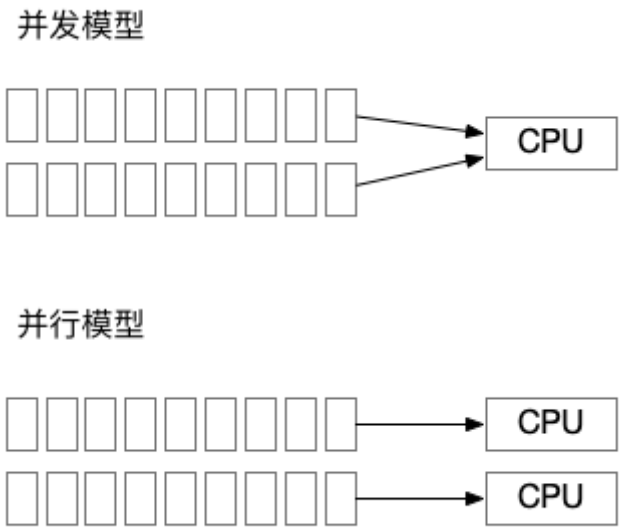
练识课堂 -- 资深Go语言工程师实践课程

第十章 并发编程

- 并发与并行
  - 并发 Concurrency 逻辑上具备同时处理多个任务的能力
  - 并行 Parallelism 物理上在同一时刻执行多个并发任务



- 并发Goroutine
  - Go 的并发通过 Goroutine 实现，Goroutine 不等同于协程Coroutine，而是两级线程模型，也称混合型线程模型
  - 主流线程模型
    - 内核级线程模型
    - 用户级线程模型 Coroutine
    - 两级线程模型 Goroutine
  - 线程默认的栈是 MB 级别， Goroutine 默认 2K
  - 使用Goroutine

```
go func() {
  fmt.Println("Goroutine")
}()
```

- 并发Channel

  接受方向、数据类型、Buffer 大小

```
ch1 := make(chan int) // 双向的 channel 可以用来收发，类型是 int
//单向的 channel 主要用于函数传递的安全因素，类似于 const 一般不直接声明
ch2 := make(chan<- int) // 接受的 channel 可以用来收发，类型是 int
ch3 := make(<-chan int) // 发送的 channel 可以用来收发，类型是 int
ch4 := make(chan int, 100) // 双向的 channel 可以用来收发，类型是 int，大小
100
```

- Channel的使用
  - ch := make(chan int) // 双向的 channel ，类型是 int
  - ch <- 1 // 写入数据
  - a := <-ch // 读取数据
  - 通道不再使用的时候可以调用 close 函数进行关闭
  - close(ch) // 关闭事件会通知到所有等在在 ch 上的 goroutine， 不能关闭仅是接收方的 ch

```
for i := range ch { // close 自动退出
  fmt.Println(i)
}
```

- 配合 select 使用
  - select 用法类似与 IO 多路复用，可以同时监听多个 channel 的消息状态

```
select {
    case <- ch1:
    ...
    case <- ch2:
    ...
    case ch3 <- 10;
    ...
    default:
    ...
}
```

  - select 可以同时监听多个 channel 的写入或读取
  - 执行 select 时，若只有一个 case 通过(不阻塞)，则执行这个 case 块
  - 若有多个 case 通过，则随机挑选一个 case 执行
  - 若所有 case 均阻塞，且定义了 default 模块，则执行 default 模块。若未定义 default 模块，则 select 语句阻塞，直到有 case 被唤醒。
  - 使用 break 会跳出 select 块。
- Channel小结
  - 向已经 close 的 channel 写入数据会导致 panic
  - 重复关闭 channel 或者 关闭为 nil 的 channel 也会引发 panic
  - 如果 channel 为 nil，收发都会阻塞
  - 同时监听多个 channel 或者 避免阻塞在 channel 上，可以配合 select 使用，select 在多个 channel 都满足条件的情况下，随机选择
  - 一般原则是写入方负责关闭 channel，读取方负责检测 channel 是否关闭
- 并发的同步模型
  - Mutex

```
// A Mutex is a mutual exclusion lock.
// The zero value for a Mutex is an unlocked mutex.
//
// A Mutex must not be copied after first use.
type Mutex struct {
  state int32
  sema  uint32
}
```

```
// A Locker represents an object that can be locked and unlocked.
type Locker interface {
  Lock()
  Unlock()
}
```

```
  var mutex sync.Mutex
  mutex.Lock()
  defer mutex.Unlock()
```

- 开箱即用，不用初始化，将并行操作修改为串行操作
- Mutex不是重入锁，避免多次锁定
- 推荐使用defer进行解锁
- 不要解锁未锁定的锁panic
- 不要重复解锁panic
- 不要在多个函数间直接传递，值copy以后，即是不同的互斥锁
- 结构体方法定义的接受对象如果包含了Mutex，则使用指针值接受对象

