对象的共享

- 可见性
  - 重排序
    - 在没有同步的情况下,编译器、处理器以及运行时等随时都有可能对操作的执行顺序进行一些意想不到的调整.
    - 在缺乏足够同步的多线程程序中,想要对内存操作的执行顺序进行判断,几乎无法得出正确的结论
  - 失效数据
  - 非原子的64位操作
    - 当线程在没有同步的情况下读取变量时,可能会得到一个失效值,但是至少这个值是由之前某个线程设置的值,而不是一个随机值.这种安全性保证也被称为最低安全性
    - 非volatile的64位数值的读写操作可以分解为两个32位操作
  - 加锁与可见性
    - 加锁的含义不仅仅局限于互斥行为,还包括内存可见性.为了确保所有线程都能看到共享变量的最新值,所有执行读取操作或者写操作的线程都必须在同一个锁上
  - Volatile变量
    - 编译器和运行时可以意识到这个变量是共享的
    - 不会将该变量上的操作与其它内存操作一起重排序
    - 变量不会被缓存在寄存器或者对其他处理器不可见的地方
    - 读取volatile变量时总会返回最新写入的值
    - 使用场景
      - 能简化代码的实现以及对同步策略的验证
      - 如果在验证正确性时需要对可见性进行复杂的判断,那么不要使用
      - 仅能保证可见性
      - 正确的使用方式
        - 确保它们自身状态的可见性
        - 确保它们所引用对象的状态的可见性
        - 标识它们所引用对象的状态的可见性

        - 标识一些重要的程序生命周期事件的发生(例如初始化或关闭)
    - 当且仅当满足以下所有条件时,才应该使用volatile变量
        - 对变量的写入操作不依赖变量的当前值,或者可以确保只有单个线程更新变量的值
        - 该变量不会与其它状态变量一起纳入不变性条件中
        - 在访问变量时不需要加锁

- 发布与逸出
    - 发布
        - 使对象能够在当前作用域之外的代码中使用
        - 发布内部状态可能会破坏封装性
    - 逸出
        - 某个不应该发布的对象被发布
    - 安全的对象构造过程
        - 不要在构造过程中使this引用逸出
        - 如果想在构造函数中注册一个事件监听或者启动线程,可以使用一个私有的构造函数和一个工厂方法

- 线程封闭
    - ad-hoc线程封闭

      You could decide on a set of conventions to restrict the use of shared state and then work to have all the code follow them. You agree on an order of who gets to do what and when that ensures locks aren't needed. If you manage it, that's great. However, you are using 100% human-level protection rather than anything language-level, which can be difficult. Part of an ad-hoc agreement could be that you share state in a very specific way; everything that's shared is marked as volatile and for each shared object, only one thread is ever allowed to be able to write to it. In a single-writer, multiple-reader scenario, you don't need synchronization if the variable is volatile. Very fragile and very subtle!

      So an example could be as simple as documenting something like this:

      ```
      // Don't use this object in other threads than Thread X
      private SomeType someObject;
      public SomeType getSomeObject() { return someObject; }
      or

      // Don't modify this from any other thread than Thread X.
      // So use it read-only for those other threads.
      private volatile int someNumber;
      ```

        - 维护线程封闭性的职责完全由程序实现
        - 非常脆弱
            - 没有任何一种语言特性,例如可见性修饰符或者局部变量,能将对象封闭到目标

线程上

    - 对线程封闭对象的引用通常保存在公有变量中
  - 在可能的情况下,应该使用更强的线程封闭技术,例如栈封闭或者ThreadLocal
- 栈封闭
  - 只能通过局部变量才能访问对象
- ThreadLocal
  - 用于防止对可变的单实例变量或全局变量进行共享
  - 注意
    - ThreadLocal变量类似于全局变量,会降低代码的可重用性,在类之间引入隐含的耦合性

- 不变性
  - 不可变对象
    - 在创建之后其状态就不能被修改
    - 线程安全
    - 满足的条件
      - 对象创建以后其状态就不能修改
      - 对象所有的域都是final类型
      - 对象是正确创建的(创建期间this引用没有逸出)
  - final域
    - 能够确保初始化过程的安全性
    -

注意 原理 避免点 问题点