Java并发编程

- Java内存模型（JMM）
  - 线程通信机制
    - 内存共享
      - Java采用
    - 消息传递
  - 内存模型
    - 重排序
      - 为了程序的性能，处理器、编译器都会对程序进行重排序处理
      - 条件
        - 在单线程环境下不能改变程序运行的结果
        - 存在数据依赖关系的不允许重排序
      - 问题
        - 重排序在多线程环境下可能会导致数据不安全
    - 顺序一致性
      - 多线程环境下的理论参考模型
      - 为程序提供了极强的内存可见性保证
      - 特性
        - 一个线程中的所有操作必须按照程序的顺序来执行
        - 所有线程都只能看到一个单一的操作执行顺序，不管程序是否同步
        - 每个操作都必须原子执行且立刻对所有线程可见
    - happens-before
      - JMM中最核心的理论，保证内存可见性
      - 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。
      - 理论
        - 如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前
        - 两个操作之间存在happens-before关系，并不意味着一定要按照happens-before原则制定的顺序来执行。如果重排序之后的执行结果与按照happens-before关系来执行的结果一致，那么这种重排序并不非法
    - as-if-serial
      - 所有的操作均可以为了优化而被重排序，但是你必须要保证重排序后执行的结果不能被改变
  - synchronized
    - 同步、重量级锁
    - 原理

- synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性
- 基于AQS实现，共享锁

- 锁对象
  - 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对象
  - 同步方法块，锁是括号里面的对象
- 实现机制
  - Java对象头
    - synchronized的锁就是保存在Java对象头中的
    - 包括两部分数据
      - Mark Word（标记字段）
        - Mark Word被设计成一个非固定的数据结构以便在极小的空间内存存储尽量多的数据，它会根据对象的状态复用自己的存储空间
        - 包括
          - 哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳
      - Klass Pointer（类型指针）
  - monitor
    - Owner
      - 初始时为NULL表示当前没有任何线程拥有该monitor record，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时又设置为NULL
- 锁优化
  - 自旋锁
    - 该线程等待一段时间，不会被立即挂起，看持有锁的线程是否会很快释放锁（循环方式）
    - 自旋字数较难控制（-XX:preBlockSpin）
    - 存在理论：线程的频繁挂起、唤醒负担较重，可以认为每个线程占有锁的时间很短，线程挂起再唤醒得不偿失
    - 缺点
      - 自旋次数无法确定
  - 适应性自旋锁
    - 自旋的次数不再是固定的，它是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定
    - 自旋成功，则可以增加自旋次数，如果获取锁经常失败，那么自旋次数会减少
  - 锁消除
    - 若不存在数据竞争的情况，JVM会消除锁机制
    - 判断依据
      - 变量逃逸
  - 锁粗化

- 将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁。例如for循环内部获取锁
- 轻量级锁
  - 在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗
  - 通过CAS来获取锁和释放锁
  - 性能依据
    - 对于绝大部分的锁，在整个生命周期内都是不会存在竞争的
  - 缺点
    - 在多线程环境下，其运行效率比重量级锁还会慢
- 偏向锁
  - 为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径
  - 主要尽可能避免不必须要的CAS操作，如果竞争锁失败，则升级为轻量级锁

- volatile
  - 特性
    - volatile可见性：对一个volatile的读，总可以看到对这个变量最终的写
    - volatile原子性：volatile对单个读/写具有原子性（32位Long、Double），但是复合操作除外，例如i++;
  - 实现机制
    - 内存屏障
  - 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值立即刷新到主内存中
    - 当读一个volatile变量时，JMM会把该线程对应的本地内存设置为无效，直接从主内存中读取共享变量
  - 操作系统语义
  - 内存模型
    - 重排序
    - happens-before
- DCL
  - 单例模式
  - DCL
    - 重排序
    - happens-beofre
  - 解决方案
    - volatile方案
      - 禁止重排序
    - 基于类初始化的解决方案
      - 利用classloder的机制来保证初始化instance时只有一个线程。JVM在类初始化阶段会获取一个锁，这个锁可以同步多个线程对同一个类的初始化