- sync.RWMutex 读写锁
  - 读读可以并行，读写不能并行
- sync.WaitGroup
  - 适合于一对多的 goroutine 的协同等待
  - 三个方法:Add Done Wait
  - 底层计数器机制实现，默认值为 0

```
func main() {
  var mutex sync.Mutex
  counter := 0
  wg := sync.WaitGroup{}
  for i := 0; i < 100; i++ {
    wg.Add(1)
    go func() {
      mutex.Lock()
      defer mutex.Unlock()
      defer wg.Done()
      counter++
    }()
  }
  wg.Wait()
  fmt.Println(counter)
}
```

第十一章 sync包

sync包有以下几个内容：

- sync.Pool 临时对象池
- sync.Mutex 互斥锁
- sync.RWMutex 读写互斥锁
- sync.WaitGroup 组等待
- sync.Cond 条件等待
- sync.Once 单次执行
- sync.Map 并发安全字典结构

1.1 临时对象池 (sync.Pool)

- Pool是用于存储那些被分配了但是没有被使用，而未来可能会使用的值，以减小垃圾回收的压力。
- Pool是协程安全的，应该用于管理协程共享的变量，不推荐用于非协程间的对象管理。
- 调动 New 函数，将使用函数创建一个新对象返回。
- 从Pool中取出对象时，如果Pool中没有对象，将执行New()，如果没有对New进行赋值，则返回Nil。
- 先进后出存储原则，和栈类似Pool一个比较好的例子是fmt包，fmt包总是需要使用一些[]byte之类的对象，Go建立了一个临时对象池，存放着这些对象，如果需要使用一个[]byte，就去Pool里面拿，如果拿不到就分配一份。这比起不停生成新的 []byte，用完了再等待gc回收来要高效得多。

示例代码：

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var pool sync.Pool
    var val interface{}
    pool.Put("1")
    pool.Put(12)
    pool.Put(true)

    for {
        val = pool.Get()
        if val == nil {
            break
        }
        fmt.Println(val)
    }
}
```

利用goroutine和channel进行go的并发模式，实现一个资源池实例，资源池可以存储一定数量的资源，用户程序从资源池获取资源进行使用，使用完成将资源释放回资源池

示例代码

```
//一个安全的资源池，被管理的资源必须都实现io.Close接口
type Pool struct{
    m sync.Mutex
    res chan io.Closer
    factory func () (io.Closer, error)
    closed bool
}
```

这个结构体Pool有四个字段

- m 是一个互斥锁，这主要是用来保证在多个goroutine访问资源时，池内的值是安全的。
- res 字段是一个有缓冲的通道，用来保存共享的资源，这个通道的大小，在初始化Pool的时候就指定的。注意这个通道的类型是io.Closer接口，所以实现了这个io.Closer接口的类型都

可以作为资源，交给我们的资源池管理。

- factory 这个是一个函数类型，它的作用就是当需要一个新的资源时，可以通过这个函数创建，也就是说它是生成新资源的，至于如何生成、生成什么资源，是由使用者决定的，所以这也是这个资源池灵活的设计的地方。
- closed字 段表示资源池是否被关闭，如果被关闭的话，再访问是会有错误的。

pool.go

```
package main

import (
    "errors"
    "fmt"
    "io"
    "sync"
    "time"
)

type Pool struct {
    m   sync.Mutex
    res chan io.Closer
    //创建资源的方法，由用户程序自己生成传入
    factory func() (io.Closer, error)
    closed  bool
    //资源池获取资源超时时间
    timeout <-chan time.Time
}

//资源池关闭标志
var ErrPoolClosed = errors.New("资源池已经关闭")

//超时标志
var ErrTimeout = errors.New("获取资源超时")

//新建资源池
func New(fn func() (io.Closer, error), size int) (*Pool, error) {
    if size <= 0 {
        return nil, errors.New("新建资源池大小太小")
    }
    //新建资源池
    p := Pool{
        factory:  fn,
        res: make(chan io.Closer, size),
    }
    //向资源池循环添加资源，直到池满
    for count := 1; count <= cap(p.res); count++ {
        r, err := fn()
        if err != nil {
            fmt.Println("添加资源失败，创建资源方法返回nil")
            break
        }
        fmt.Println("资源加入资源池")
        p.res <- r
    }
    fmt.Println("资源池已满，返回资源池")
    return &p, nil
}
```

```
//获取资源
func (p *Pool) Acquire(d time.Duration) (io.Closer, error) {
    //设置d时间后超时
    p.timeout = time.After(d)
    select {
    case r, ok := <-p.res:
        fmt.Println("获取", "共享资源")
        if !ok {
            return nil, ErrPoolClosed
        }
        return r, nil
    case <-p.timeout:
        return nil, ErrTimeout
    }
}

