并发编程

出现原因：摩尔定律（处理器的性能每隔两年翻一倍）失效，计算机硬件多核时代来临

bug源头

- 可见性：一个线程对共享变量的修改，其他线程能立马看到
- 有序性：编译器按照程序输入的顺序执行代码
- 原子性：一个或多个操作执行过程中不被中断

解决办法

锁(借鉴美团的Java"锁"事)

- Java内存模型
  - 缓存一致性协议解决可见性问题：当一个线程成功修改共享变量后，其他线程的缓存立即失效然后从主内存中读取最新数据
  - Happens-Before规则
    - 程序的顺序性规则：按照程序顺序，前面的操作Happens-Before于后续的任意操作
    - volatile变量规则：对一个volatile变量的写操作，Happens-Before于后续对这个volatile变量的读操作
    - 传递性：如果A Happens-Before B，且B Happens-Before C，那么A HappensBefore C
    - 管程中锁的规则：对一个锁的解锁Happens-Before于后续对这个锁的加锁
    - 线程start()规则：主线程A启动子线程B后，子线程B能够看到主线程在启动子线程B前的操作
    - 线程join()规则：主线程A等待子线程B完成（主线程A通过调用子线程B的join()方法实现），当子线程B完成后（主线程A中join()方法返回），主线程能够看到子线程的操作
- 线程要不要锁住同步资源?
  - 锁住 悲观锁
  - 不锁柱 乐观锁
- 锁住同步资源失败, 线程要不要阻塞?
  - 阻塞
  - 不阻塞
    - 自旋锁
    - 自适应自旋锁
- 多个线程竞争同步资源的流程细节有没有区别?
  - 不锁柱资源，多个线程中只有一个能修改资源成功，其他线程会重试 无锁
  - 同一个线程执行同步资源时自动获取资源 偏向锁
  - 多个线程竞争同步资源时，没有获取资源的线程自旋等待锁释放 轻量级锁
  - 多个线程竞争同步资源时，没有获取资源的线程阻塞等待唤醒 重量级锁
- 多个线程竞争锁时要不要排队?
  - 排队 公平锁
  - 先尝试插队，插队失败后在排队 非公平锁
- 一个线程中的多个流程能不能获取同一把锁?
  - 能 可重入锁
  - 不能 不可重入锁
- 多个线程能不能共享一把锁
  - 能 共享锁
  - 不能 排他锁

并发工具类

- lock&condition
  - 支持中断：void lockInterruptibly() throws InterruptedException;
  - 支持超时：boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
  - 支持非阻塞获取锁：boolean tryLock();
  - 弥补sychronized的不足
- Semaphore
  - init()：设置计数器的初始值。
  - down()：计数器的值减1；如果此时计数器的值小于0，则当前线程将被阻塞，否则当前线程可以继续执行。
  - up()：计数器的值加1；如果此时计数器的值小于或者等于0，则唤醒等待队列中的一个线程，并将其从等待队列中移除。
- ReadWriteLock
  - 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
  - 写锁
- StampedLock
  - 乐观读
  - 悲观读锁
- CountDownLatch：一个线程等待多个线程，例如司机要等乘客都上车了才能开车
- CyclicBarrier：一组线程互相等待
- 原子类：无锁方案的实现，大多都实现了compareAndSet()方法
- 并发容器
  - Vector：每个方法都使用sychronized加锁，性能堪忧
  - HashTable：原理同vector
  - ConcurrentHashMap
    1. 尝试获取锁 try lock()
    2. 未获取到锁则尝试自旋获取锁
    3. 自旋到一定次数后如果还未获取到锁则使用sychronized来保证能够获取到锁
- Future
- CompletableFuture
- CompletetionService
- ForkJoin