- 并发基础

- AQS
  - AbstractQueuedSynchronizer，同步器，实现JUC核心基础组件
  - 解决了子类实现同步器时涉及的大量细节问题，例如获取同步状态、FIFO同步队列
  - 采用模板方法模式，AQS实现大量通用方法，子类通过继承方式实现其抽象方法来管理同步状态
  - CLH同步队列
    - FIFO双向队列，AQS依赖它来解决同步状态的管理问题
    - 首节点唤醒，等待队列加入到CLH同步队列的尾部
  - 同步状态获取与释放
    - 独占式
      - 获取锁
        - 获取同步状态：acquire
        - 响应中断：acquireInterruptibly
        - 超时获取:tryAcquireNanos
      - 释放锁
        - release
    - 共享式
      - 获取锁
        - acquireShared
      - 释放锁
        - releaseShared
  - 线程阻塞和唤醒
    - 当有线程获取锁了，其他再次获取时需要阻塞，当线程释放锁后，AQS负责唤醒线程
    - LockSupport
      - 是用来创建锁和其他同步类的基本线程阻塞原语
      - 每个使用LockSupport的线程都会与一个许可关联，如果该许可可用，并且可在进程中使用，则调用park()将会立即返回，否则可能阻塞。如果许可尚不可用，则可以调用 unpark 使其可用
      - park()、unpark()
- CAS
  - Compare And Swap，整个JUC体系最核心、最基础理论
  - 内存值V、旧的预期值A、要更新的值B，当且仅当内存值V的值等于旧的预期值A时才会将内存值V的值修改为B，否则什么都不干
  - native中存在四个参数
  - 缺陷
    - 循环时间太长
    - 只能保证一个共享变量原子操作
    - ABA问题
      - 解决方案
        - 版本号、标记位
          - AtomicMarkableReference

- 基于时间戳
  - AtomicStampedReference

- 锁
  - ReentrantLock
    - 可重入锁，是一种递归无阻塞的同步机制
    - 比synchronized更强大、灵活的锁机制，可以减少死锁发生的概率
    - 分为公平锁、非公平锁
    - 底层采用AQS实现，通过内部Sync继承AQS
  - ReentrantReadWriteLock
    - 【两把锁】共享锁：读锁、排他锁：写锁
    - 支持公平性、非公平性，可重入和锁降级
    - 锁降级：遵循获取写锁、获取读锁在释放写锁的次序，写锁能够降级成为读锁
  - Condition
    - Lock 提供条件Condition，对线程的等待、唤醒操作更加详细和灵活
    - 内部维护一个Condition队列。当前线程调用await()方法，将会以当前线程构造成一个节点（Node），并将节点加入到该队列的尾部
- 并发工具类
  - CyclicBarrier
    - 它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)
    - 通俗讲：让一组线程到达一个屏障时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活
    - 底层采用ReentrantLock + Condition实现
    - 应用场景
      - 多线程结果合并的操作，用于多线程计算数据，最后合并计算结果的应用场景
  - CountDownLatch
    - 在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待
    - 用给定的计数 初始化 CountDownLatch。由于调用了 countDown() 方法，所以在当前计数到达零之前，await 方法会一直受阻塞。之后，会释放所有等待的线程，await 的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。如果需要重置计数，请考虑使用 CyclicBarrier。
    - 与CyclicBarrier区别
      - CountDownLatch的作用是允许1或N个线程等待其他线程完成执行；而CyclicBarrier则是允许N个线程相互等待
      - CountDownLatch的计数器无法被重置；CyclicBarrier的计数器可以被重置后使用，因此它被称为是循环的barrier
    - 内部采用共享锁来实现
  - Semaphore
    - 信号量
      - 一个控制访问多个共享资源的计数器
    - 从概念上讲，信号量维护了一个许可集。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release() 添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使

