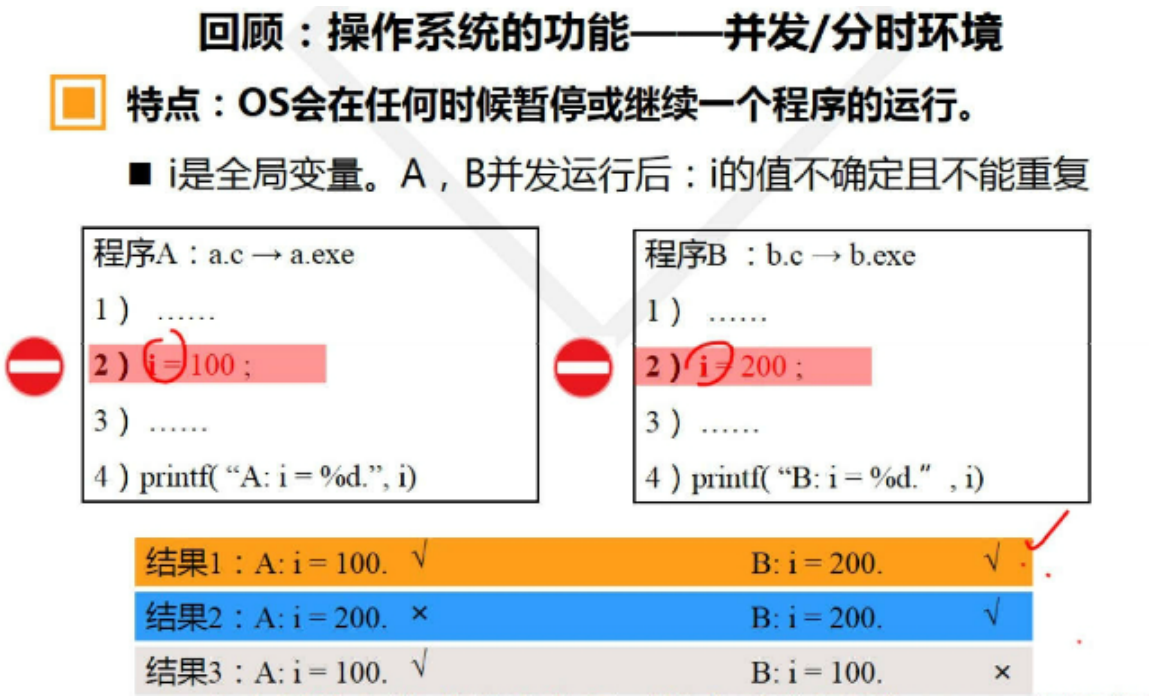
Go并发编程

讲课内容

- 基本同步原语
- 原子操作
- Channel
- 内存模型

基本同步原语



临界资源：一次只允许一个进程独占访问（使用的）资源

例：例子中的共享变量i

临界区：进程中访问临界资源的程序段

sync.Mutex

- 互斥锁
- 任何时间只允许一个**goroutine**在临界区运行
- **Unlock**未加锁的Mutex会**Panic**
- 公平
- 避免死锁
- 非重入锁

> go里没有重入锁，关于重入锁的概念，请参考java--也就是说没法对一个已经锁上的mutex来再次上锁--这会导致程序死锁，没法继续执行下去

```
mutex := &sync.Mutex{}

mutex.Lock()
// Update共享变量（比如切片，结构体指针等）
mutex.Unlock()
```

sync.RWMutex

- 可以被一堆的reader持有，或者被一个writer持有
- 适合大并发**read**的场景
- writer的Lock相对后续的reader的RLock优先级高
- 禁止递归读锁

```
mutex := &sync.RWMutex{}

mutex.Lock()
// Update 共享变量
mutex.Unlock()

mutex.RLock()
// Read 共享变量
mutex.RUnlock()
```

sync.Waitgroup

- 等待一组goroutine完成
- Add参数可以是负值；如果计数器小于0，**panic**
- 当计数器为0的时候，阻塞在Wait方法的goroutine都会被释放
- 注意事项：
  - Waitgroup - Add一定要在Wait之前设置好

```
func main() {
    var count int64
    var wg sync.WaitGroup
    for i := 0; i < 10000; i++ {
        wg.Add(1)
        go func() {
            atomic.AddInt64(&count, 1)
            wg.Done()
        }()
    }
    wg.Wait()
    fmt.Println(atomic.LoadInt64(&count))
}
```

