java并发编程笔记—volatile与synchronized关键字

|  |  |
| --- | --- |
|  | 🢂 内容概览 |
|  | Why：此文档用来做什么？它存在的意义是什么？为解决什么问题？   |  | | --- | |  |   What：当前包含了那些内容？   |  | | --- | |  |   How：此文档应如何参考？   |  | | --- | |  |   Who：此文档适用于那些人员阅读参考？   |  | | --- | |  |   Summary：摘要   |  | | --- | | 1. volatile和synchronized的区别：volatile不保证原子性，保证变量范围的内存可见性，且禁止指令重排序，不会引起上下文切换，开销相对较小；synchronized保证原子性，代码范围的内存可见性，会引起上下文切换，开销相对较大； 2. 性能分析工具：Lmbench3 3. linux监控工具：vmstat 4. jvm线程分析：jstack 进程号 |   Reference：参考文献   |  | | --- | | [Java Synchronized Mark Word](http://blog.csdn.net/xiaomin1991222/article/details/50981423)  [JVM内部细节之一：synchronized关键字及实现细节（轻量级锁Lightweight Locking）](http://www.cnblogs.com/javaminer/p/3889023.html)  [JVM内部细节之二：偏向锁（Biased Locking）](http://www.cnblogs.com/javaminer/p/3892288.html)  [Java 轻量级锁原理详解(Lightweight Locking)](http://blog.csdn.net/songylwq/article/details/5585734)  [Linux用户态和内核态](http://blog.chinaunix.net/uid-1829236-id-3182279.html) | |

目录

[1 合理使用并发 3](#_Toc453749307)

[1.1 如何减少上下文切换？ 3](#_Toc453749308)

[1.2 如何避免死锁？ 3](#_Toc453749309)

[2 volatile关键字 3](#_Toc453749310)

[2.1 cpu术语 3](#_Toc453749311)

[2.2 lock指令 3](#_Toc453749312)

[2.3 缓存一致性协议 4](#_Toc453749313)

[2.4 编程技巧：通过追加字节，提高性能 4](#_Toc453749314)

[3 Synchronized关键字 4](#_Toc453749315)

[3.1 实现基础 4](#_Toc453749316)

[3.2 Java对象头 5](#_Toc453749317)

[3.2.1 对象头存储结构 5](#_Toc453749318)

[3.2.2 Mark Word存储结构 5](#_Toc453749319)

[3.2.2.1 32位JVM 5](#_Toc453749320)

[3.2.2.2 64位JVM 6](#_Toc453749321)

[3.3 Monitor Record 6](#_Toc453749322)

[3.4 锁升级 7](#_Toc453749323)

[3.4.1 偏向锁 8](#_Toc453749324)

[3.4.2 轻量级锁 9](#_Toc453749325)

[3.4.3 重量级锁 11](#_Toc453749326)

[3.4.4 锁升级过程 11](#_Toc453749327)

[3.4.5 锁的优缺点 12](#_Toc453749328)

[4 原子操作实现原理 13](#_Toc453749329)

[4.1 cpu术语 13](#_Toc453749330)

[4.2 处理器如何实现原子操作 13](#_Toc453749331)

[4.3 Java如何实现原子操作 13](#_Toc453749332)

[5 运维经验 14](#_Toc453749333)

# 合理使用并发

## 单线程程序并不一定比多线程程序性能差

1. 当任务较轻，执行任务的消耗没有开启多线程消耗多时；
2. 当上下文切换带来的消耗较高时；
3. 当多线程的同步处理代价过大时；

## 并发的优势

1. 提高系统的吞吐率：能够合理的利用IO等待时间等情况；
2. 提高响应性：防止因为处理业务等待导致响应时间过长；
3. 充分利用多核CPU资源：单线程程序对多核CPU使用效率不高；
4. 最小化系统资源的使用：线程共享资源，避免了多进程造成的资源浪费；
5. 简化程序结构：现实世界是并行的，多线程能够更容易的表达现实世界；

## 如何减少上下文切换？

1. 无锁并发编程：不同的线程尽可能减少共享数据的范围，或者不共享数据；对于共享数据，可以考虑将数据分片，操作时，不同的线程访问不同的分片，从而减小数据共享范围；
2. CAS算法：使用CAS算法，通过乐观锁代替悲观锁，减少不必要的锁；
3. 最少线程：线程并非越多越好，使用合适的线程数，避免产生大量的上下文切换；
4. 协程：在单个线程中实现多任务调度，并在单个线程里维护多个任务间的切换；