用实际的许可对象，Semaphore 只对可用许可的号码进行计数，并采取相应的行动
- 信号量Semaphore是一个非负整数（>=1）。当一个线程想要访问某个共享资源时，它必须要先获取Semaphore，当Semaphore >0时，获取该资源并使Semaphore – 1。如果Semaphore值 = 0，则表示全部的共享资源已经被其他线程全部占用，线程必须要等待其他线程释放资源。当线程释放资源时，Semaphore则+1
- 应用场景
  - 通常用于限制可以访问某些资源（物理或逻辑的）的线程数目
- 内部采用共享锁实现

- Exchanger
  - 可以在对中对元素进行配对和交换的线程的同步点
  - 允许在并发任务之间交换数据。具体来说，Exchanger类允许在两个线程之间定义同步点。当两个线程都到达同步点时，他们交换数据结构，因此第一个线程的数据结构进入到第二个线程中，第二个线程的数据结构进入到第一个线程中

- 其他
  - ThreadLocal
    - 一种解决多线程环境下成员变量的问题的方案，但是与线程同步无关。其思路是为每一个线程创建一个单独的变量副本，从而每个线程都可以独立地改变自己所拥有的变量副本，而不会影响其他线程所对应的副本
    - ThreadLocal 不是用于解决共享变量的问题的，也不是为了协调线程同步而存在，而是为了方便每个线程处理自己的状态而引入的一个机制
    - 四个方法
      - get()：返回此线程局部变量的当前线程副本中的值
      - initialValue()：返回此线程局部变量的当前线程的“初始值”
      - remove()：移除此线程局部变量当前线程的值
      - set(T value)：将此线程局部变量的当前线程副本中的值设置为指定值
    - ThreadLocalMap
      - 实现线程隔离机制的关键
      - 每个Thread内部都有一个ThreadLocal.ThreadLocalMap类型的成员变量，该成员变量用来存储实际的ThreadLocal变量副本。
      - 提供了一种用键值对方式存储每一个线程的变量副本的方法，key为当前ThreadLocal对象，value则是对应线程的变量副本
    - 注意点
      - ThreadLocal实例本身是不存储值，它只是提供了一个在当前线程中找到副本值得key
      - 是ThreadLocal包含在Thread中，而不是Thread包含在ThreadLocal中
      - 内存泄漏问题
        - ThreadLocalMap
          - key 弱引用 value 强引用，无法回收
        - 显示调用remove()
  - Fork/Join

- 一个用于并行执行任务的框架， 是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架
- 核心思想
  - “分治”
  - fork分解任务，join收集数据
- 工作窃取
  - 某个线程从其他队列里窃取任务来执行
  - 执行块的线程帮助执行慢的线程执行任务，提升整个任务效率
  - 队列要采用双向队列
- 核心类
  - ForkJoinPool
    - 执行任务的线程池
  - ForkJoinTask
    - 表示任务，用于ForkJoinPool的任务抽象
  - ForkJoinWorkerThread
    - 执行任务的工作线程

- Java并发集合
  - ConcurrentHashMap
    - CAS + Synchronized 来保证并发更新的安全，底层采用数组+链表/红黑树的存储结构
    - 重要内部类
      - Node
        - key-value键值对
      - TreeNode
        - 红黑树节点
      - TreeBin
        - 就相当于一颗红黑树，其构造方法其实就是构造红黑树的过程
      - ForwardingNode
    - 重要操作
    - 1.8 与 1.7的区别
      - 1.7使用锁分段
      - 1.8使用CAS
  - ConcurrentLinkedQueue
    - 基于链接节点的无边界的线程安全队列，采用FIFO原则对元素进行排序，内部采用CAS算法实现
    - 不变性
      - 在入队的最后一个元素的next为null
      - 队列中所有未删除的节点的item都不能为null且都能从head节点遍历到
      - 对于要删除的节点，不是直接将其设置为null，而是先将其item域设置为null（迭代器会跳过item为null的节点）
      - 允许head和tail更新滞后。这是什么意思呢？意思就说是head、tail不总是指向第一个元素和最后一个元素（后面阐述）

