# 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（JMM）

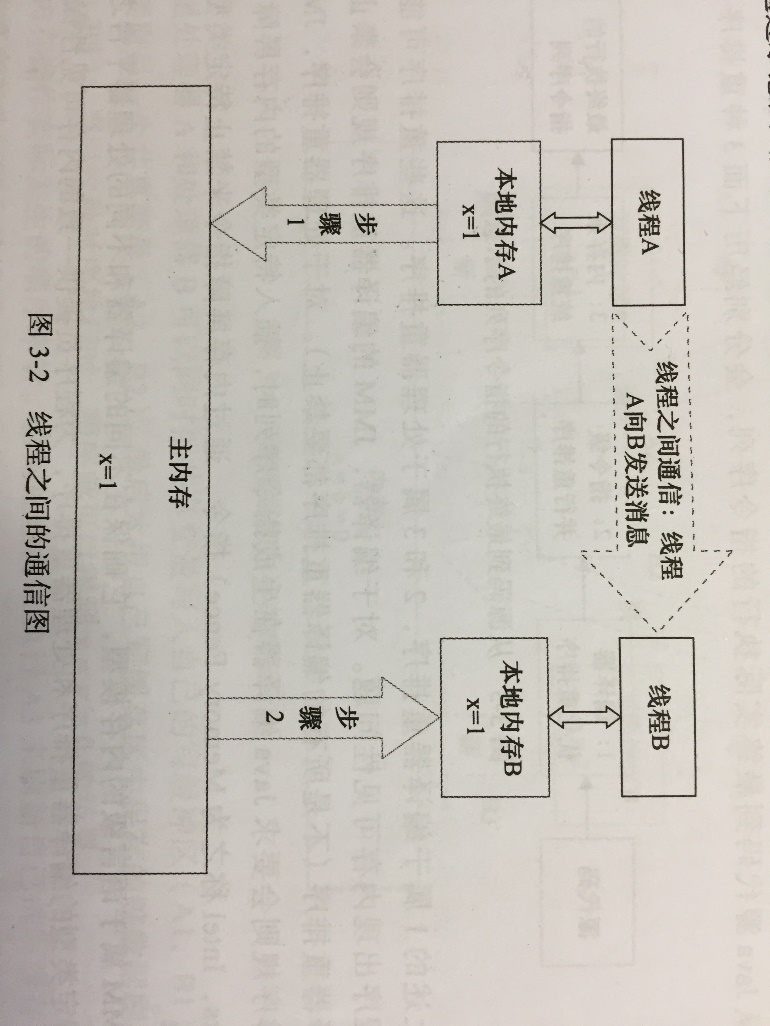
Java内存模型的基础，顺序一致性，同步原语

## 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。

JMM通过控制主内存与每个线程的本地内存之间的交互，来为Java程序提供内存可见性保护。



在执行程序时，为了提高性能，编译器和处理器常常会对指令进行重排序：

* 编译器优化的重排序：编译器在不改变单线程语义的前提下，重排语句的执行顺序
* 指令级并行的重排序：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
* 内存系统的重排序：由于处理器使用了缓存和读/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

JMM的处理器重排序规则会要求Java在编译器生成指令序列时，插入特定的内存屏障指令，来禁止特定类型的处理器重排序，保证一致的内存可以见性。

### 1.1 写缓冲区的优势

写缓冲区可以保证指令流水线持续运行，它可以避免由于处理器停顿下来等待想内存写入数据而产生的延迟。同时通过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。

### 1.2 happens-before

JSR-133内存模型使用happens-before的概念来阐述操作之间的内存可见性。在JMM中，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存在happens-before关系。

happens-before关系如下：

* 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的读
* 传递性：如果A happens-before B，且B happens-before C，那么A happens-before C

## 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。

### 2.1 数据依赖性

如果两个操作访问同一个变量，且其中一个操作为写，此时两个操作之间存在数据依赖。只要重排序两个操作的执行顺序，执行结果就会改变。编译器和处理器在重排序时，会遵守数据依赖性。

这里的依赖性仅针对于单个处理器中执行的指令序列和单个线程中执行的操作。

### 2.2 重排序对多线程的影响

|  |
| --- |
| **示例代码** |
| class ReorderExample**{**  int a **=** 0**;**  boolean flag **=** **false;**    public void writer**()** **{**  a **=** 1**;** // 1  flag **=** **true;** // 2  **}**    public void reader**()** **{**  **if** **(**flag**)** **{** // 3  int i **=** a **\*** a**;** // 4  **}**  **}**  **}** |

## 3. 顺序一致性

Java内存模型规范对数据竞争的定义：在一个线程中写一个变量，在另一个线程中读同一个变量，而且写和读没有通过同步来排序。

如果成行是正确同步的，程序的执行将具有顺序一致性--即程序的执行结果与改程序在顺序一致性内存模型中执行结果相同。

未同步程序在JMM中不但整体的执行顺序是无序的，而且所有线程看到的操作执行顺序也可能不一致。只有当一个线程把本地内存中写过的数据刷新到主内存之后，这个写操作才能对其他线程可见。在这种情况下，该线程和其他线程看到的操作执行顺序不一致。

