Java多线程与并发编程

天猫 产品技术部 陌铭

2012-03-29

- 多线程与并发的一些基本知识
- Java并发处理
- Java多线程编程
- 多线程编程应用模式
- 常见多线程资源调度
- 代码学习

多线程与并发的一些基本知识

2012-03-29

关于多线程的几个小问题?

- JVM中的线程与操作系统线程有何关系?
- 线程与CPU核数的关系，JVM支持多少线程?
- 多线程是否一定具备高性能?
- 在我们的应用中有哪些线程?
- 什么是线程的栈？栈有多大？如何修改?
- 线程间是如何通信的?

关于并发的几个小问题?

- 并发中所说的原子性是什么?
- 什么是乐观锁、什么是悲观锁?
- 除了多线程编程，我们还能看到哪些并发场景?
- 并发编程与并行编程的相同和区别?

什么是线程？

- **JVM的线程受制于以下三个条件：**
  - 操作系统支持的线程数
    - 又受限于操作系统为每个线程设置的栈空间
    - Windows（1M），Linux（8M、1024）
  - JVM设置的栈空间大小（-Xss） 默认1M
- **JVM设置的最佳线程数**
  - 和机器有关系
  - 和操作系统有关系
  - 和JVM有关系
  - 经验值：X86服务器每核50-100之间。

一些术语

- **原子性**
  - 不可分割性的操作（CPU指令、操作系统指令、VM指令）
  - CAS （Compare And Set）原语
- **线程**
  - 系统对操作进行调度的最小单元（操作系统、VM）
- **并发**
  - 当多于一个线程（进程）同时争抢同一个资源时
    - 资源包括执行程序，访问IO，访问内存数据块
- **阻塞**
  - 线程发生的被动式等待行为（如等待获取独占性资源、等待反馈等）。

一些术语

- **并发**
  - 当多于一个线程（进程）同时争抢同一个资源时
    - 资源包括执行程序，访问IO，访问内存数据块
- **互斥**
  - 为解决并发问题所采取的线程（进程）间的排他性行为。
  - 有时也将互斥描述为并发冲突
- **锁**
  - 用来实现线程（进程）间互斥的的方法
  - 乐观锁与悲观锁（准确说乐观锁并非是真正的锁）
  - 共享锁与独享锁（共享锁就是读锁，独享锁也就是互斥锁）
- **死锁**
  - 当存在两个或以上拥有互斥锁的线程（进程）同时等待对方释放锁时出现的阻塞问题。

一些术语

- **同步与异步**
  - 同步：串行化处理
  - 异步：并行化处理
- **同步处理**
  - 通过互斥锁机制确保并发线程间的顺序化处理
- **并发冲突**
  - 当出现并发时，资源的使用出现不可判断的不一致情况。
  - 因为不合理互斥导致的线程之间的死锁
- **线程安全**

在Java Concurrency in Practice中是这样定义线程安全的：

当多个线程访问一个类时，如果不用考虑这些线程在运行时环境下的调度和交替运行，并且不需要额外的同步及在调用方代码不必做其他的协调，这个类的行为仍然是正确的，那么这个类就是线程安全的。

显然只有资源竞争时才会导致线程不安全，因此*无状态对象永远是线程安全的*。

原子操作的描述是： 多个线程执行一个操作时，其中*任何一个线程要么完全执行完此操作，要么没有执行此操作的任何步骤*，那么这个操作就是原子的。

多线程与并发

- **多线程应用的好处**
  - 将串行处理变成并行处理，提高资源利用效率
  - 不必阻塞等待，异步处理
- **多线程与并发的关联**
  - 两者之间并无直接关联，只有且仅有在有多个线程访问统一资源时才 有可能出现并发问题。
- **多线程并发带来的影响**
  - 不一致的行为结果
  - 死锁
  - 资源利用率低下