✗

```
func main() {
    var count int64
    var wg sync.WaitGroup
    for i := 0; i < 10000; i++ {
        go func() {
            wg.Add(1)
            atomic.AddInt64(&count, 1)
            wg.Done()
        }()
    }
    wg.Wait()
    fmt.Println(atomic.LoadInt64(&count))
}
```

- Waitgroup - 多次Wait和多次Done

```
var count int64
var wg sync.WaitGroup
wg.Add(10)
for i := 0; i < 10; i++ {
  go func() {
    atomic.AddInt64(&count, 1)
    time.Sleep(2 * time.Second)
    wg.Done()
  }()
}
wg.Wait()
wg.Wait()

fmt.Println(atomic.LoadInt64(&count))
```

```
var count int64
var wg sync.WaitGroup
wg.Add(10)
for i := 0; i < 10; i++ {
  go func() {
    atomic.AddInt64(&count, 1)
    time.Sleep(2 * time.Second)
    wg.Done()
  }()
}
wg.Done() //多了一次Done
wg.Wait()

fmt.Println(atomic.LoadInt64(&count))
```

- Waitgroup - Add和Wait并发调用

```
for i := 0; i < 100; i++ {
  go func() {
    for {
      wg.Add(1)
      wg.Done()
    }
  }()
}
```

```
for i := 0; i < 100; i++ {
  go func() {
    for {
      wg.Wait()
    }
  }()
}
```

- Waitgroup - Wait未完成就Add

```
var wg sync.WaitGroup
wg.Add(1)
go func() {
  time.Sleep(time.Millisecond)
  wg.Done()
  wg.Add(1)
}()
wg.Wait()
```

sync.Once

- 只执行一次初始化

```
func main() {
   var once sync.Once
   var wg sync.WaitGroup
   onceBody := func() {
      fmt.Println("Only once")
   }
   for i := 0; i < 10; i++ {
      wg.Add(1)
      go func() {
         once.Do(onceBody)
         wg.Done()
      }()
```

```
    }
    wg.Wait()
}
```

输出：

```
Only once
```

sync.Pool

- 并发池，负责安全地保存一组对象
- `Get() interface{}` 用来从并发池中取出元素
- `Put(interface{})` 将一个对象加入并发池
- GC的执行可能会将Pool中对象全部移除

那么 Pool 都适用于什么场景呢？从它的特点来说，适用与无状态的对象的复用，而不适用与如连接池之类的。在 fmt 包中有一个很好的使用池的例子，它维护一个动态大小的临时输出缓冲区。

官方例子：

```
package main

import (
    "bytes"
    "io"
    "os"
    "sync"
    "time"
)

var bufPool = sync.Pool{
    New: func() interface{} {
        return new(bytes.Buffer)
    },
}

func timeNow() time.Time {
    return time.Unix(1136214245, 0)
}

func Log(w io.Writer, key, val string) {
    // 获取临时对象，没有的话会自动创建
    b := bufPool.Get().(*bytes.Buffer)
    b.Reset()
    b.WriteString(timeNow().UTC().Format(time.RFC3339))
    b.WriteByte(' ')
    b.WriteString(key)
    b.WriteByte('=')
    b.WriteString(val)
    w.Write(b.Bytes())
    // 将临时对象放回到 Pool 中
    bufPool.Put(b)
}
```

```
func main() {
    Log(os.Stdout, "path", "/search?q=flowers")
}

打印结果:
2006-01-02T15:04:05Z path=/search?q=flowers
```

sync.Map

- 使用`Store(interface {}, interface {})`添加元素。
- 使用`Load(interface {}) interface {}`检索元素。
- 使用`Delete(interface {})`删除元素。
- 使用`LoadOrStore(interface {}, interface {}) (interface {}, bool)`检索或添加之前不存在的元素。如果键之前在`map`中存在，则返回的布尔值为`true`。
- 使用`Range`遍历元素

```
m := &sync.Map{}

// 添加元素
m.Store(1, "one")
m.Store(2, "two")

// 获取元素1
value, contains := m.Load(1)
if contains {
  fmt.Printf("%s\n", value.(string))
}

// 返回已存value，否则把指定的键值存储到map中
value, loaded := m.LoadOrStore(3, "three")
if !loaded {
  fmt.Printf("%s\n", value.(string))
}

m.Delete(3)

// 迭代所有元素
m.Range(func(key, value interface{}) bool {
  fmt.Printf("%d: %s\n", key.(int), value.(string))
  return true
})
```