//放回资源池
func (p *Pool) Release(r io.Closer) {
    //上互斥锁，和Close方法对应，不同时操作
    p.m.Lock()
    defer p.m.Unlock()

    if p.closed {
        r.Close()
        return
    }
    //资源放回队列
    select {
    case p.res <- r:
        fmt.Println("资源放回队列")
    default:
        fmt.Println("资源队列已满，释放资源")
        r.Close()
    }
}

//关闭资源池
func (p *Pool) Close() {
    //互斥锁，保证同步，和Release方法相关，用同一把锁
    p.m.Lock()
    defer p.m.Unlock()

    if p.closed {
        return
    }
    p.closed = true
    //清空通道资源之前，将通道关闭，否则引起死锁
    close(p.res)
    for r := range p.res {
        r.Close()
    }
}
```

main.go

```
package main
```

```
import (
    "fmt"
    "io"
    "math/rand"
    "sync"
    "sync/atomic"
    "time"
)

const (
    maxGoroutines   = 25
    pooledResources = 2
)

//实现接口类型 资源类型
type dbConnection struct {
    ID int32
}

//实现接口方法
func (conn *dbConnection) Close() error {
    fmt.Printf("资源关闭,ID:%d\n", conn.ID)
    return nil
}

//给每个连接资源给id
var idCounter int32

//创建新资源
func createConnection() (io.Closer, error) {
    id := atomic.AddInt32(&idCounter, 1)
    fmt.Printf("创建新资源,id:%d\n", id)
    return &dbConnection{ID: id}, nil
}

//测试资源池
func performQueries(query int, p *Pool) {
    conn, err := p.Acquire(10 * time.Second)
    if err != nil {
        fmt.Println("获取资源超时")
        fmt.Println(err)
        return
    }
    //方法结束后将资源放进资源池
    defer p.Release(conn)
    //模拟使用资源
    time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond)
    fmt.Printf("查询goroutine id:%d,资源ID: %d\n", query, conn.
(*dbConnection).ID)
}

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

    p, err := New(createConnection, pooledResources)
    if err != nil {
```

```
        fmt.Println(err)
    }

    //每个goroutine一个查询，每个查询从资源池中获取资源
    for query := 0; query < maxGoroutines; query++ {
        go func(q int) {
            performQueries(q, p)
            wg.Done()
        }(query)
    }

    //主线程等待
    wg.Wait()
    fmt.Println("程序结束")
    //释放资源
    p.Close()
}
```

1.2 互斥锁 (sync.Mutex)

- 互斥锁用来保证在任一时刻，只能有一个协程访问某对象。
- Mutex的初始值为解锁状态，Mutex通常作为其它结构体的匿名字段使用，使该结构体具有Lock和Unlock方法。
- Mutex可以安全的再多个协程中并行使用。
- 如果对未加锁的进行解锁，则会引发panic。

加锁，保证数据能够正确：

```
package main

import (
    "fmt"
    "sync"
)

var num int
var mu sync.Mutex
var wg sync.WaitGroup

func main() {
    wg.Add(10000)
    for i := 0; i < 10000; i++ {
        go Add()
    }
    wg.Wait()
    fmt.Println(num)
}

func Add() {
    mu.Lock() // 加锁
    defer func() {
        mu.Unlock() // 解锁
        wg.Done()
    }()

    num++
}
```

并发没有加锁时，会发现结果不等于10000，其原因是因为出现这样的情况，就是其中一些协程刚好读取了num的值，此时该协程刚好时间片结束，被挂起，没有完成加1。

然后其他协程进行加1，然后当调度回到原来那个协程，num = 原来的那个num(而不是最新的num) + 1，导致num数据出错。

1.3 读写锁(sync.RWMutex)

- RWMutex比Mutex多了一个“写锁定”和“读锁定”，可以让多个协程同时读取某对象。
- RWMutex的初始值为解锁状态。RWMutex通常作为其它结构体的匿名字段使用。
- RWMutex可以安全的在多个协程中并行使用。

```
// Lock 将 rw 设置为写状态，禁止其他协程读取或写入
func (rw *RWMutex) Lock()

// Unlock 解除 rw 的写锁定状态，如果rw未被锁定，则该操作会引发 panic。
func (rw *RWMutex) Unlock()

// RLock 将 rw 设置为锁定状态，禁止其他协程写入，但可以读取。
func (rw *RWMutex) RLock()

// Runlock 解除 rw 设置为读锁定状态，如果rw未被锁定，则该操作会引发 panic。
func (rw *RWMutex) RUnLock()

// RLocker 返回一个互斥锁，将 rw.RLock 和 rw.RUnlock 封装成一个 Locker 接口。
func (rw *RWMutex) RLocker() Locker
```

1.4 等待组(sync.WaitGroup)

- WaitGroup 用于等待一组协程的结束。
- 主协程创建每个子协程的时候先调用Add增加等待计数，每个子协程在结束时调用Done减少协程计数。
- 主协程通过 Wait 方法开始等待，直到计数器归零才继续执行。