并发与互斥

- **互斥情况下等待带来的死锁问题**
  - 哲学家用餐故事
- **互斥（并发冲突）带来的资源利用效率的问题**
  - 理发师的故事
- **并发使用带来的状态不一致问题**
  - 简单说，并发写带来的信息不一致问题。
  - double check问题

多线程环境下的并发处理

- **分析并发情况下，线程是否安全**
  - 避免出现并发问题
  - 避免无并发问题情况下采取同步互斥
- **合理利用同步互斥，避免出现效率低下的问题**
  - 合理选择同步互斥范围，避免过大范围降低效率
- **分析锁的应用，尤其是互斥锁**
  - 避免出现两个互斥锁之间的等待，从而出现死锁
  - 使用可中断的锁
  - 合理使用共享锁

java多线程编程

2012-03-29

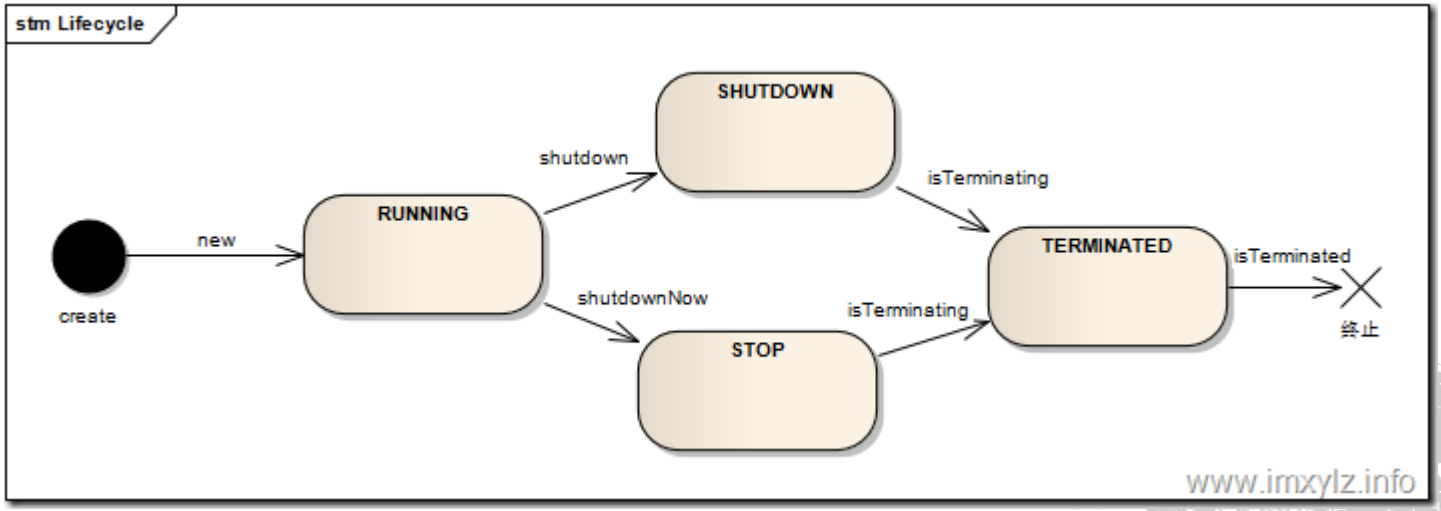
Java 基本线程处理类

- **Java.lang.Thread**
  - Thread.sleep() 与 锁对象的.wait(), lock.lock()
  - 线程优先级：默认等于创建该线程的所在线程
  - 线程状态：
    - NEW, RUNNABLE, WAITTING, TIMED_WAITTING
    - BLOCKEC, TERMINATED
  - join()：等待所在线程结束。
  - Daemon 线程：背景线程，随着JVM主线程结束而结束
  - yield()：暂停一下当前线程，并让JVM重新调度
    - 当前线程并不释放锁也不挂起

