线程安全

- **线程是cpu调度的最小单位，多线程可以充分的利用cpu的资源，但是线程之间的切换会带来的线程不安全的问题**
- **线程不安全原因**
  - 1.CPU 抢占式执行
  - 2.有些操作是非原子性的
    - 如 ++/--执行过程是非原子过程：
      load(读取)、calc(运行)、save(保存)
  - 3.指令重排序（这是编译器做的优化）
  - 4.内存不可见
    - 这是为了提高效率，JVM在执行过程中会尽可能的将数据储存在工作中执行
- **解决方案**
  - 1.volatile 关键字
    - 作用
      - 禁止指令从排序，保证内存可见性，但不能解决原子性问题
    - 实现原理
      - 操作完变量后将变量刷新到主内存中，并强制删除工作内存中的变量
  - 2.加锁
    - synchronized

      深入浅出synchronized - 简书

      Java中的每个对象都可以作为锁。

      1. 普通同步方法，锁是当前实例对象。
      2. 静态同步方法，锁是当前类的class对象。
      3. 同步代码块，锁是括号中的对象。

      - 怎么解决线程不安全问题的
        - 让关键代码排队执行，保证了原子性

          关键步骤：

          1. 尝试获取锁（如果没有拿到锁，排队等待）
          2. 释放锁

        - 释放锁时刷新变量到主内存中；
          获取锁时将线程对应的本地内存设置为无效，保证是从主内存中读取到变

量，保证了内存可见性。

- 实现原理
  - 1.从操作系统来看，依靠的是同步原语 mutex 进行同步保护的
    mutex再操作系统中就是c++的一个结构体，操作系统基于 CPU 的原子指令实现的 mutex 互斥锁
  - 2.从 JVM 来看，.java 变成 .class 时会加入一个monitor 监视器的进入和离开，就是加锁和释放锁
  - 3.从 Java 层面来看，是把锁对象放在了变量的对象头中，在对象头的偏向线程 id 中标志者当前锁的用有人
    在Hotspot中，对象在内存中分为3部分：对象头，实例数据，对齐填充
    对象头由 Mark Word 和 Kiss pointer 两部分组成
    mark word 储存了同步章台、标识、hashcode、GC 状态等
- JDK 1.6 锁升级的过程
  - JVM 将 synchronized 锁分为 无锁、偏向锁、轻量级锁、重量级锁 4种状态。会根据情况，进行依次升级。
    目的是尽量在用户空间去完成锁获取与释放，因为一旦进入"重量级锁"状态，那么将会调用内核空间，产生较大的开销
  - 无锁
    - 对共享资源不进行任何保护，或者是通过 CAS 操作进行的侠女同步
  - 偏向锁
    - 在没有发生竞争的时候，不需要同步操作。
      对象头的 mark word 中记录偏向锁 ID，让锁认识拥有它的线程
  - 轻量级锁
    - 短时间内能释放锁资源，存在少量的竞争时。
      通过 适应性自旋等待 的操作获取锁,因为自旋操作是在用户空间解决，避免了用户态和内核态之间的切换带来的消耗
  - 重量级锁
    - 原始的 Synchronized 的实现，对资源进行锁定，其他线程试图获取锁时，都会被阻塞
- Lock
  - 手动加锁和释放锁，在获取锁失败后可以去做对应的业务处理，不像 synchronized 只能死等
  - Lock 的加锁为什么要放在 try外边?
    - 1.如果放在 try 里边，try 中代码出现问题，就会执行 finally 里的释放锁操作，此时可能还没加锁
    - 2.释放锁的异常会覆盖业务的异常报错，不利于排错

- 2.释放锁的异常会覆盖业务的异常报错，不利于排错

- synchronized 和 lock 区别?
  - 1.synchronized 即可以修饰代码块，又可以修饰静态方法和普通方法而 lock 只能修饰代码块。
  - 2.synchronized 只有非公平锁的策略，而 lock 既可以是公平锁也可以是非公平锁（ReentrantLock 默认是非公平锁，也可以通过构造函数设置 true公平锁 false非公平锁）
  - 3. ReentrantLock 更加灵活（比如可以使用 tryLock，获取不到可以去执行别的业务）
  - 4. synchronized 是自动加锁和释放锁的，而 ReentrantLock 需要自己加锁和释放锁
- 死锁
  - 什么是死锁?
    - 多线程编程时，因为资源的抢占而造成线程的无限等待
    - 原因/死锁的条件：
      - 1. 互斥条件：一个资源只能被一个线程持有，当被一个线程拥有之后就不能被其他线程持有
      - 2. 请求拥有条件：一个线程持有了一个资源后，又试图请求另外一个资源
      - 3. 不可剥夺条件： 一个资源被一个线程拥有后，如果这个线程不释放此资源，那么其他线程不能强制获得次资源
      - 4. 环路等待条件： 多个线程在获取资源时形成了一个环形链
    - 例如：线程1 和线程2 分别拥有 a 和 b，线程1 又去获取锁 b，线程2 去获取锁 a
  - 怎么解决死锁问题?
    - 通过控制获取锁的顺序，破坏了环路等待条件
  - 怎么检测死锁?
    - 使用JDK bin 目录下的一些工具如
      - jconsole.exe
      - jvisualvm.exe
      - jmc.exe
- 3.使用 ThreadLocal（线程本地私有变量）
  -

自由主题

- 内核态和用户态的切换