Java并发编程的艺术

|  |  |
| --- | --- |
|  | 🢂 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？   |  | | --- | |  | |

目录

[1 并发编程的挑战 3](#_Toc446352694)

[1.1 如何减少上下文切换？ 3](#_Toc446352695)

[1.1.1 无锁并发线程： 3](#_Toc446352696)

[1.1.2 使用CAS算法 3](#_Toc446352697)

[1.1.3 使用最少线程 3](#_Toc446352698)

[1.1.4 使用协程 3](#_Toc446352699)

[1.2 如何避免死锁？ 3](#_Toc446352700)

[2 Java并发机制的底层实现原理 4](#_Toc446352701)

[2.1 volatile的作用 4](#_Toc446352702)

[2.2 cpu术语 4](#_Toc446352703)

[2.2.1 内存屏障（memory barriers） 4](#_Toc446352704)

[2.2.2 缓冲行（cache line） 4](#_Toc446352705)

[2.2.3 原子操作（atomic operations） 4](#_Toc446352706)

[2.2.4 缓存行填充（cache line fill） 4](#_Toc446352707)

[2.2.5 缓存命中（cache hit） 4](#_Toc446352708)

[2.2.6 写命中（write hit） 4](#_Toc446352709)

[2.2.7 写缺失（write misses the cache） 4](#_Toc446352710)

[2.3 volatile的实现原理 5](#_Toc446352711)

[2.3.1 lock指令 5](#_Toc446352712)

[2.3.2 缓存一致性协议 5](#_Toc446352713)

[2.3.3 volatile的实现基础 5](#_Toc446352714)

[2.3.4 volatile使用优化（通过追加字节提升性能） 5](#_Toc446352715)

[2.3.4.1 不需要将volatile变量追加到64字节的情况 6](#_Toc446352716)

# 并发编程的挑战

并发程序并不一定比串行程序快，这取决于问题的规模以及资源共享等因素；考量并发程序的性能因素主要包括如下：

1. 上下文切换；
2. 资源限制；

## 如何减少上下文切换？

### 无锁并发线程：

锁的使用会引起线程的上下文切换，造成额外的性能损耗，因此尽可能避免使用锁；设计计程序时，尽可能的使线程之间的数据隔离，避免使用锁；可以通过将数据分段、归类，不同线程处理不同类别的数据，减少线程间的数据共享；举例：通过hash算法对数据进行分段取模，不同的线程处理不同段的数据；

### 使用CAS算法

1. Compare And Swap，一种无锁算法，属于乐观锁技术，先获取旧数据进行备份，再根据旧数据计算新数据，最后比较备份&旧数据的值，如果一致，则更新；如果不一致，则说明旧数据发生过改变，不做更新，返回操作失败的消息；
2. 当多个线程同时使用CAS更新同一个变量时，只有其中一个线程能更新变量的值，剩余其它线程均会失败；失败的线程并不会挂起，而是被告知更新失败，能够再次尝试；
3. 自旋CAS，就是线程一致循环尝试CAS，直到操作成功为止；
4. 因为属于乐观锁技术，假设更新的数据无变化，所以对于线程竞争较少的情况，能够具备较好的性能；
5. java的Atomic包中的工具类就是使用CAS实现的；
6. 一般计算机底层架构都支持CAS指令，对于不支持该指令的平台，JVM使用自旋锁实现CAS操作；

### 使用最少线程

根据问题的规模选择合适的线程数，避免大量线程空闲线程处于等待状态；

### 使用协程

在单线程里实现多任务的调度，并在单线程里维护多个任务间的切换；例如：反应堆模式；

## 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，尽量保证每个锁只占用一个资源；
3. 尝试使用定时锁，能够利用超时机制跳出阻塞状态；
4. 对于数据库锁，加锁和解锁需要在一个数据库连接里，否则会出现解锁失败的情况；

# Java并发机制的底层实现原理

Java字节码🡪汇编指令

## volatile的作用

1. 在多处理器开发中保证了变量的可见性；当一个线程修改共享变量时，另一个线程能够读到修改后的变量；
2. 不会引起线程上下文的调度，因此比synchronized更加轻量；
3. 如果一个字段被声明为volatile，java线程内存模型确保所有线程看到这个变量的值是一致的；

## cpu术语

### 内存屏障（memory barriers）

一组处理器指令，用于实现对内存操作的顺序限制；

### 缓冲行（cache line）

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（atomic operations）

不可中断的一个或者一系列操作；

### 缓存行填充（cache line fill）

用来将数据从内存填充到合适缓存中的动作；缓存可以是L1/L2/L3或者全部；

### 缓存命中（cache hit）

如果进行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时，处理器从缓存中读取操作数，而不是从内存中读取；

### 写命中（write hit）

当处理器将操作数写回到一个内存缓存的区域时，会先检查这个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缓存，而不是直接回写到内存；这个过程称为写命中；

### 写缺失（write misses the cache）

一个有效的缓存行被写入到不存在的内存区域；

## volatile的实现原理

### lock指令

添加volatile的关键字会在编译时，多执行一个lock指令，lock指令的作用如下：

1. 将当前处理器缓存行的数据写回到系统内存；
2. 使cpu中相应数据的缓存地址失效；

|  |  |
| --- | --- |
|  | 注：CPU高速缓存 |
|  | **为了提高性能，cpu中提供了多级缓存，cpu不直接和内存通信，而是先将数据从内存读入到缓存中，然后再与缓存交互；但数据被操作完后，不知道什么时候会被写到内存；lock指令能够使cpu操作缓存数据后，立刻写入到内存中；** |

### **缓存一致性协议**

1. 当cpu将数据写入到内存中后，其余cpu的缓存中可能也缓存了数据，并不知道内存已经更新，导致数据的不一致；
2. 缓存一致性协议保证了cpu各个处理器缓存的一致性；
3. 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，会重新从内存中读取数据到缓存中；

### volatile的实现基础

1. Lock前缀指令会引起处理器缓存写回到内存；处理器执行写回操作需要确保能够独占共享内存；可以通过锁定缓存、锁定总线的方式实现；锁定总线的方式开销较大，通常锁定缓存；缓存锁定，通过锁定共享数据所在的缓存区域，确保处理器独占共享数据；总线锁定，通过锁定整个总线，确保处理器独占共享数据；
2. 一个处理器的缓存回写到内存会导致其它处理器的缓存无效；

### volatile使用优化（通过追加字节提升性能）

高速缓存行是64个字宽，LinkedTransferQueue将单个节点占用内存追加到64个字节，从而保证队列头和队列尾分别处于不同的缓存行，从而保证了volatile关键字不会同时锁定队列头和队列尾；当一个处理器修改队列头时，会锁定队列头所在的缓存行，因为与队列尾不在同一个缓存行，所以不影响其它处理器访问队列尾；从而提升操作效率；

#### 不需要将volatile变量追加到64字节的情况

1. 处理器缓存行不是64字宽；
2. 共享变量不会被频繁的写；不频繁写，则不需要频繁对共享缓存加锁，故追加字节反而带来了性能的消耗；
3. java7+可能会淘汰或者重排列无用字段，故追加字节的方式可能不会生效；