Java 基本线程处理类

- **Runnable**
  - 准确说它并不是线程类，而是线程可运行的接口
- **ThreadLocal**
  - 本地线程，也就是JVM保存与线程ID相关的存储结构
  - 实现类似Http Request存储范围
- **ThreadGroup**
  - 将线程分组，用于管理
- **Thread.UncaughtExceptionHandler**
  - 处理线程运行时发生的非捕捉异常
  - 非常有用，避免线程泄漏以及线程统一异常处理

- **Executors**
  - Excutors （Wrapper类）/ ThreadPoolExecutor
    - 固定线程池（单线程）、
    - 调度线程池：指定时间延后执行的线程池。
    - 缓存线程池(newCachedThreadPool)：线程保留时间为60s



Java 异步任务

2012-03-29

Java 并行框架

- **CyclicBarrier**
  - **线程执行barrier的await方法时，计数器累加**
  - **当计数器累加达到阈值，阻塞的线程就会运行**
  - **确保多个线程同时执行，用于多线程并发测试。**
- **CountDownLatch**
  - 和CyclicBarrier类似，**不过是递减为0，并且计数和等待分开**
  - **当计数器为0时，等待的线程取消等待，在应用场景有些类似于join方法**
- **Semaphore**
  - 信号量，用于计数控制
- **Phaser**
  - **Phaser： 结合了上述两者， 还支持运行期修改阈值**

多线程环境下线程有哪些问题

- **线程泄漏**
  - 当线程出现异常后，没有有效捕捉处理，导致线程泄漏，无法被处理。
  - 无法对线程进行控制，比如无法中断线程，当遇到阻塞时便会出现线程泄漏。
- **线程不可中断阻塞（死锁）**
- **线程栈溢出**
  - 线程栈过浅或线程执行方法过深导致SOFE
- **线程分配与资源分配**
  - 线程之间工作分配的效率；线程数量与系统资源的冲突

多线程编程的常见问题

- **独立创建线程并不进行任何管理**
  - 不对线程异常进行任何管理，导致线程泄漏
  - 不对线程的运行进行控制处理，导致无法控制
  - 盲目将线程设置成daemon，导致线程泄漏
- **自我管理线程池**
  - 内部创建和管理线程池，导致资源占用和浪费
  - 不正确的ThreadLocal使用，导致数据污染
- **线程中会执行非中断阻塞方法**
  - 导致线程阻塞，从而泄漏。
- **线程间通信没有合理进行同步控制，导致数据不一致**

如何优雅地进行多线程编程

- **统一的异常处理**
  - 设置线程UncaughtExceptionHandler。
- **对产生的线程分组**
- **使用控制对象来控制线程的运行**
  - 比如通过状态来控制线程的关闭
- **尽量使用可中断的锁，避免线程使用阻塞方法**
- **合理使用线程池**
  - 避免线程过多占用资源
- **避免滥用sleep**
  - 使用线程挂起代替sleep

Java 并发处理

Java的锁机制

- **锁的类型**
  - 互斥锁： synchronized
  - 共享锁：读写锁 （ReentrantLock）
- **锁的范围**
  - 类锁
    - 静态方法前使用Synchronized
  - 对象锁
    - 对象方法前使用synchronized
  - 局部锁（块锁）
    - 在方法内部局部代码块使用特定锁