```
// 计数器增加 delta，delte可以时负数
func (wg *WaitGroup) Add(delta int)

// 计数器减少1，等价于Add(-1)
func (wg *WaitGroup) Done()

// 等待直到计数器归零。如果计数器小于0，则该操作会引发 panic。
func (wg *WaitGroup) Wait()
```

1.5 条件等待(sync.Cond)

条件等待和互斥锁有不同，互斥锁是不同协程公用一个锁，条件等待是不同协程各用一个锁，但是wait()方法调用会等待（阻塞），直到有信号发过来，不同协程是共用信号。

示例代码：

```
package main

import (
    "fmt"
```

```
    "sync"
    "time"
)

func main() {
    var wg sync.WaitGroup
    cond := sync.NewCond(new(sync.Mutex))

    for i := 0; i < 3; i++ {
        go func(i int) {
            fmt.Println("协程", i, "启动。。。")
            wg.Add(1)
            defer wg.Done()
            cond.L.Lock()
            fmt.Println("协程", i, "加锁。。。")
            cond.Wait()
            fmt.Println("协程", i, "解锁。。。")
            cond.L.Unlock()
        }(i)
    }
    time.Sleep(time.Second)
    fmt.Println("主协程发送信号量。。。")
    cond.Signal()

    time.Sleep(time.Second)
    fmt.Println("主协程发送信号量。。。")
    cond.Signal()

    time.Sleep(time.Second)
    fmt.Println("主协程发送信号量。。。")
    cond.Signal()
    wg.Wait()
}
```

1.6 单次执行(sync.Once)

- Once的作用是多次调用但只执行一次，Once只有一个方法，Once.Do()，向Do传入一个函数，这个函数在第一次执行Once.Do()的时候会被调用
- 以后再执行Once.Do()将没有任何动作，即使传入了其他的函数，也不会被执行，如果要执行其它函数，需要重新创建一个Once对象。
- Once可以安全的再多个协程中并行使用，是协程安全的。

```
// 多次调用仅执行一次指定的函数f
func (o *Once) Do(f func())
```

示例代码：

```
// 示例：Once
package main

import (
    "fmt"
    "sync"
)

func main() {
```

```
    var once sync.Once
    var wg sync.WaitGroup

    onceFunc := func() {
        fmt.Println("法师爱你们哟~")
    }
    wg.Add(10)
    for i := 0; i < 10; i++ {
        go func() {
            defer wg.Done()
            once.Do(onceFunc) // 多次调用只执行一次
        }()
    }
    wg.Wait()
}
```

1.7 并发安全Map(sync.Map)

Go语言中的 map 在并发情况下，只读是线程安全的，同时读写是线程不安全的。

需要并发读写时，一般的做法是加锁，但这样性能并不高，Go语言在 1.9 版本中提供了一种效率较高的并发安全的 sync.Map，sync.Map 和 map 不同，不是以语言原生形态提供，而是在 sync 包下的特殊结构。

sync.Map 有以下特性：

- 无须初始化，直接声明即可。
- sync.Map 不能使用 map 的方式进行取值和设置等操作，而是使用 sync.Map 的方法进行调用，Store 表示存储，Load 表示获取，Delete 表示删除。
- 使用 Range 配合一个回调函数进行遍历操作，通过回调函数返回内部遍历出来的值，Range 参数中回调函数的返回值在需要继续迭代遍历时，返回 true，终止迭代遍历时，返回 false。

示例代码：

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var scene sync.Map
    // 将键值对保存到sync.Map
    scene.Store("法师", 97)
    scene.Store("老郑", 100)
    scene.Store("兵哥", 200)
    // 从sync.Map中根据键取值
    fmt.Println(scene.Load("法师"))
    // 根据键删除对应的键值对
    scene.Delete("法师")
    // 遍历所有sync.Map中的键值对
    scene.Range(func(k, v interface{}) bool {
        fmt.Println("iterate:", k, v)
        return true
    })
```

```
}
```

第十二章 CGO混合编程

> Go 有强烈的 C 背景，除了语法具有继承性外，其设计者以及其设计目标都与C语言有着千丝万缕的联系。
>
> 在Go与C语言互操作方面，Go更是提供了强大的支持。尤其是在Go中使用C，甚至可以直接在Go源文件中编写C代码，这是其他语言所无法望其项背的。
>
> 在如下一些场景中，可能会涉及到Go与C的互操作：
>
> - 提升局部代码性能时，用C替换一些Go代码。C之于Go，好比汇编之于C。
> - 嫌Go内存GC性能不足，自己手动管理应用内存。
> - 实现一些库的Go Wrapper。比如Oracle提供的C版本OCI，但Oracle并未提供Go版本的以及连接DB的协议细节，因此只能通过包装C版本OCI版本的方式以提供Go开发者使用。
>
> **前置条件**
>
> 要使用CGO特性，需要安装C / C构建工具链。
>
> 在macOS和Linux下安装GCC，使用命令安装：yum install gcc
>
> 在windows下是需要安装MinGW工具 http://www.mingw-w64.org/doku.php。
>
> 同时需要保证环境变量CGO_ENABLED被设置为1，这表示CGO是被启用的状态。在本地构建时CGO_ENABLED默认是启用的，当交叉构建时CGO默认是禁止的。
>
> 然后通过 **import "C"** 语句启用CGO特性。

