- Java并发多线程
  - JUC 和锁机制
    - synchronized关键字
      - synchronized是一种互斥锁，由JVM实现，解决多个线程之间访问资源的同步性，保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。
      - 使用方式
        - 修饰代码块[对象/类]
          - 对**给定对象加锁**，进入同步代码块前要获得给定对象的锁 synchronized(this){}
          - 对**给定类加锁**，进入同步代码块前要获得当前class的锁 synchronized(XXClass.class){}
        - 修饰实例方法
          - public synchronized void func(){}
          - 作用于**当前对象实例加锁**，进入同步代码前要获得 当前对象实例的锁
        - 修饰静态方法
          - public static synchronized void func(){}
          - 给**当前类加锁**，会作用于类的所有对象实例 ，进入同步代码前要获得 当前 class 的锁。
      - JDK1.6之前synchronized属于重量级锁
        - **虚拟机中，对象在内存中包括对象头、实例数据和对齐填充三部分，对象头包含MarkWord和类型指针，Mark Word会记录对象关于锁的信息**
        - 每个对象都会有一个与之对应的**monitor对象监视器**，monitor中存储着当前持有锁的线程以及等待锁的线程队列
        - 线程进入同步代码块/方法，monitor对象就会把当前进入的线程Id进行存储 ，设置Mark Word的monitor对象地址 ，并把阻塞的线程存储到monitor的等待线程队列中
        - **加锁依赖底层操作系统的 mutex 相关指令实现 ，所以会有用户态和内核态之间的切换 ，性能损耗十分明显**
        - 多线程同时进入临界区
      - JVM对synchronized优化
        - JDK1.6 以后引入偏向锁和轻量级锁在JVM层面实现加锁的逻辑 ，不依赖底层操作系统 ，就没有切换的消耗
        - 自旋锁
          - 让一个线程在**请求一个共享数据的锁时执行忙循环（自旋）一段时间**，如果在这段时间内能获得锁 ，就可以避免进入阻塞状态
        - 锁消除
          - 锁消除是指**对于被检测出不可能存在竞争的共享数据的锁进行消除。**
          - 通过逃逸分析来支持，如果堆上的共享数据不可能逃逸出去被其它线程访问到 ，那么就可以把它们当成私有数据对待 ，也就可以将它们的锁进行消除。
        - 锁粗化
          - 如果一系列的连续操作都对同一个对象反复加锁和解锁 ，频繁的加锁操作就会导致性能损耗。
          - **把加锁的范围扩展（粗化）到整个操作序列的外部**
        - 偏向锁
          - 引入偏向锁和轻量级锁之后 ，MarkWord对锁的记录状态：无锁unlocked/偏向锁biasble/轻量级锁lightweight locked/重量级锁inflated
          - JVM偏向于第一个获得锁的线程（没有竞争的环境），因为大部分情况下，都会是同一个线程进入同一块同步代码块的
          - 有线程执行同步代码，**比对线程ID是否相等**，相等则直接获取得到锁；否则使用 **CAS 操作将MarkWord的线程ID设置为当前线程ID**，操作成功获得锁
          - CAS操作失败，说明有竞争环境，此时会对偏向锁撤销，升级为轻量级锁。
          - 只有一个线程进入临界区
        - 轻量级锁
          - 在轻量级锁状态下，当前线程会在栈帧下创建Lock Record，**Lock Record 会把Mark Word的信息拷贝进去**，且有个Owner指针指向加锁的对象
          - 线程执行到同步代码时，则使用CAS将Mark Word指向LockRecord，假设CAS修改成功，则获取得到轻量级锁
          - 假设修改失败，则自旋（重试），自旋一定次数后，则升级为重量级锁
          - 多个线程交替进入临界区
    - ReentrantLock
      - ReentrantLock 可重入锁，**指的是一个线程能够对临界区共享资源进行重复加锁**
      - Lock lock = new ReentrantLock(); lock.lock(); lock.unlock();
      - 和synchronized比较
        - "可重入锁" 指的是自己可以再次获取自己的内部锁。 两者都是可重入锁
        - 锁的实现
          - synchronized 是 JVM 实现的
          - ReentrantLock 是 JDK 实现的，（也就是 API 层面，需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成）
        - 性能
          - 在Synchronized优化以前，synchronized的性能是比ReentrantLock差很多的，自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多
        - 功能
          - Synchronized的使用比较方便，由编译器去保证锁的加锁和释放，而ReentrantLock需要手工声明来加锁和释放锁
      - RL高级功能
        - 等待可中断