## 如何避免死锁？

1. 避免一个线程同时获取多个锁；
2. 避免一个线程在锁内同时占用多个资源，尽量保证一个锁只占用一个资源；
3. 使用定时锁tryLock代替内部锁机制，通过锁超时打破死锁；
4. 对于数据库锁，加锁和解决应该放在一个会话（连接）中；

# volatile关键字

## 核心功能

1. **保证内存可见性：**一个线程对一个被volatile修饰的变量值的更改，对于后面要访问该变量的线程总是可见的；
2. **禁止指令重排序：**禁止编译器和CPU进行指令重排序；

## cpu术语

|  |  |
| --- | --- |
| **Cpu术语** | **描述** |
| 内存屏障 | 一组处理器指令，用于实现对内存操作的限制； |
| 缓存行 | CPU高速缓存中可以分配的最小存储单位，处理器填写缓存行时，会加载整个缓存行； |
| 原子操作 | 不可中断的一个或者一系列操作； |
| 缓冲行填充 | 当处理器识别到从内存中读取操作数是可缓存的，则处理器会读取整个高速缓存行到适当的缓存；（L1,L2,L3的或所有） |
| 缓存命中 | 如果进行高速缓存行填充操作的内存位置仍然是下一次处理器访问的地址时，处理器从缓存中读取操作数，而不是从内存读取； |
| 写命中 | 当处理器将操作数写到一个内存的区域时，会首先检查缓存的内存地址是否在缓存行中，如果存在且有效，则处理器将操作数回写到高速缓存； |
| 写缺失 | 一个有效的缓存行被写入到不存在的内存区域； |

## lock指令

1. 当volatile修饰的共享变量进行写操作时，会多出一条lock指令；
2. lock指令的功能如下：
   1. 将当前处理器缓存行中的数据写回到系统内存；
   2. 写回内存操作会使其它CPU里缓存了该内存地址的数据无效；
3. lock指令会被翻译称为LOCK#信号；LOCK#信号的功能用于保证处理器声言该信号时，保证cpu操作的数据对其它cpu可见；通常有如下两种实现：
   1. 锁定总线：保持在处理器声言该信号时，能够独占内存；锁住总线，导致其它处理器不能访问总线，即其它处理器此时不可以访问内存；但锁定总线开销较大；
   2. 锁定缓存：锁定当前处理器处理数据对应的高速缓存行，并回写数据到内存；通过处理器的缓存一致性协议来通知其它cpu数据发生变更；该实现相比而言开销较小；且由于缓存一致性协议的限制，不会出现同一时间多个cpu都更新同一数据的情况；

## 缓存一致性协议

1. cpu实现了缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器对这个数据进行修改操作时，会重新从系统内存中把数据读到处理器缓存里；
2. 缓存一致性协议会阻止同时修改两个以上的处理器缓存的内存数据区域；

## 编程技巧：通过追加字节，提高性能

1. LinkedTransferQueue中，链表中的节点，通过定义无用的引用，将节点占用内存追加到64字节，从而保证处理器每次读取缓存行时，都只能读取一个节点的数据，避免了多个处理器同时访问相同缓存行的情况；
2. 如果不追加字节，则处理器每次固定读取64个字节的数据，有可能将队列的头结点和尾节点放入同一个缓存行，导致其它处理器无法同时访问同一缓存行，降低性能；
3. 适用场景：
   1. cpu字节宽度为64字节；
   2. 共享变量被频繁写；

# Synchronized关键字

## 核心功能

1. **实现操作的原子性：**通过关键字设定临界区，保证同一时间仅有一个线程可以进入临界区；
2. **保证内存的可见性：**一个线程执行临界区的代码对数据的修改，对后面进入临界区的线程可见；

## 实现基础

1. Java中的每一个对象都可以作为锁；表现如下：
   1. 普通同步方法（对象方法）：当前对象；
   2. 静态同步方法（类方法）：当前类的类对象；
   3. 同步方法块：括号中匹配的对象；
2. JVM通过进入和退出Monitor对象来实现方法同步和代码块同步；两者实现细节不同；
3. 同步代码块：通过monitorenter和monitorexit指令实现；monitorenter指令是在编译后插入到同步代码块开始位置；monitorexit插入到方法结束处和异常处；JVM保证每个monitorenter指令都有与之对应的monitorexit指令；任何对象都有一个monitor对象与之关联，当monitor被持有后，即处于锁定状态；执行monitorenter指令时，会尝试获取对象对应的monitor的所有权，即尝试获取对象的锁；
4. synchronized方法：则会被翻译成普通的方法调用和返回指令如:invokevirtual、areturn指令，在VM字节码层面并没有任何特别的指令来实现被synchronized修饰的方法，而是在Class文件的方法表中将该方法的access\_flags字段中的synchronized标志置为1，表示该方法是同步方法并使用调用该方法的对象或该方法所属的Class在JVM的内部对象表示Klass做为锁对象。