1.1 import "C"

> 如果在Go代码中出现了 **import "C"**语句则表示使用了CGO特性，紧跟在这行语句前面的注释是一种特殊语法，里面包含的是正常的C语言代码。
>
> 当确保CGO启用的情况下，还可以在当前目录中包含C/C++对应的源文件。

```go
package main

//以块注释的形式写C语言代码

/*
#include <stdio.h>
//C语言函数格式
void Print()
{
    printf("法师好帅~");
}
*/
import "C"

func main() {
    C.Print()
}
```

> C语言和Go语言混合编程需要安装GCC，同时编译速度相对会变慢。

1.2 #cgo语句

可以通过**#cgo**语句 设置编译阶段和链接阶段的相关参数。

编译阶段的参数主要用于定义相关宏和指定头文件检索路径。

链接阶段的参数主要是指定库文件检索路径和要链接的库文件。

```
// #cgo CFLAGS: -D PORT = 8080 -I./include
// #cgo LDFLAGS: -L/usr/local/lib -llibevent
// #include <libevent.h>
import "C"
```

CFLAGS部分，`-D` 部分定义了宏 PORT，值为8080；`-I` 定义了头文件包含的检索目录。

LDFLAGS部分，`-L` 指定了链接时库文件检索目录，`-l` 指定了链接时需要链接libevent库。

1.3 类型转换

在Go语言中访问C语言的符号时，一般是通过虚拟的“C”包访问，比如**C.int**对应C语言的**int**类型。

Go语言中数值类型和C语言数据类型基本上是相似的，以下是对应关系表。

C语言类型	CGO类型	Go语言类型
char	C.char	byte
singed char	C.schar	int8
unsigned char	C.uchar	uint8
short	C.short	int16
unsigned short	C.short	uint16
int	C.int	int32
unsigned int	C.uint	uint32
long	C.long	int32
unsigned long	C.ulong	uint32
long long int	C.longlong	int64
unsigned long long int	C.ulonglong	uint64
float	C.float	float32
double	C.double	float64
size_t	C.size_t	uint

如果Go语言在调用C语言函数时，需要传递参数或者接受返回值，都需要类型转换。

```
package main

//以块注释的形式写C语言代码

/*
#include <stdio.h>
```

```
//C语言函数格式
int add(int a, int b)
{
return a + b;
}
*/
import "C"
import "fmt"

func main() {
    a := 10
    b := 20
    c := C.add(C.int(a), C.int(b))

    fmt.Println(c)
    fmt.Printf("%T\n", c)
    //转成Go语言整型
    fmt.Println(int(c))
}
```

结果

```
30
main._Ctype_int
30
```

第十三章 JSON操作

1 JSON介绍

JSON（JavaScript Object Notation）是一种比XML更轻量级的数据交换格式，在易于人们阅读和编写的同时，也易于程序解析和生成。尽管JSON是JavaScript的一个子集，但JSON采用完全独立于编程语言的文本格式，且表现为键/值对集合的文本描述形式（类似一些编程语言中的字典结构），这使它成为较为理想的、跨平台、跨语言的数据交换语言。

```
{
"Company":"Lianshi",
"Subjects":[
  "Go",
  "Web",
  "Python",
  "C++"
],
"IsOk":true,
"Price":8980.88
}
```

开发者可以用 JSON 传输简单的字符串、数字、布尔值，也可以传输一个数组，或者一个更复杂的复合结构。在 Web 开发领域中， JSON被广泛应用于 Web 服务端程序和客户端之间的数据通信。

2 JSON编码

Go语言内建对JSON的支持。使用Go语言内置的encoding/json 标准库，开发者可以轻松使用Go程序生成和解析JSON格式的数据。

使用json.Marshal()函数可以对一组数据进行JSON格式的编码。

json.Marshal()函数的声明如下：

```
func Marshal(v interface{}) ([]byte, error)
```

格式化输出：

```
// MarshalIndent 很像 Marshal，只是用缩进对输出进行格式化
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

2、1 通过结构体生成JSON

示例代码:

```
package main

import (
    "encoding/json"
    "fmt"
)

type IT struct {
    Company  string
    Subjects []string
    IsOk     bool
    Price    float64
}

func main() {
    t1 := IT{"Lianshi", []string{"Go", "Web", "Python", "C++"}, true,
8980.88}