          - ReentrantLock提供了一种能够中断等待锁的线程的机制，通过 lock.lockInterruptibly() 来实现这个机制。
        - 可实现公平锁
          - ReentrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。
          - 公平锁就是竞争环境下先等待的线程先获得锁
          - 公平和非公平的区别就是：线程执行同步代码块时，是否会去尝试获取锁，尝试获取锁的就是非公平锁，进队列排队等待的是公平锁
          - ReentrantLock默认情况是非公平的，可以通过 ReentrantLock类的ReentrantLock(boolean fair)构造方法来制定是否是公平的。
        - 可实现选择性通知
          - synchronized关键字与wait()和notify()/notifyAll()方法相结合可以实现等待/通知机制。
          - synchronized
            - synchronized相当于整个 Lock 对象中只有一个Condition实例，所有的线程都注册在它一个身上
            - 如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题
          - ReentrantLock类借助于Condition接口与newCondition()方法也可以实现等待/通知机制
          - Condition实例的signalAll()方法 只会唤醒注册在该Condition实例中的所有等待线程。
    - AQS
      - AbstractQueuedSynchronizer抽象队列同步器，在java.util.concurrent.locks包下，是创建锁和同步器的框架，维护了一个FIFO的双向队列和State状态变量
      - 核心思想
        - 如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。
        - AQS 是用 CLH 队列锁(FIFO)实现该机制，即将暂时获取不到锁的线程加入到队列中 如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制
        - CLH队列是一个虚拟的双向队列（不存在队列实例，仅存在结点之间的关联关系），AQS 是将每条请求共享资源的线程封装成队列的一个结点来实现锁的分配
        - AQS使用一个 volatile 修饰的 int 类型的成员变量 state 来表示同步状态，修改同步状态成功即为获得锁，通过内置的 FIFO 队列来完成获取资源线程的排队工作。
        - volatile 保证了state在多线程之间的可见性和操作有序性， AQS 使用 CAS 对该同步状态State进行原子操作实现对其值的修改，保证原子性
      - AQS原理
        - 只有一个线程能执行，如 ReentrantLock
        - AQS对资源的共享方式
          - Exclusive（独占）
            - 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁
            - 非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的
          - Share（共享）
            - 多个线程可同时执行，如CountDownLatch、Semaphore、 CyclicBarrier
        - 自定义同步器
          - 不同的自定义同步器争用共享资源的方式也不同，自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可
          - 具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS 已经在顶层实现好
        - ReentrantLock
          - 公平锁实现
            - 若state为0，先去查看等待队列中是否有线程，若无，则采用CAS操作尝试将state设为1，若成功，则获得该锁并返回true，否则返回false。
            - 若state非0，则判断当前线程是否持有该锁，有则对state加1返回true，否则返回false。
          - 非公平锁实现
            - 若state=0，则使用CAS操作尝试将state设为1，若成功，则当前线程获得锁并返回true，否则返回false；
            - 若state非0，判断当前线程是否持有该锁，有则state+1，返回true，否则直接返回false。
    - AQS组件
      - 线程同步的工具类
        - Semaphore信号量
          - Semaphore可以指定多个线程同时访问某个资源
        - CountDownLatch倒计时器
          - CountDownLatch 是一个同步工具类，用来协调多个线程之间的同步，允许cnt个线程阻塞在一个地方，直到所有的线程的任务执行完毕【在主线程/调用线程调用await()】
          - 维护了一个计数器 cnt，每次调用 countDown() 方法cnt-1，当cnt=0，调用 await() 方法而在等待的线程就会被唤醒。
          - 调用await()通常是主线程，主线程阻塞
          - 基于AQS实现，将构造CountDownLatch的入参传递至state，countDown()就是在利用CAS将state减-1，await()实际就是让头节点一直在等待state为0时，释放所有等待的线程
