# 并发编程的挑战

由于多线程需要创建和上下文切换的开销，因此不表示多线程就一定比单线程快。。

## 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，所以多线程处理数据时，可以用一些方法来避免使用锁，如果将数据id按照hash算法取模分段，不同线程处理不同段数据。

②CAS算法

--java的atomic包使用cas算法来更新数据，不需要加锁

③使用最少线程。

--避免创建不必要的线程

④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

## 死锁

|  |
| --- |
| **package** cn.huangwei.first;  **public** **class** DeadLock {  **private** **static** String *A* = "A";  **private** **static** String *B* = "B";  **private** **void** deadLock(){  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  **synchronized**(*A*){  **try** {  Thread.*sleep*(3000);//为了等待b线程把B锁住  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **synchronized**(*B*){  System.***out***.println("a");  }  }  }  }).start();  **new** Thread(**new** Runnable(){  @Override  **public** **void** run() {  **synchronized**(*B*){  **try** {  Thread.*sleep*(3000);//为了等待a线程把A锁住  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **synchronized**(*A*){  System.***out***.println("B");  }  }  }  }).start();    }  **public** **static** **void** main(String[] args){  **new** DeadLock().deadLock();  }  } |

避免死锁的常用方法：

①避免一个线程同时获取多个锁

②避免一个额线程在锁内同时占用多个资源，尽量保证每个锁占用一个资源

③使用定时锁，使用try.lock(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中，否则会出现解锁失败

死锁必要条件：

互斥、持有等待、不可剥夺、循环等待

# Java并发机制的底层实现原理

## Volatile的原理

### CPU缓存

按照读取顺序与CPU结合的紧密程度，CPU缓存可分为：

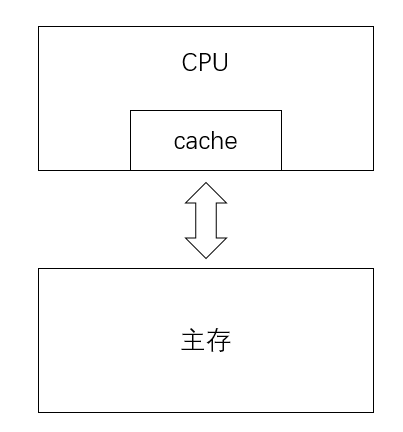
**一级缓存**：简称L1 Cache，位于CPU内核的旁边，是与CPU结合最为紧密的CPU缓存

**二级缓存**：简称L2 Cache，分内部和外部两种芯片，内部芯片二级缓存运行速度与主频相同，外部芯片二级缓存运行速度则只有主频的一半

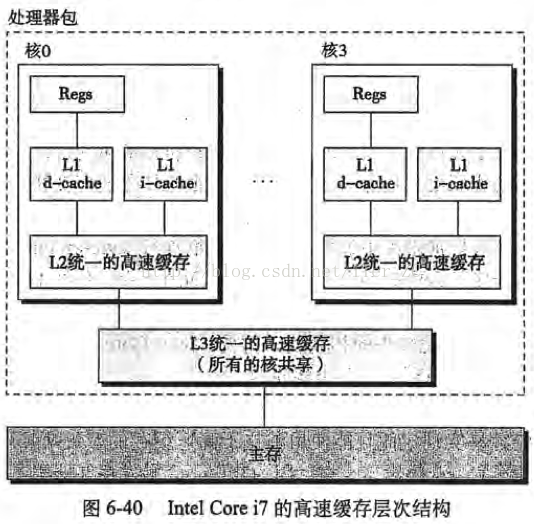
**三级缓存**：简称L3 Cache，部分高端CPU才有

当CPU要读取一个数据时，首先从**一级缓存**中查找，如果没有再从**二级缓存**中查找，如果还是没有再从**三级缓存**中或**内存**中查找。一般来说每级缓存的命中率大概都有80%左右，也就是说全部数据量的**80%**都可以在一级缓存中找到，只剩下20%的总数据量才需要从二级缓存、三级缓存或内存中读取。

**CPU–>CPU缓存–>主内存数据读取之间的关系**：



1. **程序以及数据被加载到主内存**
2. **指令和数据被加载到CPU缓存**
3. **CPU执行指令，把结果写到高速缓存**
4. **高速缓存中的数据写回主内存**