    //生成一段JSON格式的文本
    //如果编码成功， err 将赋于零值 nil，变量b 将会是一个进行JSON格式化之后的[]byte类
型
    //b, err := json.Marshal(t1)

    b, err := json.MarshalIndent(t1, "", "    ")

    if err != nil {
        fmt.Println("json err:", err)
    }
    fmt.Println(string(b))
}
```

2、2 通过map生成JSON

示例代码：

```
// 创建一个保存键值对的映射
t1 := make(map[string]interface{})
t1["company"] = "Lianshi"
t1["subjects "] = []string{"Go", "Web", "Python", "C++"}
t1["isok"] = true
t1["price"] = 8980.88

b, err := json.Marshal(t1)
//json.MarshalIndent(t1, "", "    ")
if err != nil {
fmt.Println("json err:", err)
}
fmt.Println(string(b))
```

3 JSON解码

> 可以使用json.Unmarshal()函数将JSON格式的文本解码为Go里面预期的数据结构。
>
> json.Unmarshal()函数的原型如下：
>
> ```
> func Unmarshal(data []byte, v interface{}) error
> ```
>
> 该函数的第一个参数是输入，即JSON格式的文本（比特序列），第二个参数表示目标输出容器，用于存放解码后的值。

3、1 解析JSON到结构体

> 示例代码：
>
> ```
> package main
>
> import (
>     "encoding/json"
>     "fmt"
> )
>
> type IT struct {
>     Company  string   `json:"company"`
>     Subjects []string `json:"subjects"`
>     IsOk     bool     `json:"isok"`
>     Price    float64  `json:"price"`
> }
>
> func main() {
>     b := []byte(`{
> "company": "Lianshi",
> "subjects": [
>   "Go",
>   "Web",
>   "Python",
>   "C++"
> ],
> "isok": true,
> "price": 8980.88
> }`)
> ```

```
    var t IT
    err := json.Unmarshal(b, &t)
    if err != nil {
        fmt.Println("json err:", err)
    }
    fmt.Println(t)
}
```

3、2 解析JSON到map

示例代码：

```
package main

import (
    "encoding/json"
    "fmt"
)


func main() {
    b := []byte(`{
"company": "Lianshi",
"subjects": [
  "Go",
  "Web",
  "Python",
  "C++"
],
"isok": true,
"price": 8980.88
}`)
    //默认返回值类型为interface类型 以map类型进行格式存储
//可以理解为：json的key为map的key json的value为map的value
//格式：map[string]interface{}
    var t interface{}
    err := json.Unmarshal(b, &t)
    if err != nil {
        fmt.Println("json err:", err)
    }
    fmt.Println(t)

    //使用断言判断类型
    m := t.(map[string]interface{})
    for k, v := range m {
        switch val := v.(type) {
        case string:
            fmt.Println(k, "is string", val)
        case int:
            fmt.Println(k, "is int", val)
        case float64:
            fmt.Println(k, "is float64", val)
        case bool:
            fmt.Println(k, "is bool", val)
        case []interface{}:
            fmt.Println(k, "is an array:")
            for i, u := range val {
```

```
                fmt.Println(i, u)
            }
        default:
            fmt.Println(k, "is of a type I don't know how to handle")
        }
    }
}
```

第十四章 文件操作

1 创建与打开文件

新建文件可以通过如下两个方法：

```
//根据提供的文件名创建新的文件，返回一个文件对象，默认权限是0666的文件，返回的文件对象是可读写的。
func Create(name string) (file *File, err Error)

//根据文件描述符创建相应的文件，返回一个文件对象
func NewFile(fd uintptr, name string) *File
```

示例代码：

```
func main()  {
//创建文件
f, err := os.Create("C:/Lianshi/test.txt")
if err != nil {
  fmt.Println("Create err:", err)
  return
}
//关闭文件
defer f.Close()

fmt.Println("create successful")
}
```

通过如下两个方法来打开文件：

```
//该方法打开一个名称为name的文件，但是是只读方式，内部实现其实调用了OpenFile。
func Open(name string) (file *File, err Error)
```

Open() 是以只读权限打开文件名为name的文件，得到的文件指针file，只能用来对文件进行“读”操作。如果有 “写” 文件的需求，就需要借助OpenFile函数来打开了。

```
//打开名称为name的文件，flag是打开的方式，只读、读写等，perm是权限
func OpenFile(name string, flag int, perm uint32) (file *File, err Error)
```

参数介绍

```
OpenFile()可以选择打开name文件的读写权限。这个函数有三个默认参数：
参1：name，表示打开文件的路径。可使用相对路径 或 绝对路径
参2：flg，表示读写模式，常见的模式有：
O_RDONLY(只读模式)， O_WRONLY(只写模式)， O_RDWR(可读可写模式)， O_APPEND(追加模式)。
```