          - 实例
            - 处理 6 个文件，这 6 个任务都是没有执行顺序依赖的任务，但是需要返回给用户的时候将这几个文件的处理的结果进行统计整理。
            - 定义了一个线程池和 count 为 6 的CountDownLatch对象 ，使用线程池处理读取任务，每一个线程处理完之后就将 count-1且调用CountDownLatch对象的 await()方法
            - 直到所有文件读取完之后，才会接着执行后面的逻辑。
        - CyclicBarrier循环栅栏
          - CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，区别在于CyclicBarrier可以复用
          - 当线程到达某种状态后，暂停下来等待其他线程，直到所有线程均到达，才继续执行【在任务线程调用await()】
          - 调用await()是在任务线程调用的，任务线程阻塞
          - ReentrantLock + Condition等待唤醒，维护了count和parties变量。每次调用await将count-1，并将线程加入到condition队列上。
          - 等到count为0时，将condition队列的节点添加到AQS队列并全部唤醒，将parties的值重新赋值为count的值（实现复用）
      - 其他组件
        - FutureTask
        - BlockingQueue
        - ForkJoin
    - Atomic原子类
      - 具有原子/原子操作特征的类
      - 四类原子类
        - 基本类型
          - 使用原子的方式更新基本类型
          - AtomicInteger/AtomicLong/AtomicBoolean
          - AtomicInteger主要利用CAS+volatile和native方法保证原子操作
        - 数组类型
          - 使用原子的方式更新数组里的某个元素
          - AtomicIntegerArray/AtomicLongArray/AtomicReferenceArray
        - 引用类型
          - AtomicReference
          - AtomicStampedReference 原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。
          - AtomicMarkableReference 原子更新带有标记位的引用类型
        - 对象的属性修改类型
          - AtomicIntegerFieldUpdater
          - AtomicLongFieldUpdater
          - AtomicReferenceFieldUpdater
  - 线程池
    - 使用线程池的好处
      - 线程池提供了一种限制和管理资源(包括执行一个任务)的方式，是一个典型的生产者-消费者模式
      - 降低资源消耗、提高响应速度、提高线程的可管理性
      - 提高线程的复用和固定线程的数量
    - Executor框架
      - 包括线程池的管理、线程工厂、队列和拒绝策略等
      - 框架结构
        - 执行任务需要实现的 Runnable 接口 或 Callable接口。Runnable 接口或 Callable 接口 实现类都可以被 ThreadPoolExecutor 或 ScheduledThreadPoolExecutor 执行 任务
        - ScheduledThreadPoolExecutor ThreadPoolExecutor AbstractExecutorService
        - ExecutorService Executor 任务的执行
        - ScheduledExecutorService
        - Future 接口以及 Future 接口的实现类 FutureTask 类都可以代表异步计算的结果。 异步计算的结果
      - 使用流程
        - 主线程首先要创建实现 Runnable 或者 Callable 接口的任务对象。
        - 把创建完成的实现 Runnable/Callable接口的 对象直接交给 ExecutorService 执行 ExecutorService.execute(Runnable command)/ExecutorService.submit(Runnable task)
        - 如果执行 ExecutorService.submit（...），ExecutorService 将返回一个实现Future接口的对象
        - 最后，主线程可以执行 FutureTask.get()方法来等待任务执行完成，或者执行FutureTask.cancel()来取消任务的执行
    - ThreadPoolExecutor
      - ThreadPoolExecutor是Executor的实现，更能了解线程池运行的规则，避免资源耗尽的风险
      - ThreadPoolExecutor executor = new ThreadPoolExecutor(); Runnable worker = new MyRunnable("" + i); executor.execute(worker);
      - 构造函数参数
        - 核心线程数定义了最小可以同时运行的线程数量 corePoolSize