## Java对象头

1. Synchronized的锁存放在Java对象头中；如果对象是数组类型，则对象头占3个字，否则占两个字；例如：32位JVM中，一个字为4个字节，则数组类型的对象头占用12个字节，非数组类型的对象头占用8个字节；

### 对象头存储结构

|  |  |  |
| --- | --- | --- |
| **存放内容** | **长度** | **描述** |
| Mark Word | 32/64bit | 存放对象的hashcode、锁、分代年龄等信息； |
| Class Metadata Address | 32/64bit | 存放对象对应类型数据的指针； |
| ArrayLength | 32/32bit | 存放数组长度；（若当前对象为数组类型） |

### Mark Word存储结构

Mark Work中的数据会随着对象锁标志位的变化而变化；

#### 32位JVM

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **锁状态** | **25bit** | | **4bit** | **1bit**  **是否是偏向锁** | **2bit**  **锁标志位** |
| **23bit** | **2bit** |
| 无锁 | 对象hashcode | | 对象分代年龄 | 0 | 01 |
| 轻量级锁 | 指向栈中锁记录（Monitor Record）的指针 | | | | 00 |
| 重量级锁 | 指向互斥量（重量级锁）的指针 | | | | 10 |
| GC标记 | 空 | | | | 11 |
| 偏向锁 | 线程ID | | 对象分代年龄 | 1 | 01 |

|  |
| --- |
| 问题1：当有锁状态时，原先的hashcode和分代年龄存放何处？是否是在解锁之后重新计算？  加锁时，线程将对象Mark Word存放到线程栈桢中的存储空间，称为：Displaced Mark Word；  解锁时，通过持有锁线程栈桢中存放的Mark Word替换回来； |

|  |
| --- |
| 问题2：轻量级锁和重量级锁存放指针的长度仅有30位，指针通常不是有32位，如何存放？ |

#### 64位JVM

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 锁状态 | 25bit | 31bit | 1bit  cms\_free | 4bit  分代年龄 | 1bit  偏向锁 | 2bit  锁标志位 |
| 无锁 | 未使用 | hashcode |  |  | 0 | 01 |
| 偏向锁 | 线程ID(54bit) | Epoch（2bit） |  |  | 1 | 01 |

## Monitor Record

Monitor Record是线程私有的数据结构，每一个线程都有一个可用monitor record列表，同时还有一个全局的可用列表；那么这些monitor record有什么用呢？每一个被锁住的对象都会和一个monitor record关联（对象头中的LockWord指向monitor record的起始地址，由于这个地址是8byte对齐的所以LockWord的最低三位可以用来作为状态位），同时monitor record中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。

|  |
| --- |
|  |

1. Owner：初始时为NULL表示当前没有任何线程拥有该monitor record，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时又设置为NULL；
2. EntryQ:关联一个系统互斥锁（semaphore），阻塞所有试图锁住monitor record失败的线程；
3. RcThis:表示blocked或waiting在该monitor record上的所有线程的个数；
4. Nest:用来实现重入锁的计数；
5. HashCode:保存从对象头拷贝过来的HashCode值（可能还包含GC age）；
6. Candidate:用来避免不必要的阻塞或等待线程唤醒，因为每一次只有一个线程能够成功拥有锁，如果每次前一个释放锁的线程唤醒所有正在阻塞或等待的线程，会引起不必要的上下文切换（从阻塞到就绪然后因为竞争锁失败又被阻塞）从而导致性能严重下降。Candidate只有两种可能的值：0表示没有需要唤醒的线程，1表示要唤醒一个继任线程来竞争锁；

