2.1 volatile的应用

volatile是轻量级的synchronized，它在并发处理器开发中保证了共享变量的“可见性”，使用恰当的话，比synchronized的使用和执行成本更低，因为它不会引起线程上下文切换。

**Java线程内存模型能够确保所有线程看到的volatile声明变量的值是一致的**。

CPU术语

* **缓冲行**：缓存中可以分配的最小存储单位，处理器填写缓存线时会加载整个缓存线，需要多个主内存读周期
* **原子操作**：不可中断的一个或一系列操作
* **缓存行填充**：当处理器识别到内存中读取操作时可缓存的，处理器读取整个缓存行到适当缓存
* **缓存命中**：若高速缓存行填充操作的内存位置仍然是下次处理器访问的地址，处理器从缓存中读取，而不是从内存中读取
* **写命中**：如果会写的操作数在有效的缓存中时，则将改操作数回写到缓存中，而不是内存中。
* **写缺失**：一个有效的缓存行被写到不存在的内存区域

volatile的实现原则：处理器缓存回写到内存；一个处理器的缓存回写到内存，会导致其他处理器的缓存无效。

## 2.2 synchronized的实现原理与应用

使用synchronized实现同步的基础：Java中每个对象都可以作为锁：

* 对于普通同步方法：锁是当前实例对象
* 对于静态同步方法：锁是当前类的Class对象
* 对于同步方法块：锁是synchronized括号里配置的对象

JVM是基于进入和退出Monitor对象来实现方法同步和代码块同步的。

synchronized用的锁是存在Java对象头里的，如果对象是数组，则虚拟机用3个字宽存储对象头。

java中的锁一共有四种状态：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态。这几个状态会随着竞争情况逐渐升级，但不可降级。

**偏向锁**：

考虑到大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，所以引入了偏向锁。

偏向锁使用了一种等待竞争出现才释放锁的机制，偏向锁的撤销需要等待全局安全点。

-XX:BiasedLockingStartupDelay=0 来关闭偏向锁的启动延迟

-XX:UseBiasedLocking=false 当应用程序所有的锁通常都是处于竞争情况是，可以默认不是用偏向锁，默认会进入到轻量级锁状态

**轻量级锁**：

线程在执行同步块之前，JVM会首先在当前线程的栈桢中穿件用于存储锁记录的空间，并将对象头中的Mark Word复制到锁记录中。

然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针，若成功则获得锁；若失败，表示其他线程竞争锁，当前线程使用自旋来获取锁，或升级成重量级锁。

**锁的优缺点对比**

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 使用场景 |
| 偏向锁 | 加解锁无需额外消耗，与非同步方法仅有纳秒级差距 | 若存在锁竞争，会带来额外的锁撤销消耗 | 只有一个线程访问同步块场景 |
| 轻量级锁 | 竞争的线程不会阻塞，提高程序的相应速度 | 若时钟得不到锁的竞争线程，自旋消耗CPU | 追求响应时间  同步块执行速度快 |
| 重量级锁 | 线程竞争不使用自旋，不消耗CPU | 线程阻塞，响应时间慢 | 追求吞吐量  同步块执行时间较长 |

## 2.3 原子操作实现原理

处理器实现原子操作有两个机制：

* **总线锁保证原子性**：总线锁就是使用处理器提供的一个#LOCK信号，当一个处理器在总线上输出此信号时，其他处理器的请求被阻塞，该处理器独占共享内存。
* **缓存锁保证原子性**：锁定缓存行的数据，使用缓存一致性来保证原子性。跨多个缓存行时不能使用。

Java中可以使用CAS实现原子操作。

CAS实现原子操作的三大问题：

* **ABA问题**：CAS需要在操作值时检查值是否发生了变化.若值由A变为B，再变为A，则认为没有变化，可以使用AtomicStampedReference来解决ABA问题。
* **循环时间长开销大**：自旋CAS长时间执行不成功，会给CPU带来较大的开销
* **只能保证一个共享变量的原子操作**：使用AtomicReference类来保证引用对象之间的原子性，可以将多个变量放到一个对象里。

# 第三章 Java内存模型

Java内存模型的基础，顺序一致性，同步原语

## 1. Java内存基础

线程之间如何通信和线程之间如何同步是并发变成需要处理的两个关键问题。在命令式编程中，线程的同步机制一般有两种：共享内存和消息传递。

Java中，所有实例域、静态域和数组元素都存储在堆内存中，堆内存在线程之间共享，所以会有可见性问题。