        - 当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数 maximumPoolSize
        - 当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，新任务就会被存放在队列中 workQueue
        - 当线程池中的线程数量大于 corePoolSize，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，直到等待的时间超过 keepAliveTime才会被回收销毁 keepAliveTime
        - keepAliveTime 参数的时间单位 unit
        - executor创建新线程时候会用到 threadFactory
        - 饱和策略 handler
      - 饱和策略
        - 如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任务时，ThreadPoolTaskExecutor 定义一些策略
        - 抛出 RejectedExecutionException来拒绝新任务的处理。 AbortPolicy拒绝策略
        - 直接在调用execute方法的线程中运行被拒绝的任务，如果执行程序已关闭，则会丢弃该任务。 CallerRunsPolicy执行策略
        - 不处理新任务，直接丢弃掉 DiscardPolicy丢弃策略
        - 丢弃最早的未处理的任务请求 DiscardOldestPolicy丢弃最老策略
      - 创建线程池
        - 通过ThreadPoolExecutor构造函数实现（推荐）
        - 通过Executor框架的工具类Executors实现三类ThreadPoolExecutor
          - 可重用固定线程数的线程池 FixedThreadPool
          - 只有一个线程的线程池 SingleThreadExecutor
          - 根据需要创建新线程的线程池 CachedThreadPool
    - ScheduledThreadPoolExecutor
      - 主要用来在给定的延迟后运行任务，或者定期执行任务。
    - 几个对比
      - Runnable和Callable
        - Runnable 接口不会返回结果或抛出检查异常
        - Callable仅在 Java 1.5 中引入，会返回结果或抛出检查异常
      - execute()和submit()
        - execute()方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；
        - submit()方法用于提交需要返回值的任务，线程池会返回一个 Future 类型的对象
      - submit()
        - 通过这个 Future 对象可以判断任务是否执行成功，并且可以通过 Future 的 get()方法来获取返回值
        - get()方法会阻塞当前线程直到任务完成，而使用 get（long timeout，TimeUnit unit）方法的话，如果在 timeout 时间内任务还没有执行完，就会抛出异常
      - shutdown()和shutdownNow()
        - shutdown()：关闭线程池，线程池的状态变为 SHUTDOWN。线程池不再接受新任务了，但是队列里的任务得执行完毕。
        - shutdownNow()：关闭线程池，线程的状态变为 STOP。线程池会终止当前正在运行的任务，并停止处理排队的任务并返回正在等待执行的 List
  - 多线程基础
    - 进程和线程
      - 进程
        - 资源分配的基本单位
        - Java 中，启动 main 函数就是启动了一个 JVM 的进程，而 main 函数所在的线程就是这个进程中的一个线程，也称主线程。
      - 线程
        - 资源调度的基本单位
        - 同类的多个线程共享进程的堆和方法区(jdk1.8 元空间)资源
        - 线程有自己的程序计数器、虚拟机栈和本地方法栈
          - 程序计数器私有为了线程切换后能恢复到正确的执行位置
          - 虚拟机栈和本地方法栈私有为了保证线程中的局部变量不被别的线程访问到
    - 多线程
      - 使用多线程来提高系统的资源利用率
      - **单核时代，多线程主要是为了提高单进程的CPU和I/O的效率；多核时代，多线程主要是为了提高进程利用多核CPU的能力**
      - 但多线程并发变成可能会遇到内存泄漏、死锁、线程不安全等问题
    - 线程实现方式
      - **实现 Runnable 和 Callable 接口的类只能当做一个可以在线程中运行的任务，不是真正意义上的线程，因此最后还需要通过 Thread 来调用。**
      - 实现Runnable接口
        - Runnable实现类实现接口的run()方法
        - 使用Runnable实例创建一个Thread实例，调用Thread实例的start()方法来启动线程
        - 无返回值
      - 实现Callable接口
        - 实现call方法
        - 有返回值，返回值通过FutureTask进行封装
      - 继承Thread类
        - 实现run方法
        - 当调用 start() 方法启动一个线程时，虚拟机会将该线程放入就绪队列中等待被调度，当一个线程被调度时会执行该线程的 run() 方法。
    - 线程生命周期和状态
