|  |
| --- |
|  |
| Java 并发编程 |
|  |

|  |
| --- |
| admin  [选取日期] |

Sleep操作不会释放锁

# Java并发编程

## 并发编程的挑战

## 上下文切换

单线程处理器也支持多线程执行代码，CPU通过给每个线程分配CPU时间片来实现这个机制，切换前会保存上一个任务的状态，每次上下文切换会一定程度上影响多线程的执行速。

考虑到线程的创建和上下文切换的开销，多线程编程不一定比单线程快。

查看上下文切换次数和时长的工具，vmstat查看上下文切换的次数，Lmbench3查看上下文切换的时长。

### 如何减少上下文切换

1. 无锁并发编程

多线程竞争锁时会引起上下文切换，所以可以用一些方法避免使用锁。

比如：将数据的ID按照HASH算法取模分段，不同的线程处理不同段的数据。

1. CAS算法

Java的Atomic包中的CAS算法

1. 使用最小线程
2. 协程

在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换

### 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，尽量保证每个锁只占用一个资源；
3. 尝试使用定时锁，使用lock.tryLock(timeout)来替代使用内部锁机制；
4. 对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况。

## 资源限制的挑战

资源限制是指在进行并发编程时，程序的执行速度受限于计算机硬件和软件资源。（如数磁盘读写速度，socket连接数等）

可以考虑使用集群并行执行程序，使用ODPS、Hadoop或者自己搭建服务器集群，不同的机器处理不同的数据。

要根据不同的资源限制调整程序的并发度。

## Java并发机制的底层实现原理

### 2.1 volatile的应用

Volatile是轻量级的synchronized，Java内存模型保证所有线程看到这个变量的值是一致的。

|  |  |  |
| --- | --- | --- |
| 术语 |  | 描述 |
| 内存屏障 | Memory barriers | 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，现代CPU需要执行几百次CPU指令 |
| 原子操作 | Atomic operations | 不可中断的一个或一系列操作 |
| 缓存行填充 | Cache line fill | 当处理器识别到内存中读取操作数是可缓存的，处理器读取整个高速缓存到适当的缓存 |
| 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，首先检查这个缓存的内存地址是否在缓存行中，如果存在一个有效的缓存行，则处理器将这个操作数写回到缓存，而不是写回到内存 |
| 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的两条实现原则：

1. **Lock前缀指令会引起处理器缓存回写到内存**
2. **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**

实现缓存一致性协议，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否过期，若过期就使缓存行无效并进行缓存行填充

#### 2.1.1 Volatile的使用优化

JDK7的并发包中有一个队列集合类Linked-TransferQueue，在使用Volatile时，用一种**追加字节**的方式来优化队列出队和入队的性能。将队列的头结点和尾节点扩充到64位。

大部分处理器的高速缓存行是64个字节宽且不支持部分填充缓存行，所以如果队列的头结点和尾节点都不足64字节的话，处理器会将他们都读到同一个高速缓存行中，这样修改头节点或尾节点时会将整个缓存行锁定，严重影响到队列的入队和出队效率。填满高速缓冲区的缓存行，避免头节点和尾节点加载到同一个缓存行。

当然当处理器的缓存行非64字节宽或者共享变量不会被频繁的读写的情况下不需要使用这种优化。

### Synchronized的实现原理和应用

* 对于普通方法同步，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

synchronized锁存在Java对象头中，锁分为四种状态：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。

1. **偏向锁**

当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程进入和退出同步块时不需要进行CAS操作来加锁和解锁，只需简单的测试一下对象头里的Mark Word里是否存储着指向当前线程的偏向锁。

偏向锁的撤销使用了一种等到竞争出现才释放锁的机制，需要等待全局安全点，暂停拥有偏向锁的进程，检查是否具有无锁状态。

1. **轻量级锁**

执行同步块之前，JVM会在当前线程的栈帧中创建用于存储锁记录的空间，并将对象头中的Mark Word赋值到所记录中，称为Displaced Mark Word，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。如果失败，则尝试使用自旋来获得锁

解锁时，会使用CAS操作将Displaced Mark Word替换回到对象头，如果失败，锁就会膨胀成重量级锁。

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 适用场景 |
| 偏向锁 | 加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距 | 如果线程间存在锁竞争，会带来额外的锁撤销的消耗 | 适用于只有一个线程访问同步块情景 |
| 轻量级锁 | 竞争的线程不会阻塞，提高了程序的响应速度 | 如果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，同步块执行速度快 |
| 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，同步快执行速度较长 |

