Go语言 互斥锁（Mutex）和读写互斥锁（RWMutex）

Go语言包中的 sync 包提供了两种锁类型：互斥锁sync.Mutex 和 读写锁sync.RWMutex。

互斥锁Mutex 是最简单的一种锁类型，同时也比较暴力，当一个 goroutine 获得了 Mutex 后，其他 goroutine 就只能乖乖等到这个 goroutine 释放该 Mutex。互斥锁是一种常用的控制共享资源访问的方法，它能够保证同时只有一个goroutine可以访问共享资源。

读写锁RWMutex 相对友好些，是经典的单写多读模型。在读锁占用的情况下，会阻止写，但不阻止读，也就是多个 goroutine 可同时获取读锁（调用 RLock() 方法；而写锁（调用 Lock() 方法）会阻止任何其他 goroutine（无论读和写）进来，整个锁相当于由该 goroutine 独占。从 RWMutex 的实现看，RWMutex 类型其实组合了 Mutex：

```
type RWMutex struct {
    w Mutex
    writerSem uint32
    readerSem uint32
    readerCount int32
    readerWait int32
}
```

读写锁分为两种：读锁和写锁。当一个goroutine获取读锁之后，其他的goroutine如果是获取读锁会继续获得锁，如果是获取写锁就会等待；当一个goroutine获取写锁之后，其他的goroutine无论是获取读锁还是写锁都会等待。

对于这两种锁类型，任何一个 Lock() 或 RLock() 均需要保证对应有 Unlock() 或 RUnlock() 调用与之对应，否则可能导致等待该锁的所有 goroutine 处于饥饿状态，甚至可能导致死锁。锁的典型使用模式如下：

```
package main

import (
    "fmt"
    "sync"
)

var (
    // 逻辑中使用的某个变量 这里无论是包级的变量还是结构体成员字段，都可以
    count int
    // 与变量对应的使用互斥锁 建议将互斥锁的粒度设置得越小越好，降低因为共享访问时等待的时间
    countGuard sync.Mutex
)

// 获取 count 值的函数封装，通过这个函数可以并发安全的访问变量 count
func GetCount() int {
    // 锁定 尝试对 countGuard 互斥量进行加锁。一旦 countGuard 发生加锁，如果另外一个
goroutine 尝试继续加锁时将会发生阻塞，直到这个 countGuard 被解锁
    countGuard.Lock()
    // 在函数退出时解除锁定 使用 defer 将 countGuard 的解锁进行延迟调用，解锁操作将会发生在
GetCount() 函数返回时
```

```
    defer countGuard.Unlock()
    return count
}

// 设置 count 值时，同样使用 countGuard 进行加锁、解锁操作，保证修改 count 值的过程是一个原子过程，不会发生并发访问冲突。
func SetCount(c int) {
    countGuard.Lock()
    count = c
    countGuard.Unlock()
}
func main() {
    // 可以进行并发安全的设置
    SetCount(1)
    // 可以进行并发安全的获取
    fmt.Println(GetCount())
}
#结果
1
```

在读多写少的环境中，可以优先使用读写互斥锁（sync.RWMutex），它比互斥锁更加高效。sync 包中的 RWMutex 提供了读写互斥锁的封装。

我们将互斥锁例子中的一部分代码修改为读写互斥锁，参见下面代码：

```
package main

import (
    "fmt"
    "sync"
)

var (
    // 逻辑中使用的某个变量 这里无论是包级的变量还是结构体成员字段，都可以
    count int
    // 与变量对应的使用读写互斥锁
    countGuard sync.RWMutex
)

// 获取 count 值的函数封装，通过这个函数可以并发安全的访问变量 count
func GetCount() int {
    // 锁定
    countGuard.RLock()
    // 在函数退出时解除锁定
    defer countGuard.RUnlock()
    return count
}

// 设置 count 值时，同样使用 countGuard 进行加锁、解锁操作，保证修改 count 值的过程是一个原子过程，不会发生并发访问冲突。
func SetCount(c int) {
    countGuard.Lock()
    count = c
    countGuard.Unlock()
}
```

```
func main() {
    // 可以进行并发安全的设置
    SetCount(1)
    // 可以进行并发安全的获取
    fmt.Println(GetCount())
}
```

声明 countGuard 时，从 sync.Mutex 互斥锁改为 sync.RWMutex 读写互斥锁。

获取 count 的过程是一个读取 count 数据的过程，适用于读写互斥锁。在这一行，把 countGuard.Lock() 换做 countGuard.RLock()，将读写互斥锁标记为读状态。如果此时另外一个 goroutine 并发访问了 countGuard，同时也调用了 countGuard.RLock() 时，并不会发生阻塞。

与读模式加锁对应的，使用读模式解锁。

> 需要注意的是读写锁非常适合读多写少的场景，如果读和写的操作差别不大，读写锁的优势就发挥不出来。