      - **线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换**
      - 六种状态
        - 新建New 线程被构建，还没有调用start()方法
        - 可运行RUNABLE
          - READY **调用Thread.start()方法进入READY状态**
          - RUNNING **系统调用获得时间片进入RUNNING状态，时间片结束，调用Thread.yield()方法**
        - 无限期等待WAITING 线程执行Object.wait()方法进入WAITING状态，需要其他线程通知Object.notify()才能返回RUNABLE状态
        - 限期等待TIMED_WAITING 在WAITING状态基础上增加超时限制，比如使用Thread.sleep(long millis)或者Object.wait(long millis)
        - 阻塞BLOCKED 当线程调用同步synchronized方法，没有获取到锁的情况下会进入BLOCKED
        - 死亡TERMINATED 执行完Runnable的run()方法之后进入TERMINATED
    - 线程方法
      - **Thread.start() 启动线程并使得线程进入就绪状态，等时间片分配自动执行run()方法，如果直接调用run()方法只会当成main()线程普通方法执行，不会在某个线程中执行**
      - **Thread.sleep()** 休眠当前正在执行的线程，可能会抛出InterruptedException，异常不能跨线程传播到main()，必须本地处理，**sleep()不释放锁**
      - **Thread.yield()** 对静态方法 Thread.yield() 的调用声明了当前线程已经完成了生命周期中最重要的部分，可以切换给其它线程来执行。
      - 线程协作/通信
        - **Thread.join()** 在线程中调用另一个线程的 join() 方法，会将当前线程挂起，而不是忙等待，直到目标线程结束。
        - **Object.wait()**
          - 调用wait()使得线程等待某个条件满足，**等待时挂起，释放锁**，其他线程调用notify()或notifyAll()来唤醒挂起线程
          - wait(long millis)和sleep(long millis)都是到点苏醒
          - 只能在同步方法或者同步控制块中使用，用于线程间交互通信
        - **java.util.concurrent.Condition.await()** 可以在 Condition 上调用 await() 方法使线程等待，其它线程调用 signal() 或 signalAll() 方法唤醒等待的线程。
          - 相比于 wait() 这种等待方式，await() 可以指定等待的条件，因此更加灵活。
      - sleep和wait方法的区别联系
        - 区别
          - **sleep()方法没有释放锁，wait()方法释放了锁**
          - **wait() 通常被用于线程间交互/通信，sleep() 通常被用于暂停执行。**
          - wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法。
          - sleep() 方法执行完成后，线程会自动苏醒。或者可以使用 wait(long timeout) 超时后线程会自动苏醒。
        - **联系** 两者都可以暂停线程的执行。
    - 线程上下文切换
      - 线程在执行过程中会有自己的运行条件和状态（也称上下文），比如程序计数器，栈信息等。
      - 占用CPU状态退出
        - 主动让出CPU，比如sleep() wait()
        - 时间片用完
        - 调用阻塞类型的中断
        - 被终止或结束运行
      - 占用CPU退出的前三种会发生线程切换，线程切换需要保存当前线程的上下文，留待线程下次占用时恢复现场
    - 线程安全
      - 定义：多个线程不管以何种方式访问某个类，并且在主调代码中不需要进行同步，都能表现正确的行为。
      - 实现方式
        - 不可变 **不可变的对象一定是线程安全的**
          - final修饰的基本数据类型、String、枚举类型
          - Number部分子类
            - Long、Double等数值包装类型
            - BigInteger和BigDecimal等大数据类型
          - Number的原子类AtomicInteger和AtomicLong可变
          - 集合类型可以使用Collections.unmodifiableXXX()来获取一个不可变的集合
        - 线程同步机制
          - 同步/异步
            - 同步：发出一个调用时，在没有得到结果之前，调用不返回
            - 异步：发出一个调用之后，调用直接返回，通过回调函数来返回结果
          - 互斥同步(阻塞同步)
            - **只有一个线程获得锁执行操作，其他线程等待锁**
            - synchronized和ReentrantLock
            - 线程阻塞和唤醒会带来性能问题
            - 悲观锁互斥同步属于悲观的并发策略，总是认为只要不去做正确的同步措施就会出问题
          - 非阻塞同步
            - **所有线程直接执行操作，没有竞争则操作成功，否则自旋**
