并发?

并发机制，尤其是多goroutine之间的共享变量

并发问题分析手段 & 基本模式

竞争条件

竞态条件？race condition!🥰
多个进程 同时访问/操纵 结果与特定顺序有关...🥰🥰

临界区问题，同步和互斥...🥰🥰🥰

我说婷婷，in圣经
数据竞争：数据竞争会在两个以上的goroutine并发访问相同的变量且至少其中一个为写操作时发生

避免

1. 不要写操作，例如初始化时就赋值完成，后续只读不写（需要update的就寄😊）
2. 避免从多个goroutine访问变量，一个main访问，剩下的都用channel传——《不要使用共享数据来通信；使用通信来共享数据》
   监控协程 monitor goroutine
   e.g.银行存款

```go
package bank

var deposits = make(chan int) // 用于发送存款请求
var balances = make(chan int) // 用于查询余额

func Deposit(amount int) { deposits <- amount } // 对外接口，存款
func Balance() int       { return <-balances } // 对外接口，查询余额

func teller() {
    var balance int // 实际余额变量
    for {
        select {
        case amount := <-deposits:
            balance += amount // 处理存款
        case balances <- balance: // 处理余额查询
        }
    }
}

func init() {
    go teller() // 这样的goroutine称为 monitor goroutine
}
```

数据流：

```
[调用Deposit()] --> [deposits通道] --> [teller协程更新balance]
[调用Balance()] <-- [balances通道] <-- [teller协程读取balance]
```

3. 真·互斥，同一个时刻最多只有一个goroutine在访问——临界区问题！

sync.Mutex互斥锁

该来的还是来了😊

channel cos mutex

一个容量为1的channel，只能0/1的信号量——二元信号量(binary semaphore)

```
var (
    sema = make(chan struct{}{}, 1) // cap1 channel or a binary semaphore
    balance int
)

func Deposit(amount int) {
    sema <- struct{}{} // wait(mutex)
    balance = balance + amount // 临界区
    <-sema // signal(mutex)
}

func Balance() int {
    sema <- struct{}{} // wait(mutex)
    b := balance // 临界区
    <-sema // signal(mutex)
    return b
}
```

mutex本体

sync.Mutex

```go
import "sync"

var (
    mu sync.Mutex // mutex
    balance int
)

func Deposit(amount int) {
    mu.Lock()
    balance = balance + amount
    mu.Unlock()
}

func Balance() int {
    mu.Lock()
    b := balance
    mu.Unlock()
    return b
}
```

总是忘记unlock？defer！

defer 延迟函数，最后再做
！ mutex.Unlock()的梦中情defer😘

```go
func Balance() int {
    mu.Lock()
    defer mu.Unlock()
    return balance
}
```

而且，就算程序出现panic异常，还是会defer的，还是会释放锁的！
这是真爱呀😘 😘 🥳 🥳

一个函数两次wait(mutex)?

```
// NOTE: incorrect!
func Withdraw(amount int) bool {
    mu.Lock() // Lock()
    defer mu.Unlock() // 最后再Unlock()
    Deposit(-amount) // 见上面，这里还有一次mu.Lock()
    // 死锁😋
    if Balance() < 0 { // 钱不够，不准取
        Deposit(amount)
        return false
    }
    return true
}
```

那怎么办？🤔

搞一个没有mutex的deposit()就是咯😋

```
func deposit(amount int) { balance += amount } // 默认已经在临界区内了，不需要再mutex了

func Withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()
    deposit(-amount)
    if balance < 0 {
        deposit(amount)
        return false
    }
    return true
}
```

sync.RWMutex读写锁

第一类Reader & Writer问题？！😘

在这种场景下我们需要一种特殊类型的锁，其允许多个只读操作并行执行，但写操作会完全互斥。这种锁叫作"多读单写"锁（multiple readers, single writer lock），Go语言提供的这样的锁是sync.RWMutex

```
var mu sync.RWMutex
var balance int
func Balance() int {
    mu.RLock() // readers lock
    defer mu.RUnlock()
    return balance
}
```

1. 任意数量读者可以同时获取锁
2. 读写互斥
3. 写者优先

？就多个RW就拿下我高贵的第一类Reader & Writer问题了？？ 🙃

实现机制

```
type RWMutex struct {
    w           Mutex  // 用于写锁的互斥锁
    writerSem   uint32 // 写者等待信号量
    readerSem   uint32 // 读者等待信号量
    readerCount int32  // 当前读者数量
    readerWait  int32  // 等待中的读者数量
}

// ... 后面省略，详见八个字母仙人
```