上面的程序会输出：

```
one
three
1: one
2: two
```

如你所见，`Range`方法接收一个类型为`func(key, value interface {})bool`的函数参数。如果函数返回了`false`，则停止迭代。有趣的事实是，即使我们在恒定时间后返回`false`，最坏情况下的时间复杂度仍为`O(n)`。

我们应该在什么时候使用`sync.Map`而不是在普通的`map`上使用`sync.Mutex`？

- 当我们对`map`有频繁的读取和不频繁的写入时。
- 当多个`goroutine`读取，写入和覆盖不相交的键时。具体是什么意思呢？例如，如果我们有一个分片实现，其中包含一组4个`goroutine`，每个`goroutine`负责25%的键（每个负责的键不冲突）。在这种情况下，`sync.Map`是首选。

原子操作

概念

原子操作，意思就是执行的过程不能背终端的操作。在针对某个值的原子操作执行过程中，cpu不会再去执行其他针对这个值得操作。在底层，**这会由CPU提供芯片级别的支持**，所以绝对有效。即使在拥有多CPU核心，或者多CPU的计算机系统中，原子操作的保证也是不可撼动的。

为什么选择原子操作

我们知道go语言在sync包中提供了锁的包，但是为什么我们还要使用atomic原子操作呢？总结下来有一下几个原因：

- 加锁的代价比较耗时，需要上下文切换。即使是在go语言的goroutine中也需要上下文的切换
- 只是针对基本类型，可以使用原子操作保证线程安全
- 原子操作在用户态可以完成，性能比互斥锁要高
- 针对特定需求，原子操作的步骤简单，不需要加锁-操作-解锁 这样的步骤

atomic 常见数据类型

- int32
- int64
- uint32
- uint64
- uintptr
- unsafe.Pointer

atomic常见操作有：

- 增减

```
var i64 uint64
//第一个参数必须是指针
atomic.AddUint64(&i64,5)
//在uint类型中可以使用^uint64(0)的方式打到减的效果
atomic.AddUint64(&i64, ^uint64(0))
fmt.Println(i64)
```

- 载入（保证读操作的原子性）

```
var i64 uint64
    i64 = 1
    //load 函数接收一个指针类型 返回指针指向的值
    num := atomic.LoadUint64(&i64)
    fmt.Println(num)
```

- 比较并交换

该操作在进行交换前首先确保变量的值未被更改，即仍然保持参数`old`所记录的值，满足此前提下才进行交换操作。CAS的做法类似操作数据库时常见的乐观锁机制。

需要注意的是，当有大量的goroutine 对变量进行读写操作时，可能导致CAS操作无法成功，这时可以利用for循环多次尝试。

```
var i64 uint64
    i64 = 5
    // cas接受3个参数，第一个为需要替换值得指针，第二个为旧值，第三个为新值
    // 当指针指向的值，跟你传递的旧值相等的情况下 指针指向的值会被替换
    ok := atomic.CompareAndSwapUint64(&i64,5, 50)
    fmt.Println(ok)
    // 当指针指向的值跟传递的旧值不相等，则返回false
    ok = atomic.CompareAndSwapUint64(&i64,40, 50)
    fmt.Println(ok)
```

- 交换

```
var i64 uint64
    i64 = 1
    //swap接受一个指针 一个值。函数会把值赋给指针 并返回旧值
    old := atomic.SwapUint64(&i64, 5)
    fmt.Println("old:",old,"new:",i64)
```

- 存储（保证写操作的原子性）

```
var i64 uint64
    i64 = 1
    //store 函数接受一个指针类型和一个值 函数将会把值赋到指针地址中
    atomic.StoreUint64(&i64, 5)
    fmt.Println(i64)
```

Channel

功能:

- 锁

```
type Mutex struct {
    ch chan struct{}
}

func NewMutex() *Mutex {
    mu := &Mutex{ch: make(chan struct{}, 1)}
    return mu
}

func (m *Mutex) Lock() {
    m.ch <- struct{}{}
}

func (m *Mutex) Unlock() {
    select {
    case <-m.ch:
    default:
```

```
            panic("unlock of unlocked mutex")
        }
    }

    func (m *Mutex) TryLock() bool {
        select {
        case m.ch <- struct{}{}:
            return true
        default:
        }
        return false
    }

    func (m *Mutex) IsLocked() bool {
        return len(m.ch) == 1
    }
```