```
如果读取目录只能指定O_RDONLY模式
参3：perm，表权限取值范围（0-7），表示如下：
0[---]：没有任何权限
1[--x]：执行权限(如果是可执行文件，是可以运行的)
2[-w-]：写权限
3[-wx]：写权限与执行权限
4[r--]：读权限
5[r-x]：读权限与执行权限
6[rw-]：读权限与写权限
7[rwx]：读权限，写权限，执行权限
```

示例代码：

```
func main()  {
//以可读可写方式打开文件 如果文件不存在则创建新文件 如果文件存在会覆盖其内容
f, err := os.OpenFile("C:/Lianshi/test.txt", os.O_RDWR | os.O_CREATE, 0600)

if err != nil {
  fmt.Println("OpenFile err: ", err)
  return
}
defer f.Close()
//将字符串写入到文件
f.WriteString("hello world")

fmt.Println("open successful")
}
```

2 写入文件

```
//写入byte类型的信息到文件
func (file *File) Write(b []byte) (n int, err Error)

//在指定位置开始写入byte类型的信息
func (file *File) WriteAt(b []byte, off int64) (n int, err Error)

//写入string信息到文件
func (file *File) WriteString(s string) (ret int, err Error)
```

```
/*
获取文件读写位置
offset:  矢量。 正数，向文件末尾偏移。负数，向文件开头偏移
whence:  偏移的起始位置。
io.SeekStart : 文件起始位置。
io.SeekCurrent : 文件当前位置。
io.SeekEnd: 文件末尾位置。
返回值 ret:  从文件起始位置到，读写位置的偏移量。
*/
func (f *File) Seek(offset int64, whence int) (ret int64, err error)
```

示例代码：

```
package main

import (
```

```
    "fmt"
    "os"
)

func main() {
//新建文件
    f, err := os.Create("C:/Lianshi/test.txt")
    //f, err := os.OpenFile("C:/Lianshi/test.txt",os.O_RDWR | os.O_CREATE,
0666)
    if err != nil {
        fmt.Println(err)
        return
    }
    //关闭文件
    defer f.Close()

    //写入byte类型的信息到文件
    f.Write([]byte("hello world\r\n"))
    //写入指定位置byte类型的信息到文件
    f.WriteAt([]byte("hello world\r\n"), 5)
    //写入string信息到文件
    f.WriteString("hello world\r\n")
}
```

3 读取文件

```
//读取数据到b中
func (file *File) Read(b []byte) (n int, err Error)

//从off开始读取数据到b中
func (file *File) ReadAt(b []byte, off int64) (n int, err Error)
```

示例代码：

```
package main

import (
    "fmt"
    "os"
)

func main() {
    f, err := os.Open("C:/Lianshi/test.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    //关闭文件
    defer f.Close()

    buf := make([]byte, 1024)
    //读取文件，返回值为读取有效字符个数和错误信息
    //n, _ := f.Read(buf)
    //读取指定位置
    n,_:=f.ReadAt(buf,5)
```

```
    fmt.Println(string(buf[:n]))
}
```

4 案例：大文件拷贝

示例代码：

```
package main

import (
"fmt"
"io"
"os"
)

func main() {
args := os.Args //获取用户输入的所有参数

//如果用户没有输入,或参数个数不够,则调用该函数提示用户
if args == nil || len(args) != 3 {
  fmt.Println("useage : xxx srcFile dstFile")
  return
}

srcPath := args[1] //获取输入的第一个参数
dstPath := args[2] //获取输入的第二个参数
fmt.Printf("srcPath = %s, dstPath = %s\n", srcPath, dstPath)

if srcPath == dstPath {
  fmt.Println("源文件和目的文件名字不能相同")
  return
}

srcFile, err1 := os.Open(srcPath) //打开源文件
if err1 != nil {
  fmt.Println(err1)
  return
}

dstFile, err2 := os.Create(dstPath) //创建目的文件
if err2 != nil {
  fmt.Println(err2)
  return
}

buf := make([]byte, 1024) //切片缓冲区
for {
  //从源文件读取内容，n为读取文件内容的长度
  n, err := srcFile.Read(buf)
  if err != nil && err != io.EOF {
      fmt.Println(err)
      break
  }

  if n == 0 {
      fmt.Println("文件处理完毕")
      break
```

```
    }

    //切片截取
    tmp := buf[:n]
    //把读取的内容写入到目的文件
    dstFile.Write(tmp)
}

//关闭文件
srcFile.Close()
dstFile.Close()
}
```

第十五章 表格处理

1 下载和使用xlsx包

使用的包：

```
github.com/tealeg/xlsx
```

get下载：

```
go get github.com/tealeg/xlsx
```

mod下载：

```
go mod download github.com/tealeg/xlsx
```

```
package main

import (
    "fmt"
    "github/tealeg/xlsx"
)
func main() {
    //新建表格
    file = xlsx.NewFile()
//保存文件
    err = file.Save("文件路径.xlsx")
    if err != nil {
        fmt.Printf(err.Error())
    }
}
```

2 创建和写入表格

示例代码：

