多线程&并发

线程

线程状态

- Ready
  - 调用Thread.start()
- Running
  - 获得CPU时间片
- Timed-Waiting
  - Thread.sleep(long)
  - Object.wait(long)
  - Thread.join(long)
  - Locksupport.parkNanos(long)
  - Locksupport.parkUntil(long)
- Waiting
  - Object.wait()
  - Thread.join()
  - LockSupport.park()
- Blocked
  - 获取不到锁
- Terminal
  - 运行结束

创建方法

- 创建Thread实例
- 实现Runnable
- 实现Callable

线程安全

可见性

- happen-before原则
  - 程序次序规则：一个线程内，按照代码先后顺序
  - 锁定规则：一个 unLock 操作先行发生于后面对同一个锁的 lock 操作
  - Volatile 变量规则：对一个变量的写操作先行发生于后面对这个变量的读
  - 传递规则：如果操作 A 先行发生于操作 B，而操作 B 又先行发生于操作 C，则可以得出 A 先于 C
  - 线程启动规则：Thread 对象的 start() 方法先行发生于此线程的每个一个动作
  - 线程中断规则：对线程 interrupt() 方法的调用先行发生于被中断线程的代码检测到中断事件的发生
  - 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过 Thread.join() 方法结束、Thread.isAlive() 的返回值手段检测到线程已经终止执行
  - 对象终结规则：一个对象的初始化完成先行发生于他的 finalize() 方法的开始
- volatile
  - 使用对象头标记字(Object monitor)
  - 适用场景： 单个线程写；多个线程读
  - 原则： 能不用就不用，不确定的时候也不用
  - 替代方案： Atomic 原子操作类

原子性

- 锁
  - synchronized
    - 优点
      - 简单
    - 缺点
      - 灵活性不够，不能够精细控制
      - 获取不到锁只能干等待，容易造成死锁
    - 原理
      - 使用对象头标记字(Object monitor)
  - Lock+Condition
    - ReentrantLock
    - ReentrantReadWriteLock
      - 允许多个线程同时读共享变量
      - 只允许一个线程写共享变量
      - 如果一个写线程正在执行写操作，此时禁止读线程读共享变量
    - StampedLock
      - StampedLock 的性能之所以比 ReadWriteLock 还要好，其关键是 StampedLock 支持乐观读的方式。ReadWriteLock 支持多个线程同时读，但是当多个线程同时读的时候，所有的写操作会被阻塞；而 StampedLock 提供的乐观读，是允许一个线程获取写锁的，也就是说不是所有的写操作都被阻塞。注意这里，我们用的是“乐观读”这个词，而不是“乐观读锁”，是要提醒你，乐观读这个操作是无锁的，所以相比较 ReadWriteLock 的读锁，乐观读的性能更好一些
    - 优点
      - 能够响应中断
      - 支持超时
      - 非阻塞地获取锁
  - 工具类
    - 流控
      - Semaphore
    - 实现线程同步
      - CountdownLatch
      - CyclicBarrer
      - LockSupport
  - 最佳实践
    - 永远只在更新对象的成员变量时加锁
    - 永远只在访问可变的成员变量时加锁
    - 永远不在调用其他对象的方法时加锁

解决办法

- 原子类
  - AtomicInteger
  - AtomicIntegerArray
  - AtomicLong
  - AtomicLongArray
  - AtomicLongFieldUpdater
  - LongAdder
  - LongAccumulator
  - AtomicBoolean
  - AtomicMarkableReference
  - AtomicReference
  - AtomicReferenceArray
  - AtomicReferenceFieldUpdater
  - AtomicStampedReference
  - DoubleAccumulator
  - DoubleAdder
  - Striped64
- 并发容器
  - ConcurrentHashMap
  - CopyOnWriteArrayList
  - Vector
  - Collections.Synchronized.....
- Immutability（不变性模式）
- 线程本地存储模式：ThreaLocal
- Copy-on-Write模式
- Guarded Suspension模式
- Balking 模式
- Thread-Per-Message
- Worker Thread 模式