## 锁升级

在JVM中monitorenter和monitorexit字节码依赖于底层的操作系统的Mutex Lock来实现的，但是由于使用Mutex Lock需要将当前线程挂起并从用户态切换到内核态来执行，这种切换的代价是非常昂贵的。在java 1.6中，为了减少使用重量级锁（互斥锁）带来的性能消耗，引入了引入了大量的优化，如锁粗化（Lock Coarsening）、锁消除（Lock Elimination）、轻量级锁（Lightweight Locking）、偏向锁（Biased Locking）、适应性自旋（Adaptive Spinning）等技术来减少锁操作的开销。

|  |
| --- |
| 问题：如何理解用户态和内核态？  **概念：**  用户态，普通线程执行时，所处的状态；  内核态，当需要调用系统内核资源时，进程需要由用户态切换成内核态；  **转换条件：**  系统调用，主动，比如fork调用；  异常，被动，执行程序时，遇到不可知异常；  外围设备中断，被动，cpu需要暂停当前任务，执行中断处理程序； |

1. **锁粗化（Lock Coarsening）：**减少不必要的紧连在一起的unlock，lock操作，将多个连续的锁扩展成一个范围更大的锁；
2. **锁消除（Lock Elimination）：**通过运行时JIT编译器的逃逸分析来消除一些没有在当前同步块以外被其他线程共享的数据的锁保护，通过逃逸分析也可以在线程本地Stack上进行对象空间的分配（同时还可以减少Heap上的垃圾收集开销）；
3. **轻量级锁（Lightweight Locking）：**这种锁实现的背后基于这样一种假设，即在真实的情况下我们程序中的大部分同步代码一般都处于无锁竞争状态（即单线程执行环境），在无锁竞争的情况下完全可以避免调用操作系统层面的重量级互斥锁，取而代之的是在monitorenter和monitorexit中只需要依靠一条CAS原子指令就可以完成锁的获取及释放。当存在锁竞争的情况下，执行CAS指令失败的线程将调用操作系统互斥锁进入到阻塞状态，当锁被释放的时候被唤醒（具体处理步骤下面详细讨论）。
4. **偏向锁（Biased Locking）：**是为了在无锁竞争的情况下避免在锁获取过程中执行不必要的CAS原子指令，因为CAS原子指令虽然相对于重量级锁来说开销比较小但还是存在非常可观的本地延迟（可参考这篇[文章](https://blogs.oracle.com/dave/entry/biased_locking_in_hotspot)）。
5. **适应性自旋（Adaptive Spinning）：**当线程在获取轻量级锁的过程中执行CAS操作失败时，在进入与monitor相关联的操作系统重量级锁（mutex semaphore）前会进入忙等待（Spinning）然后再次尝试，当尝试一定的次数后如果仍然没有成功则调用与该monitor关联的semaphore（即互斥锁）进入到阻塞状态。

JVM使用锁标志位标志对象的锁状态；状态值依次是：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的加剧而主键升级；当升级值重量级锁状态时，将不会降级；

|  |
| --- |
|  |

### 偏向锁

1. **问题：**大多数情况下，锁不仅不存在竞争，而且总是由同一线程多次获得；
2. **目的：**减少同一线程获取锁和释放锁带来的性能消耗；
3. **原理：**在Mark Word中记录持有偏向锁的线程ID，当线程尝试获取锁时，检查Mark Word中存放的线程ID是否是当前线程，如果是，则表示直接获取锁，无须进行CAS操作来加锁、解锁；
4. **加锁：**即线程进入synchronized同步代码，但此时并没有线程与之竞争，则默认使用偏向锁；
5. **撤销：**当线程出现竞争时，偏向锁会撤销，此时对象为无锁状态或者转而使用更加”严格”的锁机制；撤销需要等待全局安全点（即这个时间点没有执行字节码）；
6. **解锁：**偏向锁解锁过程很简单，只需要测试下是否Object上的偏向锁模式是否还存在，如果存在则解锁成功不需要任何其他额外的操作；

|  |
| --- |
|  |

|  |
| --- |
| 问题：epoch是什么？ |

### 轻量级锁

1. **加锁：**在执行同步代码之前，JVM会在当前线程的栈桢中创建用于存储锁记录的空间，并将对象头中的Mark Word复制到所记录中，然后线程尝试使用CAS将对象头中的Mark Word替换为指向所记录的指针；如果成功，则当前线程获得锁；失败，则表示存在竞争，当前线程尝试使用自旋的方式获取锁；
2. **解锁：**使用CAS操作，将线程栈桢中存放的Mark Word替换回对象头中；如果失败，则表示存在竞争，锁会膨胀为重量级锁；