这下看懂了😋

我说为啥，原来是封装好了，纯轮椅嘛🥳

内存同步

也是说过的，看似简单的赋值语句，也可能不是"原子操作"，寄存器读取，修改，写入寄存器等，也可能会出现问题
So, 可能的话，将变量限定在goroutine内部；如果是多个goroutine都需要访问的变量，使用互斥条件来访问

sync.Once惰性初始化

`sync.Once` 用于确保某个操作只执行一次的并发安全机制，为惰性初始化（Lazy Initialization）而生🥳

惰性 / 懒惰 / 懒，不到万不得已就按表不发，实在不行才进行操作/修改等😋

惰性初始化：

1. 延迟初始化：真正需要时才进行初始化
2. 线程安全：在多 goroutine 环境下也能保证只初始化一次
3. 性能优化：避免不必要的初始化开销

```
func (o *Once) Do(f func()) // 只Do一次！
```

全局唯一性：
整个 sync.Once 实例范围内只执行一次，所有 goroutine 共享同一个执行结果

e.g.

```
var (
        instance *heavyObject
        once     sync.Once
)

type heavyObject struct {
        data string
}

func getInstance() *heavyObject {
        once.Do(func() { // 只有第一次调用时才会执行初始化函数
                fmt.Println("执行初始化")
                instance = &heavyObject{data: "昂贵的初始化数据"} // 初始化创造一个实例
        })
        return instance
}

func main() {
        var wg sync.WaitGroup
        wg.Add(3)

        // 并发获取实例
        go func() {
                defer wg.Done()
                fmt.Println("goroutine1:", getInstance().data)
        }()

        go func() {
                defer wg.Done()
                fmt.Println("goroutine2:", getInstance().data)
        }()

        go func() {
                defer wg.Done()
                fmt.Println("goroutine3:", getInstance().data)
        }()

        wg.Wait()
}
```

挺适合单例模式，例如数据库的单例模式，只get一个instance，也可以避免使用全局变量带来的问题

竞争条件检测

再小心还是会犯错😋

Luckily, 动态分析工具——竞争检查器 the race detector!

锵锵！ `-race`

```
go run -race main.go    # 运行并检测
go build -race          # 构建带检测的可执行文件
go test -race           # 测试时检测
```

竞争检测实例：

```
WARNING: DATA RACE
Read at 0x00c00001a0f0 by goroutine 7:
  main.increment()
      /path/to/file.go:15 +0x38

Previous write at 0x00c00001a0f0 by goroutine 6:
  main.increment()
      /path/to/file.go:15 +0x54

Goroutine 7 (running) created at:
  main.main()
      /path/to/file.go:20 +0x78

Goroutine 6 (finished) created at:
  main.main()
      /path/to/file.go:19 +0x5a
```

快说谢谢race哥😋

Goroutines vs 线程Thread

在go里就叫goroutine，出了go就叫thread😋
咳咳，，，

动态栈

OS线程，固定大小的内存块做栈（复习：线程的栈是独立的）(usually 2MB)

goroutine，一开始很小的栈(usually 2KB)，可以动态伸缩，最大有1GB

调度

OS线程调度：内核函数scheduler调度，线程切换需要完整的上下文切换

goroutine调度：我们Go有自己的调度器(wink*)~多对多模型(M:N)，M用户空间的goroutine对N内核级线程，goroutine调度完全在用户空间完成，不需要进入内核的上下文，成本更低

GOMAXPROCS

Go的调度器使用了一个叫做GOMAXPROCS的变量来决定会有多少个操作系统的线程同时执行Go的代码
默认：有多少就是多少（e.g.八核机器， `GOMAXPROCS = 8` ）

可以显式控制

```
for {
    go fmt.Print(0)
    fmt.Print(1)
}

$ GOMAXPROCS=1 go run hacker-cliché.go
111111111111111111100000000000000000000011111...
# 只有一个核，先跑一会1再跑一会0

$ GOMAXPROCS=2 go run hacker-cliché.go
010101010101010101011001100101011010010100110...
# 两个核，01一块跑
```

Goroutine没有ID号

P有PID，T有TID，但是Goroutine没有GID

有意而为之，避免依赖？（没太懂，所以请八字母仙人解释一下：）

Go 语言刻意不暴露 goroutine 的 ID，这是为了避免开发者滥用线程本地存储(TLS)模式，从而保持代码的显式性和可维护性

1. 反对隐式状态：传统 TLS 像"隐藏的全局变量"，函数行为受不可见线程状态影响；Go 要求所有依赖项必须通过显式参数传递，使数据流清晰可见
2. 设计哲学：保持并发代码的确定性，相同输入必然得到相同输出；避免"远距离幽灵行为"（action at a distance），即看似无关的代码因共享隐式状态而产生意外交互
3. 这种设计强制开发者采用更健康的并发模式，类似"依赖注入"思想——所有依赖必须显式声明，不能偷偷从全局状态获取

race condition大笨蛋，算了，race condition天天开心

——《云边有个mutex》

🥰🥰