常见锁策略

- **悲观锁**
  - 含义
    - 认为锁冲突的概率比较高，如果不严格同步线程调用，那么会产生异常，所以一开始就会将会锁定资源，只供一个线程调用，而阻塞其他线程
- **乐观锁**
  - 含义
    - 认为数据一般不会产生并发冲突，所以在数据进行提交更新的时候，才会正式对数据是否产生并发冲突进行检测
  - 使用场景
    - 适用于多读的应用类型，这样可以提高吞吐量
  - 实现
    - 大部分使用版本号，也有使用 CAS 操作
  - CAS（compare and swap）
    - 工作过程
      - 参数
        - oldValue
          - 代表之前读到的资源对象的状态值
        - newValue
          - 代表想要将资源对象的状态值更新后的值
      - 给资源对象设置状态值
        1.线程获取资源对象时比较 oldValue 和 状态值
        2.若相同将状态值交换为 newValue，
        3.这样其他线程来获取资源对象时发现状态值 和 自己的 oldValue 不同，进行自旋（重复进行 CAS 操作，默认 10 次）
        - 存在的问题
          - 但是如果比较并交换的操作不是原子性的还会带来线程不安全的问题，所以核心问题是在 CAS 操作时必须只有一条线程执行操作
        - 如何实现 CAS 的原子性?
          - 这里总不能再进行加锁操作了，否则这不就是出现了鸡生蛋蛋生鸡的问题吗?
            CAS 的原子性是通过硬件支持的，各种不同架构的 CPU 都提供了指令级别的原子性操作，不需要通过操作系统的同步原语 如 mutex

令级别的原子性操作，不需要通过操作系统的同步原语 如 mutex，CPU 已经原生的支持了CAS

- ABA 问题
  - 举个实际的例子：A 先给 B 转账 100 元，旧值 100 预期 0，接着C 给 B 转账 100 元。
    但是 A 点了不小心点了两次转账，在第一次转账成功后，C 给 A 转账的线程获得了时间片，A 又变成了 100 元。A 给 B 转账的线程又获得了时间片，第二次转账也成功了。最后 A 剩 0 元，原本应该有 100 元。
  - 解决办法
    - AtomicReference -> AtomicStampedReference（增加版本号），假设初始版本号为 1，A 给 B 转账成功版本号加 1，第二次转账版本号还是 1，小于现在的版本号 2，操作失败
- 外部支持
- Java 层面
  - unsafe 类中的 cas 方法都使用 native 修饰，说明这些是一个本地方法，和具体的本地i实现有关
    - 如果 CPU 是X86 架构那么事实上这个本地方法将会调用系统的 Atomic::cmpxchg 原子指令

共享锁/非共享锁

- 共享锁
  - 一把锁可以被多个线程拥有
- 非共享锁
  - 一把锁只能被一个线程拥有
- 1.读写锁中的读锁是共享锁、写锁是非共享锁
- 2.读写和写锁是互斥的（防止同时读写，产生脏读）

公平锁/非公平锁

- 公平锁
  - 锁的获取顺序必须和线程方法先后顺序保持一致
  - 优点:执行有序，所以结果是可预期的
- 非公平锁
  - 锁的获取顺序和线程方法先后顺序无关（默认琐策略）
  - 优点：性能高

可重入锁

- 允许同一个线程多次获取同一把锁，不发生死锁

- 实现的方式是在锁中记录该锁持有的线程身份, 以及一个计数器(记录加锁次数). 如果发现当前加锁 的线程就是持有锁的线程, 则直接计数自增