多核cpu思考一下情况：

①核0从主存中读取数据a到核0的缓存，同时核3也做了同样的事情

②核0修改了数据a，修改后的a写入了核0的缓存， 但是没有写回主存

③核3使用数据的时候，使用的是旧的a，出现问题

Cpu厂商：当一个CPU**修改缓存中**的字节时，服务器中**其他CPU会被通知**，它们的缓存将视为无效。于是，在上面的情况下，核3发现自己的**缓存中数据已无效**，核0将立即把自己的**数据写回主存**，然后核3**重新读取**该数据。

### Lock指令的工作

有volatile变量修饰的共享变量，进行写操作时，会多出一个lock代码，而该lock前缀在多核处理器下会引发两件事情。

①将当前处理器缓存行的数据写回系统内存

②写回操作使得在其他cpu缓存的该内存地址无效。

**将当前处理器缓存行的数据写回系统内存**

在修改内存操作时，使用LOCK前缀去调用加锁的读-修改-写操作

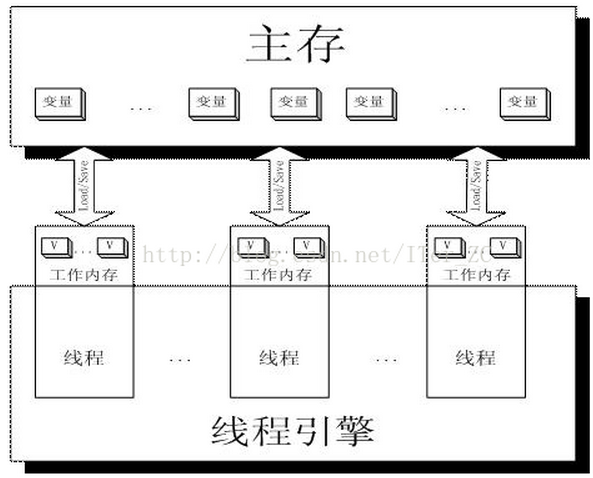
（1）在Pentium和早期的IA-32处理器中，LOCK前缀会使处理器执行当前指令时产生一个LOCK#信号，这种总是引起显式总线锁定出现

（2）在Pentium4、Inter Xeon和P6系列处理器中，如果内存访问有高速缓存且只影响一个**单独的高速缓存行**，那么操作中就会**调用高速缓存锁**，而**系统总线和系统内存**中的实际区域内**不会被锁定**。如果内存访问没有高速缓存且/或它跨越了高速缓存行的边界，那么这个处理器就会产生**LOCK#**信号，独占总线锁。由于在指令执行期间该**缓存行会一直被锁定**，其它处理器**无法读/写该指令要访问的内存区域**，因此能保证指令**执行的原子性**。这个操作过程叫做**缓存锁定**（cache locking）

**写回操作使得在其他cpu缓存的该内存地址无效**

多核处理器系统中进行写操作的时候，处理器能够**嗅探到其他处理器正在访问系统内存和它们的内部缓存**。嗅探技术能够保证它的内部缓存、系统内存和其他处理器的缓存的数据**在总线上一致**，即检测到其他处理器正在写内存地址，就会将自己的**相关缓存行失效**，等到下次访问的时候，读取新数据。

### 由lock指令会看volatile



工作内存Work Memory其实就是对**CPU寄存器和高速缓存的抽象**，或者说每个线程的工作内存也可以简单理解为**CPU寄存器和高速缓存**。

那么当写两条线程Thread-A与Thread-B同时操作主存中的一个volatile变量i时，**Thread-A写了变量i**，那么：

* Thread-A发出LOCK#指令
* 发出的LOCK#指令**锁总线（或锁缓存行）**，Thread-A向主存**回写最新修改的i，**同时让Thread-B高速缓存中的缓存行**内容失效**

