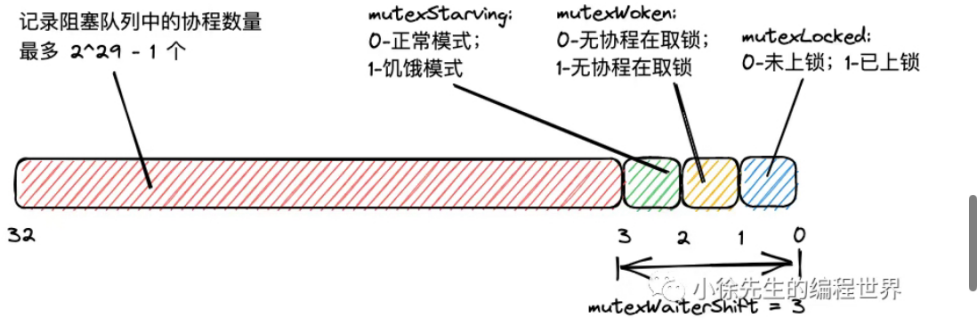
Golang随记（sync.Mutex）

- 下面两个锁都不需要显示的去初始化，无论是直接声明变量还是嵌入到结构体中，go会进行一个初始化。
- sync.Mutex为互斥锁
- sync.RWMutex为读写锁，逻辑上可以拆解为一把读锁和写锁，允许并发读，适合读多写少的场景，如果写场景很多会退化为互斥锁
  - RLock/RUnlock：读操作时调用的方法，加读锁 / 释放读锁；
  - Lock/Unlock：写操作时调用的方法，加写锁 / 释放写锁；
- 一把互斥锁的简单框架
  - 通过一个状态标识锁，比如1为lock，0为unlock
  - 一个go routine 尝试去上锁成功后，将状态位置1，解锁再置0
  - 如果获取锁失败（状态位为1），则需要等待解锁
  - 对状态位的修改需要去保证原子操作（通过cpu原语实现）以及double check（修改之前再确认状态位的值）
- 针对尝试获取锁时发现锁被获取了有两种方式
  - 阻塞/唤醒：阻塞挂起该go routine，等锁释放后通过回调函数通知。不会浪费cpu资源，但会切换上下文消耗，适合高并发场景（占用锁时间比较长）
  - cas+自旋，不停的去尝试获取锁。不需要进行上下文切换，但会浪费cpu资源。适合并发强度低的场景（占用锁时间比较短）
- sync.mutex结合了上述两种方式：
  - 先使用cas+自旋，出现下面3种情况会转为阻塞/唤醒
  - 自旋累计到4次未获取锁
  - cpu单核并且只有单个p调度器（如果自旋则会一直空转，锁得不到释放）
  - 当前p的执行队列有待执行的G（不影响gmp调度效率）
- 对于阻塞/唤醒模式，sync.mutex也有两种模式
  - 正常模式/非饥饿模式：当锁被释放时，会唤醒阻塞队列中的g（fifo，唤醒的是队头的g，阻塞队列内部是公平的），但这个g会与新来的g（请求锁但还没加入阻塞队列的g）进行竞争，并且处于劣势（新来的g状态本来就是runable，而阻塞的g则不是而是唤醒时转换；此外阻塞队列被唤醒的g只有一个，而新来的g根据并发场景的不同可能有多个）。长时间获取不到锁处于阻塞状态的g会陷入饥饿状态。
  - 饥饿模式：锁的所有权按阻塞队列的顺序来获取，新进入的g不得抢占，而是加入阻塞队列的队尾。
- 默认模式为正常模式，当阻塞队列中有g超过1ms没获得锁会转入饥饿模式
- 当阻塞队列为空或获取锁的g等待时间小于1ms则会转入正常模式
- go之所以选择切换这两种模式是性能与公平的不断平衡（正常模式性能更好，阻塞g是需要消耗的）
- mutex的数据结构
  - state（int32），不同的bit标识是否上锁，是否处于饥饿模式，阻塞队列中是否有g被唤醒，阻塞的g数量（最多2^29-1）
  - sema，阻塞和唤醒g的信号量

Mutex.state 字段为 int32 类型，不同 bit 位具有不同的标识含义：



Mutex.state字段

- 对state的一些位操作
  - state & mutexLocked：判断是否上锁
  - state | mutexLocked：加锁；
  - state & mutexWoken：判断是否存在抢锁的协程；
  - state | mutexWoken：更新状态，标识存在抢锁的协程；
  - state &^ mutexWoken：更新状态，标识不存在抢锁的协程；
  - state & mutexStarving：判断是否处于饥饿模式；
  - state | mutexStarving：置为饥饿模式；
  - state >> mutexWaiterShif：获取阻塞等待的协程数；
  - state += 1 << mutexWaiterShif：阻塞等待的协程数 + 1
- sync.mutex的lock的流程
  - 先通过cas操作，如果state为0（意味着阻塞队列无g，正常模式，没有协程在获取锁，未上锁），则直接获取锁。
  - 如果锁被占用，并且当前模式为正常模式，满足自旋条件则进入自旋。
  - 自旋出来后，要么加锁成功，要么阻塞挂起，这里也得考虑当前锁的模式。不过这里对sync.mutex中state的修改是类似乐观锁模式，先复制old值为new，对new进行修改，最后再通过cas操作进行更新。
- sync.mutex的unlock流程
  - 通过原子操作解锁，如果当前参与竞争锁的只有自身一个g，则直接返回
  - 解锁时会再次检查这把锁是否被占用，如果被没有被占用也就是尝试去释放一把没被加锁的锁，则会返回fatal
  - 饥饿模式下，直接唤醒队列头部的g（信号量）；正常模式下，如果没有其它活跃协程介入，则无需关心后续流程
- sync.rwmutex的结构
  - sync.mutex
  - writersem，阻塞等待写锁的g队列，信号量
  - readersem，阻塞等待读锁的g队列，信号量
  - readercount，标识当前等待或占用读锁的g数量
  - readerwait，标识当前占用读锁的g数量（针对写的g，去看还有多少个读的g去释放后才能去占用锁）

- 共享读锁的g数量上限值为2^29
- 读锁流程（对于结构体的几个变量都是原子操作）
  - 尝试去将readcount的数量加一，如果小于0，表示有写锁，则直接阻塞挂起
- 读锁解锁流程
  - 尝试去将readcount的数量减1，如果小于0并且自己是最后一个释放读锁的，尝试去唤醒写锁阻塞的g
- 写锁加锁流程
  - 对rwmutex内置的互斥锁进行加锁操作，使得只有一个写的g
  - 如果存在未释放读锁的g，则将当前尝试写的g添加到写锁的阻塞队列中挂起
- 写锁解锁流程
  - 唤醒读锁阻塞队列中的所有g（数量上更有优势）
- 读写交互过程
  - 如果当前没有阻塞的写锁的g，尝试读的g会直接获取读锁；如果当前有阻塞的写锁的g，则不能获取到读锁，加入阻塞队列
  - 最后一个释放读锁的g则会唤醒阻塞的写锁的g
  - 如果有多个写锁的g，只会有一个可能会获取写锁，而另外的尝试获取写锁的g放入到mutex的阻塞队列中