### 原子操作的实现原理

Atomic operation意为不可中断的一个或一系列操作。

|  |  |  |
| --- | --- | --- |
| 术语名称 | 英文 | 解释 |
| 缓存行 | Cache line | 缓存的最小操作单位 |
| 比较并交换 | Compare and Swap |  |
| CPU流水线 | CPU pipeline | CPU中由几个不同功能的电路单元组成一条指令处理流水线 |
| 内存顺序冲突 | Memory order violation | 内存顺序冲突一般是由**假共享**引起的，假共享指多个CPU同时修改同一个缓存行的不同部分而引起其中一个CPU的操作无效，当出现内存顺序冲突时，CPU必须清空流水线 |

#### 处理器如何实现原子操作

首先处理器会保证基本的内存操作的原子性。保证从系统内存中读取或写入一个字节是原子的。但是对于复杂的内存操作处理器是不能保证其原子性的，需要使用总线锁定和缓存锁定两个机制。

1. **总线锁定**

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，其他处理器的请求将被阻塞住。

1. **缓存锁定**

内存区域如果被缓存在处理器的缓存行中，并且在Lock操作期间被锁定，那么当它执行锁操作回写到内存时，处理器不在总线上声明LOCK#信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性。

#### Java中如何实现院子操作

1. **锁**

偏向锁、轻量级锁、互斥锁

JVM使用锁的方式都使用了循环CAS

1. **循环CAS**

CAS需要在操作值的时候，检查值有没有发生变化，如果没有变化则更新。

需要注意

1. ABA问题，一个值原来是A变为B由变为A。解决方法是加入版本号如1A2B3A
2. 循环时间长开销大
3. 只能保证一个共享变量的原子操作

## Java内存模型

### 3.1 Java内存模型的基础

#### 3.1.1 Java内存模型的抽象结构

在命令式编程中，线程之间的通信机制有两种：共享内存和消息传递。

抽象角度来看，线程之间的共享变量存储在**主内存**（Main Memory）中，每个线程都有一个私有的**本地内存**（Local Memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。涵盖了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。

#### 3.1.2 从源代码到指令序列的重排序

1. **编译器优化的重排序**

编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序；

1. **指令级并行的重排序**

现代处理器采用了指令级并行技术（Instruction-Level parallelism，ILP）来将多条指令重叠执行，如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

1. **内存系统的重排序**

由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

#### 3.1.3 并发编程模型的分类

现代的处理器使用写缓冲区临时保存向内存写入的数据，写缓冲区可以保证指令流水线持续运行，避免由于处理器停顿下来等待向内存写入数据而产生的延迟。同时，以批处理的方式刷新写缓冲区，以及合并写缓冲区中对同一内存地址的多次写，减少对内存总线的占用。

**处理器对内存的读/写操作的执行顺序，不一定与内存实际发生的读/写操作顺序一致！**

|  |  |  |
| --- | --- | --- |
|  | Processor A | Processor B |
| 代码 | a=1;  x=b; | b = 2;  y = a; |
| 运行结果 | 初始状态：a=b=0  处理器允许执行后得到结果：x=y=0  原因是a、b被写入了Local Area缓冲区，而读取是直接在共享内存中读取的 | |

为了保证内存的可见性，Java编译器在生成指令序列的适当位置会插入内存屏障指令来禁止特定类型的处理器重排序。

#### 3.1.4 happens-before简介

如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系。

规则：

* **程序顺序规则**

一个线程中的每个操作，happens-before于该线程中的任意后续操作

* **监视器锁规则**

对一个锁的解锁，happens-before于随后对这个锁的加锁

* **Volatile变量规则**

对一个volatile域的写，happens-before于任意后续对这个volatile域的读写

* **传递性**

如果A happens-before B，且B happens-before C，那么A hannpns-before C

两个操作之间具有happens-before关系，并不意味着一前一个操作要在后一个操作之前执行，仅仅要求前一个操作对后一个操作课件，且前一个操作按顺序排在第二个操作之前。

### 3.2 重排序

重排序是指编译器和处理器为了优化程序性能而对指令序列进行重新排序的一种手段。

#### 3.2.1 数据依赖性

如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间就存在数据依赖性。

编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序，这里的数据依赖性仅针对单个处理器中执行的指令序列和单个序列中执行的操作。

#### 3.2.2 as-if-serial语义

在单线程下，无论如何排序，程序执行结果不能被改变。

这样使得单线程程序员无需担心重排序会干扰他们，也不许担心内存可见性问题。

#### 3.2.3 控制依赖关系

If和if语句中操作具有控制依赖关系，当代码中存在控制依赖关系时，会影响指令序列执行的并行度。为此编译器和处理器会采用**猜测Speculation**执行来克服控制相关性对并行度的影响。比如可以先执行if的执行语句，将其结果保存到**重排序缓存Reorder Buffer中**，再执行if判断语句，判断是否应用这个变更。