对于未同步或未正确同步的多线程程序，JMM只提供最小安全性：线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值（0,null, false），不会是无中生有的值。

JMM不保证对64位的long型和double型变量的写操作具有原子性（32位处理器），可以使用volatile来声明，保证其原子性。

## 4. volatile的内存语义

对volatile变量的单个读/写（只是读和写操作，不包含复合操作），可看成是使用同一个锁对这些单个读/写操作做了同步。

volatile变量具有以下特性：

* 可见性：对一个volatile变量的读，总能看到所有线程对这个volatile变量最后的写入
* 原子性：对任意单个volatile变量的读/写具有原子性，但类似于volatile++这种复合操作，不具备原子性

### 4.1 volatile写-读的内存语义

volatile写的内存语义： 当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量的值刷新到住内存中

volatile读的内存语义：当读取一个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。

### 4.2 volatile内存语义的实现

|  |  |  |  |
| --- | --- | --- | --- |
| **是否能重排序** | **第二个操作** | | |
| **第一个操作** | **普通读/写** | **volatile读** | **volatile写** |
| 普通读/写 |  |  | NO |
| volatile读 | NO | NO | NO |
| volatile写 |  | NO | NO |

上表中表示在程序中，当第一个操作为普通变量的读或写时，如果第二个操作是volatile写，则编译器不能重排序这两个操作。

从上表中可以看出

* 当第二个操作是volatile写时，不管第一个操作是什么，都不能重排序，这个规则确保volatile写之前的操作不会被编译器重排序到volatile写之后
* 当第一个操作是volatile读时，不管第二个操作是什么，都不能重排序。这个规则确保volatile读之后的操作不会被编译器重排序到volatile读之前
* 当第一个操作是volatile写，第二个操作是volatile读时，不能重排序

正确使用volatile变量的条件：对变量的写操作不依赖于当前值；该变量没有包含在具有其他变量的不变式中（条件判断等）

编译器不会对volatile读与volatile读后面的任意内存操作重排序；编译器不会对volatile写与volatile写前面的任意内存操作重排序。

## 5. 锁的内存语义

锁是Java并发编程中最重要的同步机制。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。

当线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存中。

当线程获取锁时，JMM会把该线程对应的本地变量内存置为无效。

### 5.1 锁内存语义的实现

ReentrantLock的实现依赖于Java同步框架AbstractQueuedSynchronizer，该框架使用一个整型的volatile变量来维护同步状态。通过CAS获取锁，CAS同时具有volatile读和volatile写的语义：

* 公平锁和非公平锁释放时，最后都要写一个volatile变量state
* 公平锁获取时，首先会去读volatile变量
* 非公平锁获取时，首先会用CAS更新volatile变量，这个操作同时具有volatile读和写的内存语义

### 5.2 concurrent包的实现

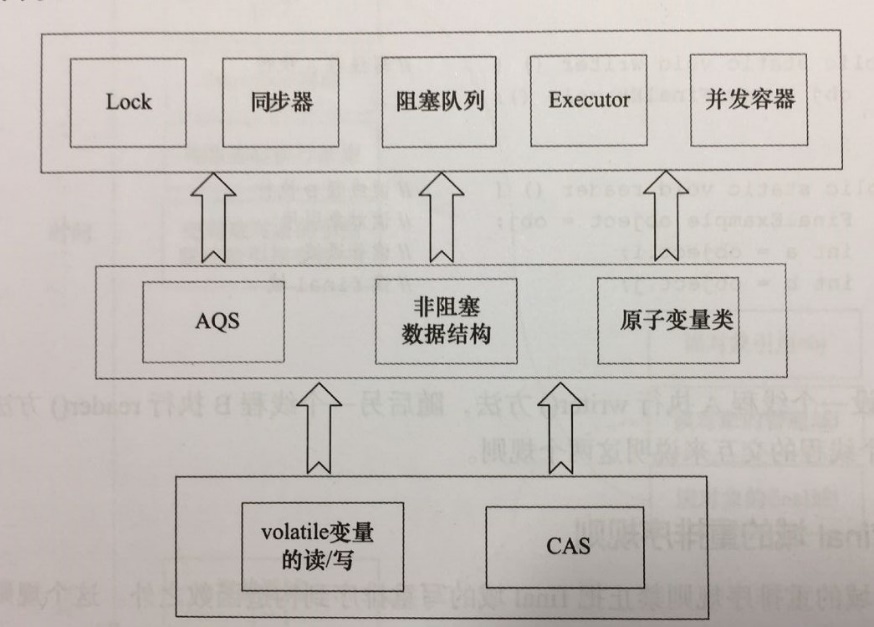
因Java的CAS同时具有volatile读和volatile写的内存语义，因此Java线程之间的通信现在有了以下4种方式：