Java的锁机制

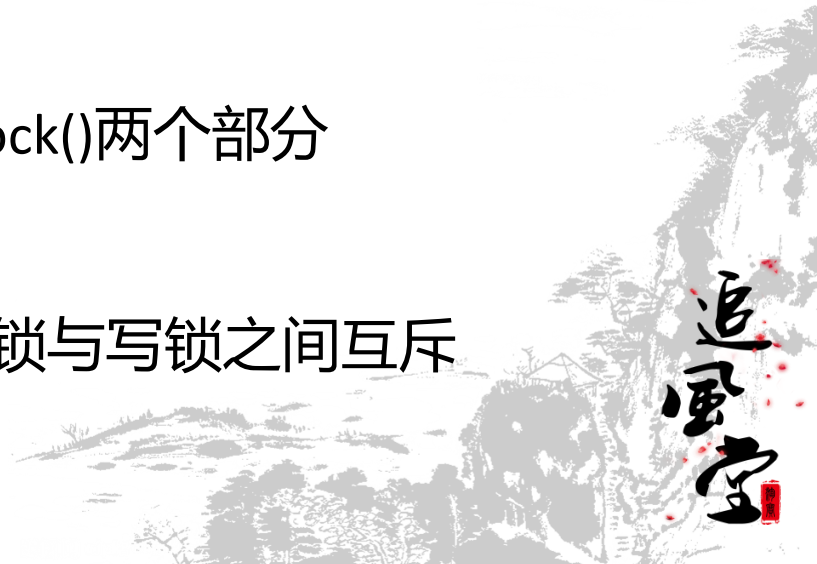
- **锁的行为**
  - 不可中断阻塞
    - lock(), synchronized(lockobject)
  - 可中断阻塞
    - lockInterruptly
  - 可中断性尝试性锁
    - tryLock()

Java 并发相关关键字与方法

- **synchronized**
  - 同步关键字，用于同步程序块（资源）
- **volatile**
  - 用于描述变量同步，避免线程栈中对象值与内存值的不同步问题。
- **Object**
  - 仅有该对象是锁时，才能调用下面方法。
  - wait()：持有改锁的当前线程挂起，释放锁
  - notify()/notifyAll(): 唤醒在该锁挂起的其他线程

Java 并行框架

- **Lock**
  - lock() //不可中断锁，相当于synchronized(xx) 开始
  - lockInterruptly() //可中断锁
  - trylock() //尝试锁
  - unlock() //释放锁，相当于synchronized(xx) 结束
- **ReadWriteLock**
  - 包括读锁readLock()和写锁writeLock()两个部分
  - 读锁是共享锁
  - 写锁是互斥锁，读锁与写锁，写锁与写锁之间互斥



Java 并行框架

- **Fork-Join**
  - 同并行编程的分治与归并算法原理类似
  - 主要步骤
    - 在RecursiveAction/RecursiveTask中实现最小计算逻辑
    - 实现类中实现原子任务的分解与归并
- **AtomicXXX**
  - 基于CAS的基本数据类型原子性操作。
  - 比如AtomicInteger就可以用于计数控制，
  - 比如AtomicReference常用于初始化控制
  - 比如AtomicBoolean用于状态控制
- **AtomicMarkableReference/AtomicStampedReference**
  - 用于基于Object的状态值（Boolean, Integer）进行更新
  - 可以用于缓存的状态实现，以及内存数据的版本戳、乐观锁

并发编程下的常见问题

- **基本不做任何同步控制**
- **滥用synchronized**
  - 盲目放大同步控制范围
  - 对线程安全的部分依旧使用同步控制
- **不清晰甚至错误的锁机制**
  - 导致出现死锁情况
- **滥用同步集合类**
  - 以为使用了同步集合类比如ConcurrentHashMap就没有同步问题

如何进行优雅的并发编程

- **尽量减少对全局变量的改变**
  - 保证方法本身就具备线程安全
- **尽量不暴露内部状态，减少同步问题发生可能性**
  - 考虑Immutable应用（比如CopyOnWrite模式）
- **使用同步块而不是同步方法**
  - 减少同步范围，提高效率
- **合理使用同步**
  - 识别是否具有线程冲突，避免不合理同步控制
- **使用原子操作（比如CAS）代替传统变量**
- **使用信号量来控制线程挂起和唤醒而不是固定的sleep**

多线程编程应用模式

2012-03-29

Immutable

- **不改变内部状态，确保对象本身没有线程安全问题**
  - 特殊形式：没有全局属性
  - 全局属性不能被外部改变
    - 初始化就固定
    - 只暴露属性读的方法
  - 采用保护的方式避免外部修改问题
    - 比如CopyOnWrite方式
      - 将原有值复制一份，在复制后的数据基础上进行修改
      - 将原指向原有值的引用指向复制的值。
      - 确保上述操作写锁同步
    - 对属性的修改进行同步控制