**Thread-B读取变量i**，那么：

Thread-B发现对应地址的**缓存行被锁了，等待锁的释放**，等到ThreadA写入操作结束，释放锁，缓存一致性协议会保证它读取到最新的主存值。

**Volatile性质**

①保证可见性

|  |
| --- |
| public class VolatileTest {  Boolean volatile isStop = false;  public void test(){  Thread t1 = new Thread(){  public void run() {  isStop=true;  }  };  Thread t2 = new Thread(){  public void run() {  while (!isStop);  }  };  t2.start();  t1.start();  }  public static void main(String args[]) throws InterruptedException {  for (int i =0;i<25;i++){  new VolatileTest().test();  }  }  } |

如果不加volatile，这个线程永远无法结束，因为t1做的修改总是不会立即写回到主存，导致t2不能读取到修改后的值；加了volatile之后，首先修改isstop的值，然后立即写回主存，当回写主存的时候，使得其他线程工作内存中的值失效，由于修改操作时原子操作，在修改的过程中，t2发现主存中对应地址的缓存行被锁住，等待释放锁，t1执行完修改操作，写入主存后，由于t2缓存中的值已失效，那么就会重新读取主存中的值，从而得到最新的值，保证可见性。

②不保证原子性

对于i++；操作，实际有三个过程，read，modify，load三个步骤，即使i加了volatile，但是i++操作并不是原子操作；如果线程1和线程2同时读取i的值，假设为1；然后线程1read和modify之后停止i=2，不会影响线程2执行，因为没有写回内存，线程2执行read，modify，load操作，此时i内存中为2，其他线程的i的主存对应的缓存地址失效，由于该操作i++不是原子性操作,不可能重新读取内存中的值，因此i=2写入内存，出错

③保证部分有序性

1.a

2.b

3.volatile

4.c

5.d

lock前缀指令实际上相当于一个内存屏障，它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

## Synchronized的实现原理及其应用

普通同步方法，锁是当前实例对象

静态同步方法，锁是当前类的class字节码对象

对于同步块，就是synchronized配置的对象

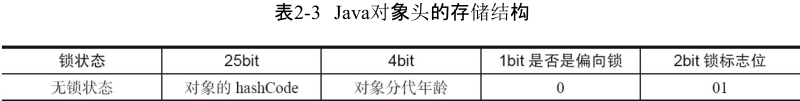
monitorenter指令是在编译后插入到同步代码块的开始位置，而monitorexit是插入到方法结束处和异常处，JVM要保证每个monitorenter必须有对应的monitorexit与之配对。任何对象都有一个monitor与之关联，当且一个monitor被持有后，它将处于锁定状态。线程执行到monitorenter指令时，将会尝试获取对象所对应的monitor的所有权，即尝试获得对象的锁。

### Java对象头

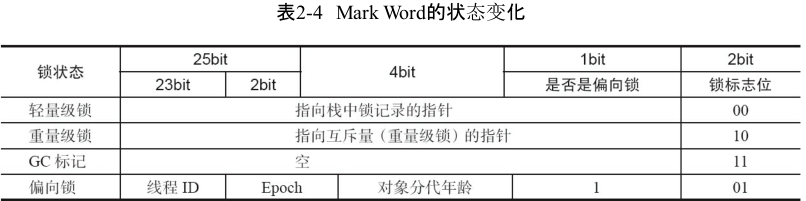
Synchronized用的锁是存在java对象头中的，如果数组类型，用3个字宽存储对象头，如果对象时非数组，则用2字宽存储对象头