- 信号(shutdown/close/finish)
- 数据交流(queue)

内存模型

内存模型是非常重要的，理解Go的内存模型就可以明白很多奇怪的竞态条件问题，"The Go Memory Model"的原文在这里，读个四五遍也不算多。

这里并不是要翻译这篇文章，英文原文是精确的，但读起来却很晦涩，尤其是happens-before的概念本身就是不好理解的，很容易跟时序问题混淆。大多数读者第一遍读Go的内存模型时基本上看不懂它在说什么。所以我要做的事情是用不怎么精确但相对通俗的语言解释一下。

先用一句话总结，Go的内存模型描述的是"在一个groutine中对变量进行读操作能够侦测到在其他goroutine中对该变量的写操作"的条件。

内存模型相关bug一例

为了证明这个重要性，先看一个例子。下面一小段代码：

```
package main

import (
  "sync"
  "time"
)

func main() {
  var wg sync.WaitGroup
  var count int
  var ch = make(chan bool, 1)
  for i := 0; i < 10; i++ {
    wg.Add(1)
    go func() {
      ch <- true
      count++
      time.Sleep(time.Millisecond)
      count--
      <-ch
```

```
      wg.Done()
    }()
  }
  wg.Wait()
}
```

以上代码有没有什么问题？这里把buffered channel作为semaphore来使用，表面上看最多允许一个goroutine对count进行++和--，但其实这里是有bug的。根据Go语言的内存模型，对count变量的访问并没有形成临界区。编译时开启竞态检测可以看到这段代码有问题：

```
go run -race test.go
```

编译器可以检测到16和18行是存在竞态条件的，也就是count并没像我们想要的那样在临界区执行。继续往下看，读完这一节，回头再来看就可以明白为什么这里有bug了。

happens-before

happens-before是一个术语，并不仅仅是Go语言才有的。简单的说，通常的定义如下：

假设A和B表示一个多线程的程序执行的两个操作。如果A happens-before B，那么A操作对内存的影响将对执行B的线程(且执行B之前)可见。

无论使用哪种编程语言，有一点是相同的：如果操作A和B在相同的线程中执行，并且A操作的声明在B之前，那么A happens-before B。

```
int A, B;
void foo()
{
  // This store to A ...
  A = 5;
  // ... effectively becomes visible before the following loads. Duh!
  B = A * A;
}
```

还有一点是，在每门语言中，无论你使用那种方式获得，happens-before关系都是可传递的：如果A happens-before B，同时B happens-before C，那么A happens-before C。当这些关系发生在不同的线程中，传递性将变得非常有用。

刚接触这个术语的人总是容易误解，这里必须澄清的是，happens-before并不是指时序关系，并不是说A happens-before B就表示操作A在操作B之前发生。它就是一个术语，就像光年不是时间单位一样。具体地说：

1. A happens-before B并不意味着A在B之前发生。
2. A在B之前发生并不意味着A happens-before B。

这两个陈述看似矛盾，其实并不是。如果你觉得很困惑，可以多读几篇它的定义。后面我会试着解释这点。记住，happens-before 是一系列语言规范中定义的操作间的关系。它和时间的概念独立。这和我们通常说”A在B之前发生”时表达的真实世界中事件的时间顺序不同。

A happens-before B并不意味着A在B之前发生

这里有个例子，其中的操作具有happens-before关系，但是实际上并不一定是按照那个顺序发生的。下面的代码执行了(1)对A的赋值，紧接着是(2)对B的赋值。

```
int A = 0;
int B = 0;
void main()
{
    A = B + 1; // (1)
    B = 1; // (2)
}
```

根据前面说明的规则，(1) happens-before (2)。但是，如果我们使用gcc -O2编译这个代码，编译器将产生一些指令重排序。有可能执行顺序是这样子的：

```
将B的值取到寄存器
将B赋值为1
将寄存器值加1后赋值给A
```

也就是到第二条机器指令(对B的赋值)完成时，对A的赋值还没有完成。换句话说，(1)并没有在(2)之前发生!