|  |
| --- |
|  |

|  |
| --- |
| 问题：偏向锁如何变为轻量级锁？  （1）初始时对象处于biasable状态，并且ThreadID为0即biasable & unbiased状态（这里不讨论epoch和age）  （2）当一个线程试图锁住一个处于biasable & unbiased状态的对象时，通过一个CAS将自己的ThreadID放置到Mark Word中相应的位置，如果CAS操作成功进入第（3）步否则进入（4）步  （3）当进入到这一步时代表当前没有锁竞争，Object继续保持biasable状态，但是这时ThreadID字段被设置成了偏向锁所有者的ID，然后进入到第（6）步  （4）当前线程执行CAS获取偏向锁失败（这一步是偏向锁的关键），表示在该锁对象上存在竞争并且这个时候另外一个线程获得偏向锁所有权。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，并从偏向锁所有者的私有Monitor Record列表中获取一个空闲的记录，并将Object设置为LightWeight Lock状态并且Mark Word中的LockRecord指向刚才持有偏向锁线程的Monitor record，最后被阻塞在安全点的线程被释放，进入到轻量级锁的执行路径中，同时被撤销偏向锁的线程继续往下执行同步代码。  （5）当一个线程试图锁住一个处于biasable & biased并且ThreadID不等于自己的ID时，这时由于存在锁竞争必须进入到第（4）步来撤销偏向锁。  （6）运行同步代码块 |

### 重量级锁

1. 重量级锁，即为所有线程都会阻塞等待，直到持有锁的线程释放锁后，才进行新一轮的竞争；
2. 因为自旋锁会消耗CPU，为了避免无用的消耗，锁升级重量级锁后，就无法降级为轻量级锁；此时，未持有锁的线程在尝试获取锁时，都会进入阻塞状态，等到锁释放后才会被唤醒，进行新一轮的争夺；

### 锁升级过程

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
| 问题：重量级锁解锁后，对象是否还能够回到偏向锁状态？ |

### 锁的优缺点

|  |  |  |  |
| --- | --- | --- | --- |
| **锁** | **优点** | **缺点** | **适用场景** |
| 偏向锁 | 加锁和解锁不需要额外消耗，和执行非同步方法相比仅存在纳秒级差距； | 如果线程间存在锁竞争，会带来额外的锁撤销消耗； | 适用于仅有一个线程访问同步块的场景； |
| 轻量级锁 | 竞争的线程不会阻塞，提高了程序的响应速度； | 始终得不到锁竞争的线程，使用自旋会消耗CPU； | 追求响应时间，同步块执行非常块； |
| 重量级锁 | 线程竞争不使用自旋，不会消耗CPU； | 线程阻塞，响应时间缓慢； | 追求吞吐量，同步块执行速度较长； |

# 原子操作实现原理

## cpu术语

|  |  |
| --- | --- |
| **Cpu术语** | **描述** |
| Cache line | 缓存行，高速缓存操作的最小单位； |
| Compare And Swap | CAS，比较并交换；操作包含2个值：期望操作前的值，新值；执行CAS操作时，先比较期望值与当前值是否相等，如果相等，则执行交换，设置新值；否则，不进行交换； |
| CPU pipeline | Cpu流水线，将一条指令的执行分为若干个过程，放在CPU不同的电路单元中执行，虽然完整一条指令需要若干个CPU时钟周期，但是多条指令同时执行，则平均单条执行可以在一个CPU时钟周期内完成； |
| Memory Order Violation | 内存顺序冲突，由假共享引起，假共享是指多个cpu同时修改同一个缓存行的不同部分引起其中一个cpu的操作无效；当出现内存顺序冲突后，必须要清空cpu流水线； |

## 处理器如何实现原子操作

1. 处理器保证从内存中读写一个字节是原子的，当一个处理器访问一个字节时，其余处理器不能够访问这个字节的内存地址；
2. 多处理器的原子操作，通过对缓存加锁或者总线加锁来保证；
3. 访问跨缓存行、跨总线宽度、跨页表访问的原子操作，通过对缓存加锁或者总线加锁来保证；

## Java如何实现原子操作

1. Java通过锁机制和循环CAS方式实现原子操作；
2. CAS操作使用处理器提供的CMPXCHG指令实现；
3. CAS实现原子操作的缺点：
   1. ABA问题：通过引入版本号来解决；
   2. 自旋带来的CPU消耗；
   3. 只能保证一个共享变量的原子操作：多个对象放入到一个对象中操作；比如：AtomicReference类；
4. 锁机制：除偏向锁外，其余锁都通过CAS+自旋来加锁和解锁；

# 运维经验

1. 通过jstack分析jvm中线程的运行状态，查看是否有死锁（BLOCKED）或者大量等待（WAITING）线程；