* A线程写volatile变量，随后B线程读取这个变量
* A线程写volatile变量，随后B线程用CAS更新这个volatile变量
* A线程使用CAS更新一个volatile变量，随后B线程使用CAS更新这个volatile变量
* A线程使用CAS更新一个volatile变量，随后B线程读取这个volatile变量

Java的CAS会使用现代处理器提供的高效机器级别的原子指令，这些原子指令以原子方式对内存执行读-改-写操作。这是concurrent包得以实现的基石：

首先声明共享变量volatile；然后使用CAS的原子条件的更新来实现线程之间的同步；同时配合以volatile的读/写和CAS所具有的volatile读和写的内存语义来实现线程之间的通信。

下图中展示了concurrent包实现的示意图。



## 6. 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. 构造函数内对一个final域的写入，与随后把这个被构造函数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（JMM禁止编译器把final域的写重排序到构造函数之外）。

2. 初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序

|  |
| --- |
| public class FinalExample **{**  int i**;**  final int j**;**  static FinalExampleobj**;**  public FinalExample **()** **{**  i **=** 1**;**  j **=** 2**;**  **}**  public static void writer **()** **{** // 写线程A执行  obj **=** **new** FinalExample**();**  **}**  public static void reader **()** **{**  FinalExample object **=** obj**;** // 读对象引用  int a **=** object**.**i**;** // 读普通域  int b **=** object**.**j**;** // 读final域  **}**  **}** |

写final域的重排序规则可以确保：在对象引用被任意线程可见之前，对象的final域已经被正确的初始化过了，而普通对象不具有这个保障。（线程B看到对象obj时，很可能obj对象还没有构造完成，i的写入操作还没有完成）

读final域的重排序规则可以确保：在读一个对象的final域之前，一定会先读包含这个final域的对象的引用。

对于引用类型，写final域的重排序规则对编译器和处理器增加了如下约束：在构造函数内对一个final引用对象的成员域的写入，与随后在构造函数外把这个被构造函数对象的引用赋值给一个引用变量，这两个操作不能被重排序。

final引用不能从构造函数中“溢出” P60

## 7. happens-before 原则

JMM中定义了如下happens-before规则：

* 程序顺序规则：一个线程中的每个操作，happens-before于该线程的任意后续操作
* 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：对某个volatile变量的写，happens-before于任意后续对这个变量的读。
* 传递性
* start()规则：如果线程A执行操作ThreadB.start()，那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作
* join()规则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()成功返回

## 8.双重检查锁定与延迟初始化

|  |
| --- |
| public class DoubleCheckedLocking **{**  private static Instance instance**;**  public static Instance getInstance **()** **{**  **if** **(**instance **==** **null)** **{** // 第一次检查  synchronized **(**DoubleCheckedLocking**.**class**)** **{** // 加锁  **if** **(**instance **==** **null)** **{** // 第二次检查  instance **=** **new** Instance**();** // 问题根源在此  **}**  **}**  **}**  **return** instance**;**  **}**  **}** |

上面代码的问题根源(instance=new Instance())创建了一个对象，这一行代码可以分解成3行伪代码：

memory = allocate(); // 1：分配对象的内存空间

ctorInstance(memory); // 2: 初始化对象

instance = memory; // 3: 设置instance指向新分配的空间

上面代码中2和3可能会被重排序，线程B可能会访问到一个未初始化的对象。

解决方案：

* 使用volatile来修饰Instance，volatile禁止了2和3的重排序
* 基于类的初始化解决方案：JVM在类的初始化阶段，JVM会获取一个锁，这个所可以同步多个线程对同一个类的初始化

volatile除了可以对静态字段实现延迟实例化外，还可以对实例字段实现延迟初始化，基于类的实例化只能实例化静态字段。字段延迟初始化降低了初始化类或创建实例的开销，但增加了访问被延迟初始化字段的开销

|  |
| --- |
| **基于类初始化的解决方案** |
| public class InstanceFactory **{**  private static class InstanceHolder **{**  public static Instance instance **=** **new** Instance**();**  **}**  public static Instance getInstance**()** **{**  **return** InstanceHolder**.**instance**;**  **}**  **}** |
| **基于volatile的解决方案** |
| public class DoubleCheckedLocking **{**  private static volatile Instance instance**;**  public static Instance getInstance **()** **{**  **if** **(**instance **==** **null)** **{** // 第一次检查  synchronized **(**DoubleCheckedLocking**.**class**)** **{** // 加锁  **if** **(**instance **==** **null)** **{** // 第二次检查  instance **=** **new** Instance**();** // 问题根源在此  **}**  **}**  **}**  **return** instance**;**  **}**  **}** |