### 3.3 顺序一致性 Sequentially Consistent

#### 3.3.1 数据竞争

多个线程间没有通过同步来进行排序。

#### 3.3.2 顺序一致性内存模型

两大特性：

1. 一个线程中的所有操作必须按照程序的顺序来执行
2. 所有线程都只能看到一个单一的操作执行顺序。在顺序一致性内存模型中，每个操作都必须原子执行且立刻对所有线程可见

概念上，顺序一致性模型有一个单一的全局内存，这个内存通过一个左右摆动的开关可以连接到任意一个线程，同时每一个线程必须按照程序的顺序来执行内存读写操作。

顺序一致性模型与JMM内存模型不是一个模型！JMM不保证未同步程序的执行结果与该程序在顺序一致性模型中的执行结果一致，因为如果想要保证执行结果一致，JMM需要禁止大量的处理器和编译器的优化，这对程序的执行性能会产生很大的影响。

#### 3.3.3 Volatile

当读一个Volatile变量时，JMM会把该线程对应的本地内存置为无效，线程接下来将从主内存中读取共享变量。

线程A写入一个volatile变量，随后线程B读这个volatile变量，这个过程实质上是线程A通过主内存向线程B发送消息。

### 锁的内存语义

锁除了让临界区互斥执行外，还可以让释放锁的线程向获取同一个锁的线程发送消息。

锁释放-获取的内存语义的实现至少有下面两种方式：

1. 利用volatile变量的写-读所具有的内存语义
2. 利用CAS所附带的volatile读和volatile写的内存语义

#### 3.4.1 Double-check Locking

注意double check Locking时会引发错误

Public static Instance getInstance(){

If(instance == null){

If(instance==null){

Synchronized(this){

If(instance==null){

Instance = new Instance(); //会引发错误，除非将Instance改为volatile

}

}

}

}

}

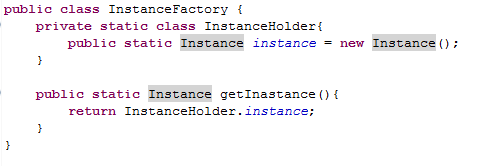
解决方法：

1. **使用volatile进行解决**

在上面的instance上加入Instance修饰符

除了可以对静态字段实现延迟初始化，还可以对实例字段实现延迟初始化

1. **基于类初始化的解决方案**



##### 类的初始化

根据Java语言规范，在首次发生下列任意一种情况时，一个类或接口类型T将立即被初始化：

1. T是一个类，而且一个T类型的实例被创建
2. T是一个类，且T中声明的一个静态方法被调用
3. T中声明的一个静态字段被赋值
4. T中声明的一个静态字段被使用，而且这个字段不是一个常量字段
5. T是一个顶级类，且一个断言语句嵌套在T内部被执行

对于每一个类或接口C，都有一个唯一的**初始化锁LC**与之对应，每个线程至少获取一次锁来确保这个类已经被初始化过了。

### 3.4 Final域的内存语义

对于final域，编译器和处理器要遵守两个重排序规则：

1. 在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序
2. 初次读一个包含final域的对象引用，与随后初次读这个final域，这两个操作之间不能重排序

### happens-before

JMM一方面要为程序员提供足够强的内存,另一方面，对编译器和处理器的限制要尽可能的放松。

### 处理器的内存模型

根据对不同类型的读、写操作组合的执行顺序的放松，可以把常见处理器的内存模型划分为如下几种：

1. **TSO**

放松程序中写-读操作的顺序，由此产生了Total Store Ordering内存模型

1. **PSO**

继续放松程序中写-写操作的顺序，由此产生了Relaxed Memory Order

1. **RMO,PowerPC**

继续放松程序中读-写和读-读操作的顺序，由此产生了Relaxed Memory Order内存模型和PowerPC内存模型

这里的读-写放松操作，是以两个操作之间不存在数据依赖性为前提的

越是追求执行性能的语言，内存模型设计的会越若

#### JMM的内存可见性保证

按程序类型，Java程序的内存可见性保证可以分为下列3类：

1. **单线程程序**

不会出现内存可见性问题，编译器、runtime和处理器会共同确保单线程程序的执行结果与该程序在顺序一致性模型中的执行结果相同

1. **正确同步的多线程程序**

正确同步的多线程程序将具有顺序一致性（程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同）

1. **未同步/未正确同步的多线程程序**

提供**最小安全性保障**：线程执行是要读取到的值，要么是之前某个线程写入的值，要么是默认值。

## Java并发编程基础

### 线程简介

#### 什么是线程

现代操作系统调度的最小单元是线程，也叫轻量级进程Light Weight Process。在一个进程里可以创建多个线程，这些线程都拥有各自的计数器、堆栈和局部变量等属性，并且能够访问共享的内存变量。