            - 使用基于冲突检测的乐观并发策略：先执行操作，如果没有其他线程竞争则操作成功，否则采取补偿措施（不断重试，直到成功）
            - 乐观锁需要操作和冲突检测两步具备**原子性**，不能使用互斥同步，依靠硬件完成
            - 硬件支持的原子性操作CAS(Compare And Swap)
              - CAS 指令需要有 3 个操作数，分别是内存地址 V、旧的预期值 A 和新值 B；只有当 V 的值等于 A，才将 V 的值更新为 B。
              - ABA问题：如果一个变量初次读取是 A 值，它的值被改成了 B，后来又被改回A，CAS 误认为它从来没有被改变过。 添加版本号
        - 无同步方案
          - 不涉及共享数据，不需要同步措施来保证正确性
          - 栈封闭 局部变量存储在虚拟机栈，线程私有
          - 线程本地存储ThreadLocal
            - 实现了线程的数据隔离
            - 原理
              - 每个Thread都有一个ThreadLocal.ThreadLocalMap对象，不存在多线程竞争
              - 当调用一个 ThreadLocal 的 set(T value) 方法时，先得到当前线程的 ThreadLocalMap 对象，然后将 ThreadLocal->value 键值对插入到该 Map 中。
              - ThreadLocalMap可以存储以ThreadLocal为 key ，Object 对象为 value 的键值对。
            - 内存泄漏
              - ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用,而 value 是强引用。
              - 如果 ThreadLocal 没有被外部强引用的情况下，在垃圾回收的时候，key 会被清理掉，而 value 不会被清理掉。
              - ThreadLocalMap 中就会出现 key 为 null 的 Entry，假如我们不做任何措施的话，value 永远无法被 GC 回收，就可能会产生内存泄露。
              - ThreadLocalMap 实现中已经考虑了这种情况，调用方法之后会清理key为null的记录
    - 线程死锁
      - 多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。
      - 四个条件：互斥、占有和等待、不可抢占、环路等待
  - Java内存模型JMM
    - 「Java内存模型」希望 屏蔽各种硬件和操作系统的访问差异，保证了Java程序在各种平台下对内存的访问都能得到一致效果。目的是解决多线程存在的原子性、可见性（缓存一致性）以及有序性问题。
    - 背景
      - 可见性：CPU架构下存在高速缓存，高速缓存解决CPU与内存速度不一致问题，但每个核心下的L1/L2缓存不共享(不可见)，即缓存不一致
      - 有序性：为了让CPU提高运算效率，处理器可能会对输入的代码进行「乱序执行」，也就是所谓的「指令重排序」
        - 编译器优化导致重排序
        - 指令集并行重排序
        - 内存系统重排序 CPU架构下有store buffer /invalid queue 缓冲区，这种「异步」
      - 原子性：一次对数值的修改操作往往是非原子性的
        - 使用「总线锁」 某个核心在修改数据的过程中，其他核心均无法修改内存中的数据，独占内存
        - 缓存一致性协议
          - 缓存行进行"加锁"，「缓存行」其实就是 高速缓存存储的最小单位，对缓存行标记状态，通过"同步通知"的方式，来实现(缓存行)数据的可见性
          - MESI协议
            - Modified （修改状态）、Exclusive （独占状态）、Share（共享状态）、Invalid（无效状态）
            - 判断「对象状态」，根据「对象状态」做不同的策略。关键就在于某个CPU在对数据进行修改时，需要「同步」通知其他CPU
          - MESI原理
            - 当每个CPU读取共享变量之前，会先识别数据的「对象状态」（修改、独占、共享、无效）
            - 修改状态：
              - 当前CPU正在修改该变量的值，同时会向其他CPU发送该数据状态为invalid(无效)的通知
              - 得到其他CPU响应后（其他CPU将数据状态从共享(share)变成invalid(无效)），当前CPU将高速缓存的数据写到主存，状态从modify(修改)变成exclusive(独占)
            - 独占状态：当前CPU将要得到的变量数据是最新的，没有被其他CPU所同时读取
            - 共享状态：当前CPU将要得到的变量数据还是最新的，有其他的CPU在同时读取，但还没被修改
            - 无效状态：重新从主存读取最新数据
          - CPU优化
            - 当CPU修改数据时，需要「同步通知」其他CPU，等待其他CPU响应接收到invalid(无效)后，它才能将高速缓存数据写到主存
            - 异步化
              - 把最新修改的值写到「store buffer」中，并通知其他CPU记得要改状态，随后CPU干其他事情，等到收到其它CPU发过来的响应消息，再将数据更新到主存中
              - 其他CPU接收到invalid通知时，接收到的消息放入「invalid queue」中，只要写到「invalid queue」就会直接返回告诉修改数据的CPU已经将状态置为「invalid」
