Java并发编程

理论基础

并发编程的源头

缓存导致的可见性问题

一个线程对共享变量的修改，另一个线程能立马看到，称为可见性

多核时代，每颗 CPU 都有自己的缓存，这时 CPU 缓存与内存的数据一致性就没那么容易解决了，当多个线程在不同的 CPU 上执行时，这些线程操作的是不同的 CPU 缓存，比如下图中，线程 A 操作的是 CPU-1 上的缓存，而线程 B 操作的是 CPU-2 上的缓存，很明显，这个时候线程 A 对变量 V 的操作对于线程 B 而言就不具备可见性了。

线程切换带来的原子性问题

一个或多个操作在CPU执行的过程中不被中断的特性称为原子性

CPU 能保证的原子操作是 CPU 指令级别的，而不是高级语言的操作符

操作系统做任务切换，可以发生在任何一条 CPU 指令执行完

编译优化带来的有序性

有序性是指程序按代码的先后顺序执行

编译器为了优化性能，有时候会改变程序中语句的先后顺序，例如程序中：“a=6；b=7；”编译器优化后可能变成“b=7；a=6；”，在这个例子中，编译器调整了语句的顺序，但是不影响程序的最终结果。不过有时候编译器及解释器的优化可能导致意想不到的 Bug。

解决问题

Java内存模型

Java内存模型规范了JVM如何按需禁用缓存和编译优化的方法

- Volatile
  - 保证了有序性和可见性
- Synchronized
  - 保证了原子性、有序性、可见性
- Final
  - final修饰符与对象的安全发布
  - final关键字表示已经定义了常量，任意线程都不可以修改， 不可用
- Happens-Before原则
  - 前一个操作对后一个操作是可见的
  - 具体有
    - 程序的顺序性规则：前面的操作Hb与后续的任何操作
    - Volatile变量规则：一个Volatile变量的写操作Hb与后续对着Volatile变量的读操作
    - 传递性：A HB B，且B HB C,那么A HB C
    - 管程中锁的规则：一个锁的解锁Hb与后续这个锁的加锁
    - 线程Start()规则：主线程A启动子线程B，子线程B能够看到主线程A在启动子线程B前的操作
    - 线程Join()规则：主线程A等待子线程B完成（主线程A通过调用子线程B的join方法实现），当子线程B完成后（主线程A中的join方法返回），主线程能够看到子线程的操作。也就是如果在线程 A 中，调用线程 B 的 join() 并成功返回，那么线程 B 中的任意操作 Happens-Before 于该 join() 操作的返回

互斥锁

主要是通过的存屏障(memory barrier)禁止重排序的。即时编译器根据具体的底层体系架构，将这些内存屏障替换成具体的 CPU 指令。对于编译器而言，内存屏障将限制它所能做的重排序优化。而对于处理器而言，内存屏障将会导致缓存的刷新操作。比如，对于volatile，编译器将在volatile字段的读写操作前后各插入一些内存屏障。

- Synchronized
  - synchronized 关键字可以用来修饰方法，也可以用来修饰代码块
    - 当修饰静态方法的时候，锁定的是当前类的 Class 对象
    - 当修饰非静态方法的时候，锁定的是当前实例对象 this
  - 不能用多把锁保护同一个资源，但可以用一把锁保护多个资源
- 死锁
  - 一组相互竞争原的线程因相互等待，导致“永久”阻塞的现象
  - 必要条件
    - 互斥，共享资源X和Y只能被一个线程占用
    - 占用且等待，线程T1已获取共享资源X，在等待共享资源Y的同时不释放共享资源X
    - 不可抢占，其他线程不能强行抢占线程T1的资源
    - 循环等待，线程T1等待线程T2占有的资源，线程T2 等待线程T1占有的资源
  - 预防死锁
    - 对于占有且等待，破坏它，需要一次性申请所有资源
    - 对于不可抢占，占用部分资源的线程进一步请其他资源，如果申请不到就主动释放它占有的资源
    - 循环等待，我们可以按照顺序申请资源来预防

线程间的协作机制，等待——通知机制

- 线程首先获取互斥锁，当线程要求的条件不满足时，释放互斥锁，进入等待状态，当要求的条件满足时，通知等待的线程重新获取互斥锁
- synchronized 配合 wait()、notify()、notifyAll() 这三个方法就能轻松实现。
- notify() 是会随机地通知等待队列中的一个线程，而 notifyAll() 会通知等待队列中的所有线程
- 尽量使用 notifyAll()

管程Monitor：并发编程的万能钥匙

管程，指的是管理共享变量以及对共享变量的操作过程，让他们支持并发

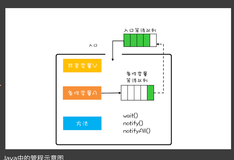
管程和信号量是等价的，所谓等价指的是用管程能够实现信号量，也能用信号量实现管程

在管程模型里，共享变量和对共享变量的操作是被封装起来的，图中最外层的框就代表封装的意思。框的上面只有一个入口，并且在入口旁边还有一个入口等待队列。当多个线程同时试图进入管程内部时，只允许一个线程进入，其他线程则在入口等待队列中等待

