JUC

- **进程&线程**

1

- 进程

  1. 系统进行资源分配和调度的基本单位
  2. 线程的容器

- 线程

  1. 操作系统进行运算调度（分配处理器时间资源）的最小单位
  2. 进程中一个单一顺序的控制流

  - State
    - New

      新建，还没有开始运行
    - RUNNABLE

      准备就绪，已被 JVM 执行，但需要等待操作系统的其他资源（如：处理器）
    - BLOCKED

      阻塞，等待获取管程锁（lock 或 wait）
    - WAITING

      等待其他线程提供的资源，Object.wait, Thread.join, LockSupport#park，需要 Object.notify, LockSupport#unpark 唤醒
    - TIMED_WAITING

      等待一段时间，sleep, wait, join, parkNanos, parkUntil
    - TERMINATED

      终止线程，线程执行结束
  - wait vs. sleep

    1. sleep 是 Thread#sleep，wait 是 Object.wait
    2. sleep 不会释放锁，也不需要占有锁，wait 会释放锁，而且需要先占有锁才能 wait
    3. 都可以被 interrupted 终端
  - 用户线程 vs. 守护线程

    用户线程：普通线程
    守护线程：setDaemon,如 GC 线程

    当主线程结束时，存在用户线程存活，JVM 存活
    当不存在用户线程，只存在守护线程，JVM 结束
  - 创建方式

    1. 实现 Thread
    2. Runnable
    3. Callable - FeatureTask

- 管程

  monitor，JVM 中同步是基于进入和退出管程对象实现的，每个对象都有一个管程对象，随着 Java 对象一同创建和销毁
- 并发 vs. 并行

  并发：宏观上的同一时刻访问统一资源
  并行：时间点上多想工作一起执行

- **lock**

2

Lock 接口:
void lock();
void lockInterruptibly() throw InterruptedException;
boolean tryLock();
boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
void unlock();
Condition newCondition();

- 锁对象
  - 锁对象

    同步代码块
  - 锁 this 实例

    同步 this 实例的方法
  - 锁 Class 对象

    同步 static 方法
- 锁的类型
  - 公平锁

        - 公平
          按顺序，直接入队
        - 非公平
          优先级，先抢，synchronized 非公平，ReentrantLock(false)
      - 读写锁
        ReadWriteLock <- ReentrantReadWriteLock
        锁降级：获取写锁 -> 获取读锁 -> 释放写锁(降级为读锁) -> 释放读锁
        - 写锁
          独占锁
        - 读锁
          共享锁
      - 乐观锁
        - 乐观锁
        - 悲观锁
      - 可重入锁
        递归锁，线程可以进入任意已拥有锁锁住的代码块，避免死锁
        - 不可重入锁
          - StampedLock
        - 可重入锁
      - 自旋锁
        SpinLock，减少线程切换，不阻塞
        循环消耗 CPU
    - synchronized vs. Lock
      1. synchronized 是 JVM 层面的 Java 语言关键字；Lock 是一个类，api 层面
      2. synchronized 异常或执行结束后系统自动释放锁；Lock 必须用户手动释放锁，否则可能出现死锁
      3. synchronized 不可以终端，Lock 可以通过 tryLock 或 lockInterruptibly 中断
      4. synchronized 只能随机或全部唤醒，Lock 可以通过绑定多个 Condition 唤醒指定线程
      5. synchronized 只能是非公平锁，Lock 默认非公平，可以构造公平锁
      6. Lock 的读写锁实现可以优化多读场景性能
      - tryLock
        额外的尝试机制，失败不会死锁
    - deadlock
      持有自己的，试图获取别人的
      - jps -l | grep app-name | awk '{print $1}'
      - jstack pid
        Found on Java-level deadlock...
- **线程通信**
  3
  - synchronized
    - synchronized
    - wait
      IllegalMonitorStateException
      当在非 synchronized 范围内调用 wait、notify 时的异常
    - notify/notifyAll
      只对当前 wait 状态的线程有效
  - Lock
    - lock
    - await
      超时唤醒
      同 wait，await 和 signal 需在 lock 范围内调用，否则抛出 IllegalMonitorStateException
    - signal/signalAll
  - LockSupport
    创建锁和其他同步类的基本线程阻塞原语
    每个线程持有一个 permit，park 消耗一个 permit，unpark 获得一个 permit，permit 的数量最多是1
    - park
    - unpark
- **Collection**
  4
  ConcurrentModifyException
  ---add()---
  ensureCapacityInternal(size+1);
  elementData[size++]=e;
  return true;
  - List

- Vector

  ---add()---
  synchronized boolean add(E e){
  ...
  }
- Collections.syncronizedList
- CopyOnWriteArrayList