- head的不变性和可变性
- tail的不变性和可变性
- 精妙之处：利用CAS来完成数据操作，同时允许队列的不一致性，弱一致性表现淋漓尽致

- ConcurrentSkipListMap
  - 第三种key-value数据结构：SkipList（跳表）
  - SkipList
    - 平衡二叉树结构
    - Skip list让已排序的数据分布在多层链表中，以0-1随机数决定一个数据的向上攀升与否，通过"空间来换取时间"的一个算法， 在每个节点中增加了向前的指针，在插入、删除、查找时可以忽略一些不可能涉及到的结点，从而提高了效率
    - 特性
      - 由很多层结构组成，level是通过一定的概率随机产生的
      - 每一层都是一个有序的链表，默认是升序，也可以根据创建映射时所提供的Comparator进行排序，具体取决于使用的构造方法
      - 最底层(Level 1)的链表包含所有元素
      - 如果一个元素出现在Level i 的链表中，则它在Level i 之下的链表也都会出
      - 每个节点包含两个指针，一个指向同一链表中的下一个元素，一个指向下面一层的元素
    - 查找、删除、添加
- ConcurrentSkipListSet
  - 内部采用ConcurrentSkipListMap实现
- COW
  - CopyOnWriteList

- atomic
  - 基本类型类
    - 用于通过原子的方式更新基本类型
    - AtomicBoolean
      - 原子更新布尔类型
    - AtomicInteger
      - 原子更新整型
    - AtomicLong
      - 原子更新长整型
  - 数组
    - 通过原子的方式更新数组里的某个元素
    - AtomicIntegerArray
      - 原子更新整型数组里的元素
    - AtomicLongArray
      - 原子更新长整型数组里的元素
    - AtomicReferenceArray
      - 原子更新引用类型数组里的元素
  - 引用类型

- 如果要原子的更新多个变量，就需要使用这个原子更新引用类型提供的类
- AtomicReference
  - 原子更新引用类型
- AtomicReferenceFieldUpdater
  - 原子更新引用类型里的字段
- AtomicMarkableReference
  - 原子更新带有标记位的引用类型

- 字段类
  - 如果我们只需要某个类里的某个字段，那么就需要使用原子更新字段类
  - AtomicIntegerFieldUpdater
    - 原子更新整型的字段的更新器
  - AtomicLongFieldUpdater
    - 原子更新长整型字段的更新器
  - AtomicStampedReference
    - 原子更新带有版本号的引用类型
- JDK1.8新增XXXAdder
  -
  -

- 阻塞队列
  - ArrayBlockingQueue
    - 一个由数组实现的FIFO有界阻塞队列
    - ArrayBlockingQueue有界且固定，在构造函数时确认大小，确认后不支持改变
    - 在多线程环境下不保证“公平性”
    - 实现
      - ReentrantLock
      - Condition
  - LinkedBlockingQueue
    - 基于链接，无界的FIFO阻塞队列
  - PriorityBlockingQueue
    - 支持优先级的无界阻塞队列
    - 默认情况下元素采用自然顺序升序排序，可以通过指定Comparator来对元素进行排序
    - 二叉堆
      - 分类
        - 最大堆
          - 父节点的键值总是大于或等于任何一个子节点的键值
        - 最小堆
          - 父节点的键值总是小于或等于任何一个子节点的键值
      - 添加操作则是不断“上冒”，而删除操作则是不断“下掉”
    - 实现
      - ReentrantLock + Condition

- 二叉堆
- DelayQueue
  - 支持延时获取元素的无界阻塞队列
  - 应用
    - 缓存：清掉缓存中超时的缓存数据
    - 任务超时处理
  - 实现
    - ReentrantLock + Condition
    - 根据Delay时间排序的优先级队列：PriorityQueue
  - Delayed接口
    - 用来标记那些应该在给定延迟时间之后执行的对象
    - 该接口要求实现它的实现类必须定义一个compareTo 方法，该方法提供与此接口的 getDelay 方法一致的排序。
- SynchronousQueue
  - 一个没有容量的阻塞队列
  - 应用
    - 交换工作，生产者的线程和消费者的线程同步以传递某些信息、事件或者任务
  - 难搞懂，与Exchanger 有一拼
- LinkedTransferQueue
  - 链表组成的的无界阻塞队列
  - 相当于ConcurrentLinkedQueue、SynchronousQueue (公平模式下)、无界的LinkedBlockingQueues等的超集
  - 预占模式
    - 有就直接拿走，没有就占着这个位置直到拿到或者超时或者中断
- LinkedBlockingDeque
  - 由链表组成的双向阻塞队列
  - 容量可选，在初始化时可以设置容量防止其过度膨胀，如果不设置，默认容量大小为Integer.MAX_VALUE
  - 运用
    - “工作窃取”模式