Java程序天生就是多线程程序：

1. **Signal Dispatcher**

分发处理发送给JVM信号的线程

1. **Finalizer**

调用对象finalize方法的线程

1. **Reference Handler**

清除Reference的线程

1. **Main**

main线程，用户程序入口

#### 为什么要使用多线程

1. **更多的处理器核心**
2. **更快的响应时间**
3. **更好的编程模型**

#### 线程优先级

理论上说，优先级高的线程分配时间片的数量要多于优先级低的线程。

但是线程优先级不能作为程序正确性的依赖，因为有的操作系统可以不理会Java线程对于优先级的设定。

#### 线程的状态

1. **NEW**

被构建但是未调用start方法

1. **RUNNABLE**

将操作系统中的就绪和运行两种状态称为Runnable

1. **BLOCKED**

线程阻塞于锁

1. **WAITING**

进入该状态表示当前线程需要等待其他线程作出一些特定动作

1. **TIME\_WAITING**

超时等待，可以在指定的时间自行返回

1. **TERMINATED**

终止状态，表示当前线程已经执行完毕

#### Daemon线程

Daemon线程是一种支持型线程，因为它主要被用作程序中后台调度以及支持性工作。这意味着当一个Java虚拟机中不存在非Daemon线程的时候，Java虚拟机将退出，可以通过Thread.setDaemon(true)将线程设置为Daemon线程。（需要在启动线程之前设置）

在构建Daemon线程识，不能依靠finally块中的内容来确保关闭或清理资源的逻辑。

### 终止和启动线程

#### 4.2.1 构造线程

新线程的构造是由其parent线程来进行空间分配的，而child线程继承了parent是否为Daemon、优先级和加载资源的contextClassLoader以及可继承的ThreadLocal，同时还会分配一个唯一的ID来标志这个child线程。

#### 启动线程

调用start（）：当前线程同步告知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。

启动线程前，最好为线程设置名称。

#### 理解中断

中断可以理解为线程的一个标志位属性，表示一个运行中的线程是否被其他线程进行了中断操作。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。

中断操作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。

#### 过期的suspend、resume、stop

suspend暂停，resume恢复，stop停止

但是因为在调用后线程不会释放已经占有的资源（比如锁），容易引发死锁问题。

#### 安全的终止线程

除了使用interrupt方法，还可以使用一个boolean变量来控制是否需要停止任务并终止该线程。

### 线程间通信

#### volatile和synchronized关键字

过多的使用volatile是不必要的，因为它会降低程序执行的效率。

对于同步块的实现使用了monitorenter和monitorexit指令，而同步方法则是依靠方法修饰符上的ACC\_SYNCHRONIZED来完成的。

#### 等待/通知机制

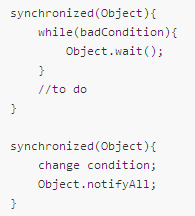
是指一个线程A调用了对象O的wait()方法进入等待状态，而另一个线程B调用了对象O的notify()或norifyAll()状态，线程A收到通知后进行后续操作。

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。

使用wait()、notify()和notifyAll()时需要注意的细节：

1. **使用时需要先对调用对象加锁**；
2. 调用wait()方法后，线程状态由RUNNING变为WAITING,并将当前线程放置到对象的等待队列；
3. Notify()或notifyAll()方法调用后，等待线程依旧不会从wait()返回，需要调用notify()或notifyAll()的线程释放锁后，等待线程才有机会从wait()返回；
4. notify方法将线程状态从WAITING变为BLOCKED
5. 从wait()方法返回的前提是获得了调用对象的锁

**等待通知的经典范式**



#### 管道输入/输出流

主要用于线程之间的数据传输，而传输的媒介为内存。

PipedOutputStream,PipedInputStream面向字节

PipedReader,PipedWriter面向字符

对于Piped类型的流，必须要先进行绑定，也就是调用connect()方法，如果没有进行绑定，那么对该流的访问将会抛出异常。

#### Thread.join的使用

如果一个线程执行了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。

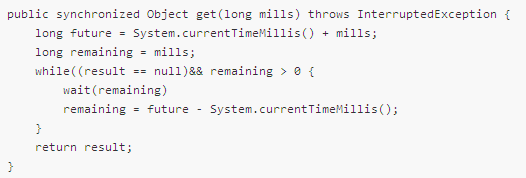
#### ThreadLocal的使用

线程变量，是一个以ThreadLocal对象为键，任意对象为值的存储结构。这个结构被附带在线程上，也就是说一个线程可以根据一个ThreadLocal对象查询到绑定在这个线程上的一个值。

ThreadLocal<T>

### 线程应用实例

1. 等待超时模式



CountDownLatch可以保证所有线程同步执行