读写锁应用

- **利用共享锁，最大化并发效率**
  - 简单说就是允许并发读、单线程写
    - 读锁是可重入，也就是允许并发访问
    - 写锁是互斥，包括写锁之间以及写锁和读锁之间
  - 读写锁在Java的标准库中已经实现

生产者-消费者

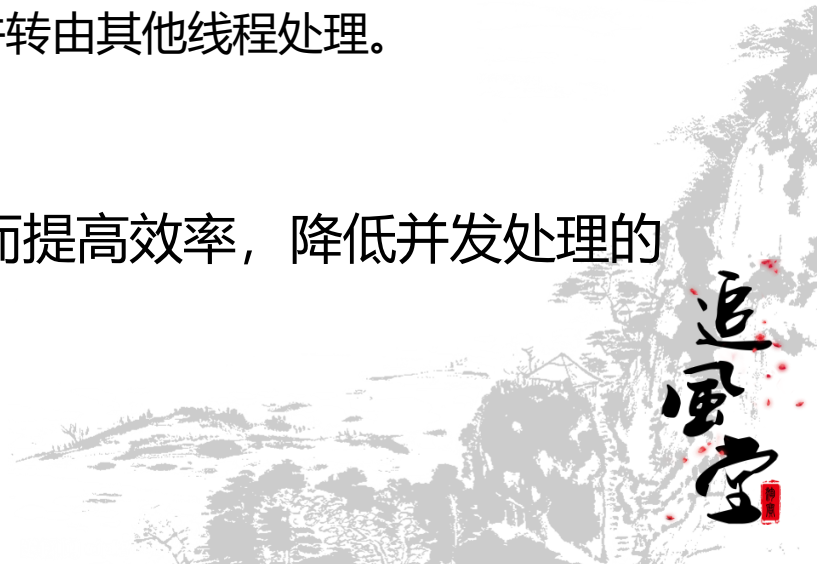
- **将逻辑（任务）的产生与处理分离**
  - 任务生产由生产线程完成，任务处理由消费线程完成
    - 在实际实现中，生产线程往往是透明的。
  - 生产消费的核心逻辑其实就是将处理异步化
    - 异步并非不是实时，很多所谓的实时处理其实都是异步
    - 避免任务处理成为瓶颈进而影响任务生产
  - 使用队列(FIFO)来作为生产者、消费者间的数据载体
    - 作为生产线程和消费线程通信的核心，同步控制很重要
  - 合理地适配生产-消费之间的速度
    - 根据复杂度、任务数量调整生产者-消费者之间的比重
    - 合理设置通信间数据载体的大小，也就是buffer。
  - 应用场景：其实MQ在服务端就是应用生产者-消费者
- **异步化，更高的线程并发健壮性**

工作线程模式（Worker Thread）

- **独立的线程处理专门的逻辑**
  - 线程可以重复利用
    - 结合线程池使用，线程可被管理控制
      - 线程池必须是可控大小的线程池
  - 线程的职责和状态简单
    - 状态可以简述为启动->等待<->处理任务、结束
    - 外部应用传递任务，通过信号量将等待的工作线程激活，线程进行任务处理，处理完成继续等待。
  - 应用场景：工作线程池
    - Web容器中的Http线程池，MQ服务端中的Queue
- **专属资源、高效率**

职介者

- **利用一个线程担任职介，进行数据的分发**
  - 统一管理其他线程，这样线程本身就不存在同步问题
  - 职介者不处理任何逻辑，仅仅用于传递数据
- **职介者模式最典型的应用就是多路复用**
  - 多路复用原是指重复利用一个通道传输不同的数据
    - 比如Socket，IO等
    - 利用一个职介线程收取socket数据，并转由其他线程处理。
- **职介者模式的优点**
  - 将多线程并发变成多线程有序，从而提高效率，降低并发处理的难度和并发带来的效率问题