- 线程池
  - 好处
    - 降低资源消耗
      - 通过重复利用已创建的线程降低线程创建和销毁造成的消耗
    - 提高响应速度
      - 当任务到达时，任务可以不需要等到线程创建就能立即执行
    - 提高线程的可管理性
      - 进行统一分配、调优和监控
  - Executor
    - Executors

- 静态工厂类，提供了Executor、ExecutorService、ScheduledExecutorService、ThreadFactory、Callable 等类的静态工厂方法
- ThreadPoolExecutor
  - 参数含义
    - corePoolSize
      - 线程池中核心线程的数量
    - maximumPoolSize
      - 线程池中允许的最大线程数
    - keepAliveTime
      - 线程空闲的时间
    - unit
      - keepAliveTime的单位
    - workQueue
      - 用来保存等待执行的任务的阻塞队列
      - 使用的阻塞队列
        - ArrayBlockingQueue
        - LinkedBlockingQueue
        - SynchronousQueue
        - PriorityBlockingQueue
    - threadFactory
      - 用于设置创建线程的工厂
      - DefaultThreadFactory
    - handler
      - RejectedExecutionHandler，线程池的拒绝策略
      - 分类
        - AbortPolicy：直接抛出异常，默认策略
        - CallerRunsPolicy：用调用者所在的线程来执行任务
        - DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务
        - DiscardPolicy：直接丢弃任务
  - 线程池分类
    - newFixedThreadPool
      - 可重用固定线程数的线程池
      - 分析
        - corePoolSize和maximumPoolSize一致
        - 使用“无界”队列 LinkedBlockingQueue
        - maximumPoolSize、keepAliveTime、 RejectedExecutionHandler无效
    - newCachedThreadPool
      - 使用单个worker线程的Executor
      - 分析

- corePoolSize和maximumPoolSize被设置为1
- 使用LinkedBlockingQueue作为workerQueue
- newSingleThreadExecutor
  - 会根据需要创建新线程的线程池
  - 分析
    - corePoolSize被设置为0
    - maximumPoolSize被设置为Integer.MAX_VALUE
    - SynchronousQueue 作为WorkerQueue
    - 如果主线程提交任务的速度高于maximumPool中线程处理任务的速度时，CachedThreadPool会不断创建新线程 ， 可能会耗尽CPU和内存资源
- 任务提交
  - Executor.execute()
  - ExecutorService.submit()
- 任务执行
  - 执行流程
- 线程池调优
  - 两种模型
- 线程池监控

- ScheduledThreadPoolExecutor
  - 继承自ThreadPoolExecutor
  - 给定的延迟之后运行任务，或者定期执行任务
  - 内部使用DelayQueue来实现 ， 会把调度的任务放入DelayQueue中。DelayQueue内部封装PriorityQueue，这个PriorityQueue会对队列中的ScheduledFutureTask进行排序

- Future
  - 异步计算
  - Future
    - 提供操作
      - 执行任务的取消
      - 查询任务是否完成
      - 获取任务的执行结果
  - FutureTask
    - 实现RunnableFuture接口，既可以作为Runnable被执行，也可以作为Future得到Callable的返回值
    - 内部基于AQS实现

- JVM体系
  - JVM内存模型
    - 程序计数器
    - 堆
    - 栈
    - 虚拟机栈

- 方法区
- 常用JVM排查命令
- 常见GC算法
  - 串行
  - 并行
  - G1
- GC判断算法
  - 引用计数法
  - 可达性分析法