```
package main

import (
    "fmt"
```

```
    "github/tealeg/xlsx"
)

//列
type Cell struct {
    Cname []string
}

//行
type Row struct {
    Cell
}

//表
type Sheet struct {
    SheetName string
    Row
}

//创建表格
func CreateExcel(data Sheet) {

    var file *xlsx.File
    var sheet *xlsx.Sheet
    var row *xlsx.Row
    var cell *xlsx.Cell
    var err error
    //新建表格
    file = xlsx.NewFile()
    //设置当前页名称
    sheet, err = file.AddSheet(data.SheetName)

    if err != nil {
        fmt.Printf(err.Error())
    }
    //添加行
    row = sheet.AddRow()
    //设置行高
    row.SetHeightCM(1.0)
    //循环添加列 并设置内容
    for _,v:=range data.Cname  {
        cell = row.AddCell()
        cell.Value=v
    }
    //保存文件
    err = file.Save("D:/学员信息表.xlsx")
    if err != nil {
        fmt.Printf(err.Error())
    }

}
func main() {
    //设置表头信息
    sheet := Sheet{"学员信息表",
        Row{Cell{[]string{"姓名", "性别", "年龄", "成绩", "住址"}}}}
    CreateExcel(sheet)
}
```

3 读取和操作表格

> 示例代码：

```
package main

import (
    "fmt"
    "github/tealeg/xlsx"
)

type Person struct {
    Name       string //微信昵称
    Education  string //学历
    University string //高校
    Industry   string //行业
    Workyear   string //工作年限
    Position   string //职位
    Salary     string //薪资
    Language   string //编程语言
}

func Getxlsx() {
    var per []Person
    //读取Excel文件
    file, err := xlsx.OpenFile("D:/Go语言工程师信息表.xlsx")
    if err != nil {
        fmt.Printf("open failed: %s\n", err)
        return
    }
    //页
    for _, sheet := range file.Sheets {
        //行
        for _, row := range sheet.Rows {
            //列
            var temp Person
            //将Excel每一列文件读取放在字符串切片中
            var str []string
            for _, cell := range row.Cells {
                str = append(str, cell.String())
            }
            //按照列顺序将数据存储在结构体中
            temp.Name = str[0]
            temp.Education = str[1]
            temp.University = str[2]
            temp.Industry = str[3]
            temp.Workyear = str[4]
            temp.Position = str[5]
            temp.Salary = str[6]
            temp.Language = str[7]

            //将结构体放在结构体切片per中
            per = append(per, temp)
        }
    }
    fmt.Println(per)
}
```

```
func main() {
    Getxlsx()
}
```

第十六章 Go mod

Go mod模块是相关Go包的集合。modules源代码交换和版本控制的单元。go命令直接支持使用modules，包括记录和解析对其他模块的依赖性。modules替换旧的基于GOPATH的方法来指定在给定构建中使用哪些源文件。

设置 GO111MODULE

- off

  go命令行将不会支持module功能，寻找依赖包的方式将会沿用旧版本那种通过vendor目录或者GOPATH模式来查找。

- on

  go命令行会使用modules，而一点也不会去GOPATH目录下查找。

- auto

  默认值，go命令行将会根据当前目录来决定是否启用module功能

当modules 功能启用时，依赖包的存放位置变更为$GOPATH/pkg，允许同一个package多个版本并存，且多个项目可以共享缓存的 module。

1 go mod 命令

go mod 有以下命令来管理包。

命令	说明
download	download modules to local cache(下载依赖包)
edit	edit go.mod from tools or scripts（编辑go.mod)
graph	print module requirement graph (打印模块依赖图)
init	initialize new module in current directory（在当前目录初始化mod）
tidy	add missing and remove unused modules(拉取缺少的模块，移除不用的模块)
vendor	make vendored copy of dependencies(将依赖复制到vendor下)
verify	verify dependencies have expected content (验证依赖是否正确）
why	explain why packages or modules are needed(解释为什么需要依赖)

2 go mod 创建项目

在GOPATH 目录之外新建一个目录，并使用go mod init 初始化生成go.mod文件

```
go mod init 项目名
```

> go.mod文件一旦创建后，它的内容将会被go toolchain全面掌控。go toolchain会在各类命令执行时，比如go get、go build、go mod等修改和维护go.mod文件。
>
> go.mod 提供了module, require、replace和exclude 四个命令
>
> - module 语句指定包的名字（路径）
> - require 语句指定的依赖项模块
> - replace 语句可以替换依赖项模块
> - exclude 语句可以忽略依赖项模块

```
package main

import "github.com/astaxie/beego"

func main(){
    beego.Run()
}
```

> 执行 go run test.go 运行代码会发现 go mod 会自动查找依赖自动下载并完成。