Java对象头的mark word默认存储对象的hashcode，分代年龄，锁标记位



存储数据会随锁标志位变化



### 锁的升级与对比

锁的状态: 无锁状态、偏向锁状态、轻量级锁、重量级锁

**1.偏向锁**

当一个线程访问同步快并获取锁时，会在**对象头和栈帧中的锁记录里存储锁偏向的线程ID**，以后进入该线程和退出同步块时，不需要进行CAS操作来加锁和解锁

**偏向锁的撤销**

偏向锁是一种竞争出现才会释放锁的机制；偏向锁的撤销，需要等到全局安全点（时间点上没有正在执行的字节码）。

①暂停拥有偏向锁的线程

②检查持有偏向锁的线程是否活着

---2.1如果没有活着，将对象头设置为无锁状态，

---2.2如果活着，遍历偏向锁记录，要么重新偏向其他线程，要么将自己变为无锁状态，最后唤醒其他线程。

**偏向锁的获得过程：**

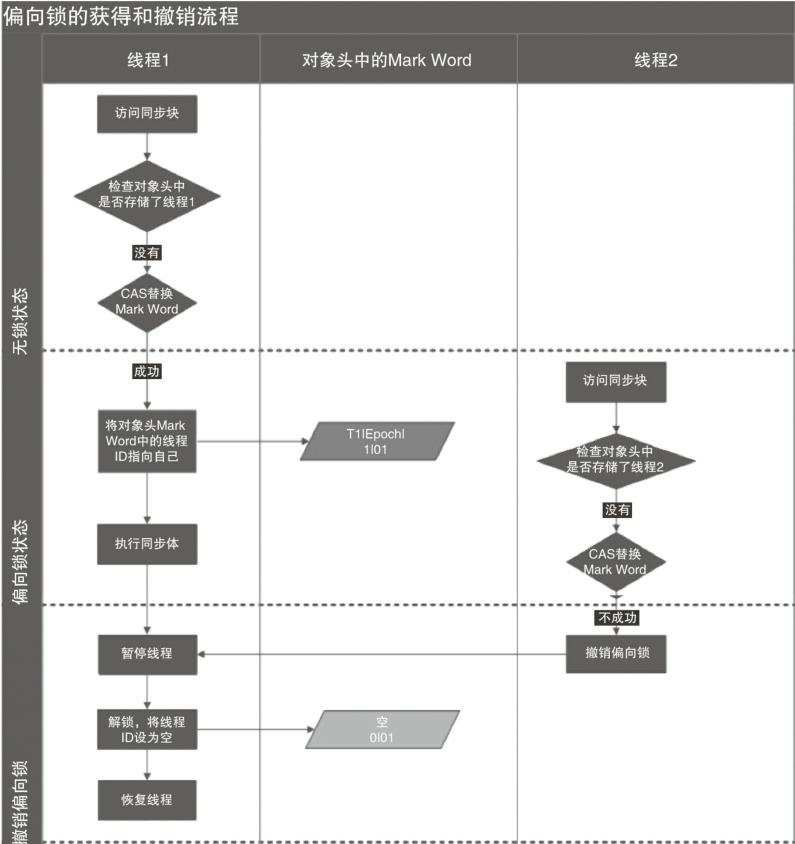
①判断markword是否存储指向当前线程的偏向锁，比较线程id和自身id是否一样

---1.1如果存在指向当前线程的偏向锁，执行同步代码

---1.2如果没有，使用cas竞争偏向锁

------1.2.1如果竞争成功，然后将线程id设置为自身ID，执行同步体

------1.2.2如果竞争失败，撤销偏向锁



**关闭偏向锁**

关闭偏向锁的启用延时：-XX:BiasedLockingStartupDelay=0

如果需要关闭：-XX:-UseBiasedLocking=false;