那么，这里违反了happens-before关系了吗？让我们来分析下，根据定义，操作(1)对内存的影响必须在操作(2)执行之前对其可见。换句话说，对A的赋值必须有机会对B的赋值有影响.

但是在这个例子中，对A的赋值其实并没有对B的赋值有影响。即便(1)的影响真的可见，(2)的行为还是一样。所以，这并不能算是违背happens-before规则。

A在B之前发生并不意味着A happens-before B

下面这个例子中，所有的操作按照指定的顺序发生，但是并能不构成happens-before 关系。假设一个线程调用pulishMessage，同时，另一个线程调用consumeMessage。 由于我们并行的操作共享变量，为了简单，我们假设所有对int类型的变量的操作都是原子的。

```
int isReady = 0;
int answer = 0;
void publishMessage()
{
  answer = 42; // (1)
  isReady = 1; // (2)
}
void consumeMessage()
{
  if (isReady) // (3) <-- Let's suppose this line reads 1
  printf("%d\n", answer); // (4)
}
```

根据程序的顺序，在(1)和(2)之间存在happens-before 关系，同时在(3)和(4)之间也存在happens-before关系。

除此之外，我们假设在运行时，isReady读到1(是由另一个线程在(2)中赋的值)。在这中情形下，我们可知(2)一定在(3)之前发生。但是这并不意味着在(2)和(3)之间存在happens-before 关系!

happens-before 关系只在语言标准中定义的地方存在，这里并没有相关的规则说明(2)和(3)之间存在happens-before关系，即便(3)读到了(2)赋的值。

还有，由于(2)和(3)之间，(1)和(4)之间都不存在happens-before关系，那么(1)和(4)的内存交互也可能被重排序 (要不然来自编译器的指令重排序，要不然来自处理器自身的内存重排序)。那样的话，即使(3)读到1，(4)也会打印出“0“。

Go关于同步的规则

我们回过头来再看看"The Go Memory Model"中关于happens-before的部分。

如果满足下面条件，对变量v的读操作r可以侦测到对变量v的写操作w：

1. r does not happen before w.
2. There is no other write w to v that happens after w but before r.

为了保证对变量v的读操作r可以侦测到某个对v的写操作w，必须确保w是r可以侦测到的唯一的写操作。也就是说当满足下面条件时可以保证读操作r能侦测到写操作w：

1. w happens-before r.
2. Any other write to the shared variable v either happens-before w or after r.

关于channel的happens-before在Go的内存模型中提到了三种情况：

1. 对一个channel的发送操作 happens-before 相应channel的接收操作完成
2. 关闭一个channel happens-before 从该Channel接收到最后的返回值0
3. 不带缓冲的channel的接收操作 happens-before 相应channel的发送操作完成

先看一个简单的例子：

```
var c = make(chan int, 10)
var a string
func f() {
    a = "hello, world"  // (1)
    c <- 0  // (2)
}
func main() {
    go f()
    <-c   // (3)
    print(a)  // (4)
}
```

上述代码可以确保输出"hello, world"，因为(1) happens-before (2)，(4) happens-after (3)，再根据上面的第一条规则(2)是 happens-before (3)的，最后根据happens-before的可传递性，于是有(1) happens-before (4)，也就是a = "hello, world" happens-before print(a)。

再看另一个例子：

```
var c = make(chan int)
var a string
func f() {
    a = "hello, world"  // (1)
    <-c   // (2)
}
func main() {
    go f()
    c <- 0  // (3)
    print(a)  // (4)
}
```

根据上面的第三条规则(2) happens-before (3)，最终可以保证(1) happens-before (4)。

如果我把上面的代码稍微改一点点，将c变为一个带缓存的channel，则print(a)打印的结果不能够保证是"hello world"。

```
var c = make(chan int, 1)
var a string
func f() {
    a = "hello, world"  // (1)
    <-c   // (2)
}
func main() {
    go f()
    c <- 0  // (3)
    print(a)  // (4)
}
```

因为这里不再有任何同步保证，使得(2) happens-before (3)。可以回头分析一下本节最前面的例子，也是没有保证happens-before条件。

参考链接

https://tiancaiamao.gitbooks.io/go-internals/content/zh/10.1.html

https://zhuanlan.zhihu.com/p/31122953

学习链接

https://dave.cheney.net/paste/concurrency-made-easy.pdf