  ---java.util.concurrent---
  写时复制，读写分离，写线程会阻塞，读效率较高

  volatile 保证读取的数据总是最小的
  独占锁保护数据修改操作

- HashSet
  - CopyOnWriteArraySet

    底层实现是 CopyOnWriteArrayList
- HashMap
  - ConcurrentHashMap

    synchronized + 判空机制优化
  - HashTable

    同 Vector，通过 synchronized 关键字实现，效率较低

- **工具类**

5

  - CountDownLatch

    减少计数
    1. await 阻塞一个线程，直到其他线程达到计数器为0的条件
    2. countDown 计数器减 1
  - CyclicBarrier

    循环栅栏
    await 阻塞所有线程，每 await 依次计数器 +1，直到一起达到设定值，一起启动
  - Semaphore

    信号灯，类似线程池
    1. acquire 获取许可，许可不够时阻塞，直到获得被释放的许可后启动
    2. release 释放许可
- **BlockingQueue**

6

  取空添满时阻塞，相比 Condition 手动阻塞，BlockingQueue 是自动阻塞的
  - 类型
    - ArrayBlockingQueue

      数组 + 2*整型（头尾位置）
      读写采用同一个锁对象

    - LinkedBlockingQueue

      链表 + 整型（容量）
      分离锁，读写锁分离，元素需要封装在 Node

    - DelayQueue

      获取数据的操作会被延迟指定时间
    - PriorityBlockingQueue

      优先级阻塞队列，通过构造函数的 Compator 参数决定，不阻塞生产者，读空时阻塞
      公平锁
    - SynchronousQueue

      公平模式：公平锁 + FIFO
      非公平模式：非公平锁 + LIFO
    - LinkedTransferQueue

      链表 + 预占
      消费者取数时，队列不为空则直接取走，为空则阻塞消费者线程，并构造一个空 Node，直到生产者填充该 Node
    - LinkedBlockingDeque

      链表，双向阻塞
  - 操作
    - add,remove,element

      添加，移除，查看队首 - 抛异常
    - offer,poll,peek

      添加，取出，取队首 - bool/null，可带超时时间
    - put,take

      添加，取出
- **TheadPool**

7

1. 降低线程创建与销毁的消耗
2. 无需等待线程创建，响应快
3. 统一的分配，调优，监控

- Executors
  - newFixedThreadPool
  - newSingleThreadExecutor

    顺序，无界队列
  - newCacheThreadPool

    最大线程数：Integer.MAX_VALUE
    默认回收时间：1 min
  - newScheduledThreadPool

    一个 Timer 一个 Thread
  - newWorkStealingPool

    ForkJoinPool
- ThreadPoolExecutor
  - 七参数
    - corePoolSize

      核心常驻线程数
    - maximumPoolSize

      最大线程数
    - keepAliveTime

      多余空闲线程存活时间
    - unit
    - workQueue

      阻塞队列
    - threadFactory

      线程工厂类
    - handler - RejectedExecutionHandler

      拒绝策略
      触发：Task 数 > workQueue.size()+maximumPoolSize
      - AbortPolicy

        抛异常
      - DiscardPolicy

        直接丢弃
      - CallerRunsPolicy

        返回调用者自己执行，如果任务提交过快，可能导致进程阻塞
      - DiscardOldestPolicy

        丢弃最久
  - 合理参数
    - CPU 密集型

      CPU 核数 + 1
    - IO 密集型

      CPU 核数/(1-阻塞系数)
      阻塞系数在 0.8~0.9

- **Future**

8

---FutureTask---
cancel
get, 阻塞直到得到返回结果或异常
isDone

---缺点---
1. 不支持手动完成
2. 不支持进一步非阻塞调用（回调）
3. 不支持链式调用
4. 不支持多个 Feature 合并前置作用
5. 不支持异常处理

- CompletableFuture

  定制流程
  - runAsync

    无返回
  - supplyAsync

    有返回
  - thenApply

    线程串行化

        - thenApplyAsync
        - thenAccept
          消费处理结果，无返回
        - exceptionally
          异常处理
        - handle
          类似 thenApply/thenRun，同事可以处理异常
        - thenCompose
          合并
        - anyOf
          任意一个返回，整个任务结束
        - allOf
          所有任务完成后做一些事
    - ForkJoinTask
      把大任务拆分成多个小任务，分治。
- **AQS**
  9
    - lock
    - acquire
    - tryAcquire(arg)
    - addWaiter(Node.EXCLUSIVE)
    - acquireQueue(addWaiter(Node.EXCLUSIVE), arg)
- **volatile**
  轻量级同步机制
    - JMM
      描述程序访问变量的一组规范
        - 可见性
          总线嗅探

          ---MESI 协议---
          当 CPU 写共享数据（其他CPU也存有该变量复本）时，会发出信号通知其他 CPU 将该内存变量缓存设置为无效
          其他 CPU 读取无效变量时，会从内存中重写读取
        - 原子性
        - 有序性
    - 可见性
    - 不保证原子性
        - Atomic
    - 禁止指令重排
        - MemoryBarrier，内存屏障
- **CAS**
  比较并交换
    - 循环时间长，CPU 开销大
    - 只能保证一个对象原子性
    - ABA 问题
        - AtomicStampedReference
          时间戳原子引用