            - 问题
              - 相同核心读写共享变量
                - 问题：CPU连续修改值，之前修改值还在store buffer
                - 解决：CPU在读取的时候，需要去「store buffer」看看存不存在，存在则直接取，不存在才读主存的数据。
              - 不同核心读写共享变量
                - CPU1修改了A值，并写到「store buffer」，通知CPU2 invalid操作，而CPU2可能还没收到invalid通知，就去做了其他的操作，导致CPU2读到的还是旧值。
                - 即便CPU2收到了invalid通知，但CPU1的值还没写到主存，那CPU2再次向主存读取的时候，还是旧值
                - 这种现象称为CPU乱序执行，即后面指令查不到前面指令的执行结果
            - 内存屏障
              - 解决「异步优化」导致「CPU乱序执行」/「缓存不及时可见」的问题
              - 分类
                - 屏障：插入一条"屏障指令"，使得缓冲区(store buffer/invalid queue)在屏障指令之前的操作均已被处理，进而达到 读写 在CPU层面上是可见和有序的。
                - 读屏障：CPU当发现读屏障的指令时，会把该指令「之前」存在于「invalid queue」所有的指令都处理掉
                - 写屏障：CPU当发现写屏障的指令时，会把该指令「之前」存在于「store Buffer」所有写指令刷入高速缓存
                - 全能屏障：包括读写屏障
      - 抽象结构
        - 线程之间的「共享变量」存储在「主内存」中，每个线程都有私有的「本地内存」，「本地内存」存储了该线程读/写共享变量的副本
        - 本地内存是Java内存模型的抽象概念，本地内存存储在高速缓存或者寄存器中
        - 线程对变量的所有操作都必须在「本地内存」进行，「不能直接读写主内存」的变量，不同线程之间的变量值传递需要通过主内存来完成
        - 8个主内存和本地内存交互操作
          - read：把一个变量的值从主内存传输到本地内存中
          - load：在 read 之后执行，把 read 得到的值放入本地内存的变量副本中
          - use：把本地内存中一个变量的值传递给执行引擎
          - assign：把一个从执行引擎接收到的值赋给本地内存的变量
          - store：把本地内存的一个变量的值传送到主内存中
          - write：在 store 之后执行，把 store 得到的值放入主内存的变量中
          - lock：标识「主内存」共享变量被某个线程独占
          - unlock：释放「主内存」共享变量
      - happen-before规则
        - 阐述「操作之间」的内存「可见性」，前一个操作的结果对后续操作必须是可见的，即「指令重排需要按照一定的规则」
        - 8条规则
          - 单一线程原则(Single Thread Rule)、监视器锁原则(Monitor Lock Rule)、Volatile变量原则(Valatile Variable Rule)、线程启动原则(Thread Start Rule)
          - 线程join原则(Thread Join Rule)、线程中断原则(Thread Interruption Rule)、对象终结原则(Finalizer Rule)、传递性(Transitivity)
      - volatile语义
        - 特性：可见性和有序性（禁止重排序）
        - 可见性：当对volatile变量执行写操作后，JMM会把本地内存中的最新变量值强制刷新到主内存；写操作会导致其他线程中的缓存无效
        - 有序性：volatile是通过编译器在生成字节码时，在指令序列中添加"内存屏障"来禁止指令重排序的。
        - 实现volatile有序性和可见性定义四种内存屏障规范
          - LoadLoad/LoadStore/StoreLoad/StoreStore
          - HotSpot虚拟机汇编实现是lock前缀指令
        - happen-before中volatile变量原则 对一个 volatile 变量的写操作相对于后续对这个 volatile 变量的读操作可见
        - 与Synchronized比较
          - Synchronized保证内存可见性和操作的原子性和有序性，Volatile保证内存可见性和操作有序性(禁止重排序)。
          - volatile不需要加锁，比Synchronized更轻量级，并不会阻塞线程（volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。）
          - volatile标记的变量不会被编译器优化，synchronized标记的变量可以被编译器优化（如编译器重排序的优化）.
          - volatile是变量修饰符，仅能用于变量，而synchronized是一个方法或块的修饰符。
      - Synchronized
        - synchronized关键字的语义JMM有两个规定，保证其实现内存可见性
        - 线程加锁时，清空工作内存中共享变量的值，使用共享变量需要从主内存中重新取值。
        - 线程解锁时，必须把工作内存中共享变量的最新值写到主内存中；