线程池

- 核心参数
  - 核心线程数
    - 表示线程池保有的最小线程数
  - 最大线程数
    - 表示线程池创建的最大线程数
  - 空闲时间
    - 如果一个线程空闲了keepAliveTime & unit这么久，而且线程池的线程数大于 corePoolSize ，这个空闲的线程就要被回收
  - 工作队列
    - 工作队列，存放等待的任务
  - 线程工厂
    - 通过这个参数你可以自定义如何创建线程，例如你可以给线程指定一个有意义的名字
  - 拒绝策略
    - AbortPolicy：拒绝接收任务并抛出RejectedExecutionException异常（默认策略）
    - DiscardPolicy：直接丢弃新任务
    - DiscardOldestPolicy：丢弃等待队列头部任务接收新任务
    - CallerRunsPolicy：提交任务的线程自己去执行该任务
- 运行过程
  - 1、任务提交进来，如果当前线程数还没达到核心线程数，则创建新线程执行任务
  - 2、线程数已经到达核心线程数，则把任务放入等待队列
  - 3、等待队列已满且当前线程数还未到达最大线程数，则创建新线程执行任务
  - 4、当线程数已到达最大线程数，则执行拒绝策略
  - 5、线程空闲时间已超过指定空闲时间后，销毁线程只保留核心线程数
- 便捷创建线程池方法（不推荐使用）
  - newFixedThreadPool
  - newSingleThreadExecutor
  - newCachedThreadPool
  - newScheduledThreadPool
  - Executors 提供的很多方法默认使用的都是无界的 LinkedBlockingQueue，高负载情境下，无界队列很容易导致 OOM，而 OOM 会导致所有请求都无法处理，这是致命问题。所以强烈建议使用有界队列
- Future
  - cancel()：取消任务的方法
  - isCancelled()：判断任务是否已取消的方法
  - isDone()：判断任务是否已结束的方法
  - get()
  - get(timeout, unit)
  - 获得任务执行结果
- CompletableFuture
  - 实现了CompletionStage接口
    - 串行关系
      - thenApply
      - thenAccept
      - thenRun
      - thenCompose
    - 汇聚关系
      - AND 汇聚关系
        - thenCombine
        - thenAcceptBoth
        - runAfterBoth
      - OR 汇聚关系
        - applyToEither
        - acceptEither
        - runAfterEither
- CompletionService
  - 当需要批量提交异步任务的时候建议你使用 CompletionService。CompletionService 将线程池 Executor 和阻塞队列 BlockingQueue 的功能融合在了一起，能够让批量异步任务的管理更简单。除此之外，CompletionService 能够让异步任务的执行结果有序化，先执行完的先进入阻塞队列，利用这个特性，你可以轻松实现后续处理的有序性，避免无谓的等待，同时还可以快速实现诸如 Forking Cluster 这样的需求
- Fork/Join
  - Fork/Join 并行计算框架主要解决的是分治任务。分治的核心思想是“分而治之”：将一个大的任务拆分成小的子任务去解决，然后再把子任务的结果聚合起来从而得到最终结果。这个过程非常类似于大数据处理中的 MapReduce，所以你可以把 Fork/Join 看作单机版的 MapReduce。Fork/Join 并行计算框架的核心组件是 ForkJoinPool。ForkJoinPool 支持任务窃取机制，能够让所有线程的工作量基本均衡，不会出现有的线程很忙，而有的线程很闲的状况，所以性能很好。Java 1.8 提供的 Stream API 里面并行流也是以 ForkJoinPool 为基础的。不过需要你注意的是，默认情况下所有的并行流计算都共享一个 ForkJoinPool，这个共享的 ForkJoinPool 默认的线程数是 CPU 的核数；如果所有的并行流计算都是 CPU 密集型计算的话，完全没有问题，但是如果存在 I/O 密集型的并行流计算，那么很可能会因为一个很慢的 I/O 计算而拖慢整个系统的性能。所以建议用不同的 ForkJoinPool 执行不同类型的计算任务。