管程里还引入了条件变量的概念，而且每个条件变量都对应有一个等待队列

那条件变量和条件变量等待队列的作用是解决线程同步问题

Java 参考了 MESA 模型，语言内置的管程（synchronized）对 MESA 模型进行了精简。MESA 模型中，条件变量可以有多个，Java 语言内置的管程里只有一个条件变量



Java中的管程示意图

并发编程里两大核心问题——互斥和同步，都可以由管程来帮你解决

Java的线程

通用的线程生命周期

五态图

初始状态，指的是线程已经被创建，但是还不允许分配 CPU 执行。这个状态属于编程语言特有的，不过这里所谓的被创建，仅仅是在编程语言层面被创建，而在操作系统层面，真正的线程还没有创建。

可运行状态，指的是线程可以分配 CPU 执行。在这种状态下，真正的操作系统线程已经被成功创建了，所以可以分配 CPU 执行。

当有空闲的 CPU 时，操作系统会将其分配给一个处于可运行状态的线程，被分配到 CPU 的线程的状态就转换成了运行状态。

运行状态的线程如果调用一个阻塞的 API（例如以阻塞方式读文件）或者等待某个事件（例如条件变量），那么线程的状态就会转换到休眠状态，同时释放 CPU 使用权，休眠状态的线程永远没有机会获得 CPU 使用权。当等待的事件出现了，线程就会从休眠状态转换到可运行状态。

线程执行完或者出现异常就会进入终止状态，终止状态的线程不会切换到其他任何状态，进入终止状态也就意味着线程的生命周期结束了。

Java线程的生命周期

状态图

- NEW（初始化状态）
- RUNNABLE（可运行 / 运行状态）
- BLOCKED（阻塞状态）
- WAITING（无时限等待）
- TIMED_WAITING（有时限等待）
- TERMINATED（终止状态）

休眠状态，也就是说只要 Java 线程处于这三种状态之一，那么这个线程就永远没有 CPU 的使用权。

状态转换

- RUNNABLE 与 BLOCKED 的状态转换
  - 线程等待 synchronized 的隐式锁
- RUNNABLE 与 WAITING 的状态转换
  - 获得 synchronized 隐式锁的线程，调用无参数的 Object.wait() 方法
  - 调用无参数的 Thread.join() 方法。其中的 join() 是一种线程同步方法。例如有一个线程对象 thread A，当调用 A.join() 的时候，执行这条语句的线程会等待 thread A 执行完，而等待中的这个线程，其状态会从 RUNNABLE 转换到 WAITING。当线程 thread A 执行完，原来等待它的线程又会从 WAITING 状态转换到 RUNNABLE。
  - 调用 LockSupport.park() 方法。其中的 LockSupport 对象，其实 Java 并发包中的锁，都是基于它实现的。调用 LockSupport.park() 方法，当前线程会阻塞，线程的状态会从 RUNNABLE 转换到 WAITING。调用 LockSupport.unpark(Thread thread) 可唤醒目标线程，目标线程的状态又会从 WAITING 状态转换到 RUNNABLE。
- RUNNABLE 与 TIMED_WAITING 的状态转换
  - 调用带超时参数的 Thread.sleep(long millis) 方法；
  - 获得 synchronized 隐式锁的线程，调用带超时参数的 Object.wait(long timeout) 方法；
  - 调用带超时参数的 Thread.join(long millis) 方法；
  - 调用带超时参数的 LockSupport.parkNanos(Object blocker, long deadline) 方法；
  - 调用带超时参数的 LockSupport.parkUntil(long deadline) 方法。
- 从 NEW 到 RUNNABLE 状态
  - Java 线程要执行，就必须转换到 RUNNABLE 状态。从 NEW 状态转换到 RUNNABLE 状态很简单，只要调用线程对象的 start() 方法就可以了
- 从 RUNNABLE 到 TERMINATED 状态
  - 线程执行完 run() 方法后，会自动转换到 TERMINATED 状态，当然如果执行 run() 方法的时候异常抛出，也会导致线程终止。
  - 强制中断run() 方法的执行
    - stop() 方法
      - stop() 方法会真的杀死线程，不给线程喘息的机会，如果线程持有 ReentrantLock 锁，被 stop() 的线程并不会自动调用 ReentrantLock 的 unlock() 去释放锁，那其他线程就再也没机会获得 ReentrantLock 锁
    - interrupt() 方法
      - interrupt() 方法仅仅是通知线程，线程有机会执行一些后续操作，同时也可以无视这个通知

创建多少线程合适

- CPU密集型
  - CPU核数+1
- I/O密集型
  - CPU核数/(1-阻塞系数)，阻塞系数0.8~0.9
- 最佳线程数 =CPU 核数 * [1+ (I/O 耗时 / CPU 耗时)]

归纳

类比