多线程资源调度

2012-03-29

流水线模式

- **采取FIFO方式（顺序算法）**
- **有专门的分配器将生产内容分配到各个消费线程上。**
  - 最为常见，比如MQ中的消息处理
- **在这基础上会产生以下几种改变**
  - 基于优先级的分配
    - 将高优先级的内容（任务）先进行分配
  - 基于可用性的分配
    - 只将新任务分配那些空闲状态的线程
- **优缺点：管理成本略高，原则上无并发问题**

竞争模式

- **属于并行算法的一种**
- **没有分配器，线程主动竞争获取资源**
- **优缺点：**
  - 不需要对任务的分配过程进行管理
  - 会产生并发问题
  - 优异性能与差性能的机率并存。

工作窃取模式

- **相当于流水线模式与竞争模式的结合**
- **空闲的工作线程可以主动领取更多任务甚至窃取其他线程待办的任务。**
- **分配器的角色变为对线程的管理而非任务的分配**
- **优缺点：**
  - 性能不受限于线程任务分配，具有更高效率
  - 线程间原则上不存在并发冲突
  - 较为复杂的线程处理算法，实现较难（JDK7中支持fork/join）。

代码学习

2012-03-29

常见代码问题分析

```
public Object getCatLock(Long catId) {
    synchronized (categoryFlag) {
        Object catLock = categoryLock.get(catId);
```

```
private MicCategoryDTO getCategoryByIdProxy(Long catId, MallForestDO mallForestDO) {
    MicCategoryDTO categoryDTO = null;
    if (mallForestDO == null) {
        mallForestDO = mallForest;
    }
    if (mallForestDO != null) {
        categoryDTO = mallForestDO.getCategoryById(catId);
    }
    if (categoryDTO == null) {
        synchronized (mallForestDO.getMallForestCache().getMallForestLock().getCatLock(catId)) {
            DefaultStdCategoryDO categoryDO = (DefaultStdCategoryDO) defaultReadService.getStdCategory(catId);
            if (categoryDO != null) {
                DefaultStdCategoryDO parentCat = categoryDO.getParent();
                if (parentCat != null) {
                    this.getCategoryByIdProxy((long) parentCat.getParentId(), mallForestDO);
                }
                categoryDTO = new MicCategoryDTO();
                this.parsePureCategory(categoryDTO, categoryDO);
                this.getAllCatProperties(catId, categoryDTO, mallForestDO);
                mallForestDO.setCategory(catId, categoryDTO);
            }
        }
    }
    return mallForestDO.getCategoryById(catId);
}
```

常见代码问题分析

```
    private Lock lock = new ReentrantLock();
    private Condition notEmpty = lock.newCondition();
    private Condition notFull = lock.newCondition();
public class Sample {
    private final ConcurrentHashMap<String, Object> catLocks =
            new ConcurrentHashMap<String, Object>() ;

    public Object getCateLockbyId(String catId) {
        Object lock = catLocks.get(catId);
        if (lock == null) {
            catLocks.putIfAbsent(catId, new Object());
            lock = catLocks.get(catId);
        }
        return lock ;
    }
            } finally {
                lock.unlock();
            }
        }
                }
            }
        }
    }
```

代码学习之三

- 模拟多线程并发的几种方式
- 确保返回处理的异步执行
- 限制方法调用的并发控制
- 可进行超时控制的同步调用
- 基于任务分解的并行计算
- 优雅、高效且安全的延迟装载

有用推荐

书籍与资料

- Java并发编程实战
- Java并发编程-设计原则与模式
- Java多线程设计模式
- B2B 温绍锦 的 Java并发程序设计
- 深入浅出Java Concurrency
- IBM DW: JDK5.0中的并发
- IBM DW: Java多线程与并发编程专题
- IBM DW：Java中的fork-join模式

Q & A

THANKS

-THE END-