设置之后进入轻量级锁状态

**2.轻量级锁**

加锁过程：

①访问同步块

②在栈帧中创建存储锁记录的空间，并将mark word复制到锁记录中

③使用CAS修改markword

---3.1修改成功，替换为轻量级锁，执行同步体

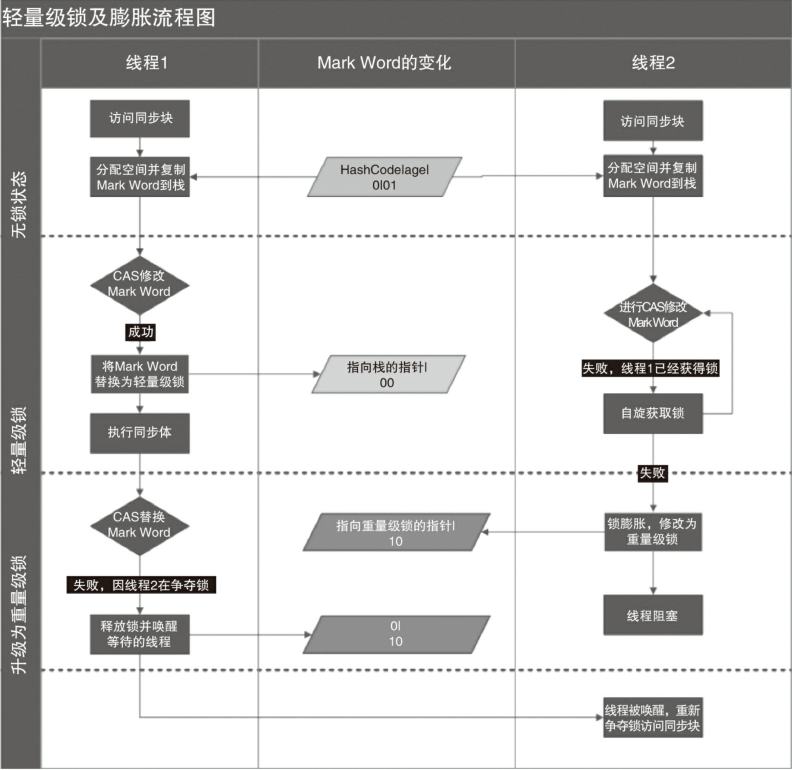
---3.2修改失败，表示其他线程竞争锁，自旋获得锁，锁膨胀为重量级锁，记录指向重量级锁的指针，线程阻塞等待所释放

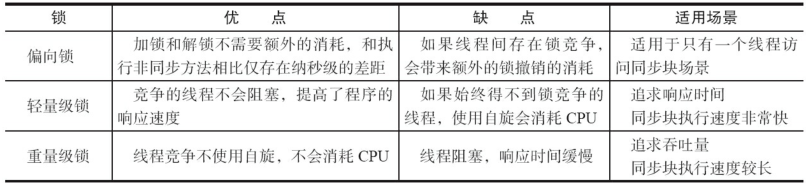
解锁过程：

①用cas替换markword为无锁状态

---1.1如果成功，没有竞争发生

---1.2如果失败，表示当前锁存在竞争，释放锁并唤醒等待线程





## 原子操作实现原理

①通过总线锁保证原子性

多个处理器想要对i++；进行操作，如i=1；CPU1和CPU2进行i++操作，期望结果为3，但是可能输出为2，原因是两个cpu读取的时候可能都读到了i=1；然后写入主存中；如果想要读改写的操作是原子的，要保证cpu1读改写共享变量的时候，cpu2不能操作该共享变量的内存地址的缓存，包括读改写。**解决**：可以使用总线锁，处理器输出lock#信号，其他处理器的请求被阻塞。

②缓存锁定保证原子性

缓存锁定：内存区域如果被缓存到处理器的缓存行中，并且在lock操作期间被锁定，那么它执行的锁操作写回内存时，使用缓存一致性机制来保证操作的原子性，因为缓存一致性机制会阻止两个以上处理器缓存同时修改内存区域，当有一个处理器回写被锁定的缓存行数据时，会使其他缓存行数据失效。两种请况不能使用缓存锁定：**操作数据不能被缓存在处理器内部和处理器不支持锁定。**

**③**java如何实现原子操作

Java中通过锁和循环CAS来实现原子操作，

①使用循环cas实现原子操作

|  |
| --- |
| Private void safeCount(){  For( ; ; ){  Int I = atomicI.get();  Boolean suc = atomicI.compareAndSet(i, i++);  If(suc){  Breakl;  }  }  } |

②CAS实现原子操作的三大问题

1. ABA

CAS的思路是，在操作值的时候，**检查值有没有发生变化，如果没有发生变化就更新**。但是如果一个值原来是A，变成B，又变成了A，那么CAS发现不了，就会出现错误的更新。解决思路：**使用版本号，在变量之前追加版本号，每次更新时候，版本号加1**，这么ABA，变为1A2B3A，atomic包提供一个**atomicStampedReference**来解决，首先校验当前引用是否等于与其引用，并且检查当前标志是否等于与其标志，如果全部相等，则以原子方式将该引用和该标志的值都更新为指定值。

1. 循环时间长且开销大

自旋cas如果不成功，就会给cpu带来很大开销

1. 只保证一个共享变量的原子操作

多个共享变量操作时，循环cas就无法保证操作原子性，这个时候，可以用锁解决，或者把多个共享变量装到一个对象中，atomicReference已经可保证对象之间的原子性