Day5 goroutine和channel

tony

目录

Goroutine使用介绍

1. 创建goroutine

```
package main

import (
    "fmt"
)

func hello() {
    fmt.Println("Hello world goroutine")
}
func main() {
    go hello()
    fmt.Println("main function")
}
```

Goroutine使用介绍

2. 修复代码

```
package main

import (
    "fmt“
    “time”
)

func hello() {
    fmt.Println("Hello world goroutine")
}
func main() {
    go hello()
    time.Sleep(1*time.Second)
    fmt.Println("main function")
}
```

Goroutine使用介绍

3. 启动多个goroutine

```
package main

import (
    "fmt"
    "time"
)

func numbers() {
    for i := 1; i <= 5; i++ {
        time.Sleep(250 * time.Millisecond)
        fmt.Printf("%d ", i)
    }
}
func alphabets() {
    for i := 'a'; i <= 'e'; i++ {
        time.Sleep(400 * time.Millisecond)
        fmt.Printf("%c ", i)
    }
}
func main() {
    go numbers()
    go alphabets()
    time.Sleep(3000 * time.Millisecond)
    fmt.Println("main terminated")
}
```

Goroutine介绍

1. 进程和线程

A. 进程是程序在操作系统中的一次执行过程，系统进行资源分配和调度的一个独立单位。

B. 线程是进程的一个执行实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位。

C.一个进程可以创建和撤销多个线程;同一个进程中的多个线程之间可以并发执行.

Goroutine

2. 进程和线程

Goroutine

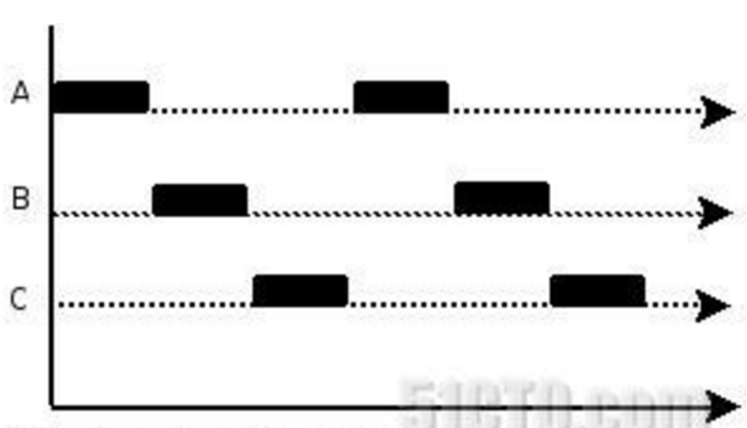
3. 并发和并行

A. 多线程程序在一个核的cpu上运行，就是并发

B. 多线程程序在多个核的cpu上运行，就是并行

Goroutine

4. 并发和并行



Concurrency : 1. Single Processor

Parallelism : 1. Multiprocessores, Multicore

Goroutine

5. 协程和线程

协程：独立的栈空间，共享堆空间，调度由用户自己控制，本质上有点类似于用户级线程，这些用户级线程的调度也是自己实现的

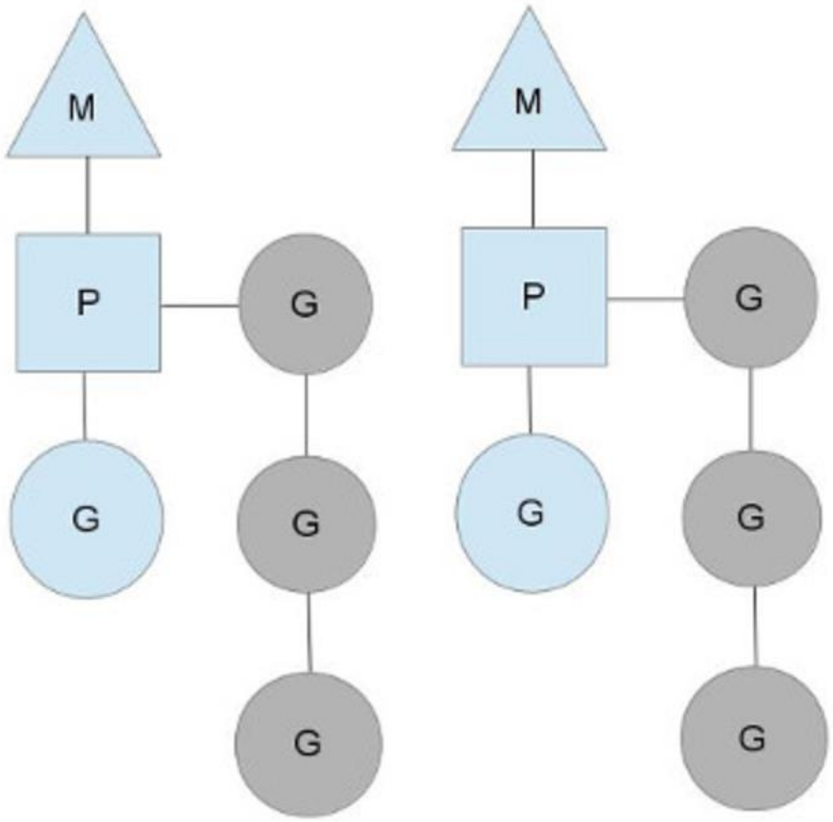
线程：一个线程上可以跑多个协程，协程是轻量级的线程。

Goroutine

6. goroutine调度模型

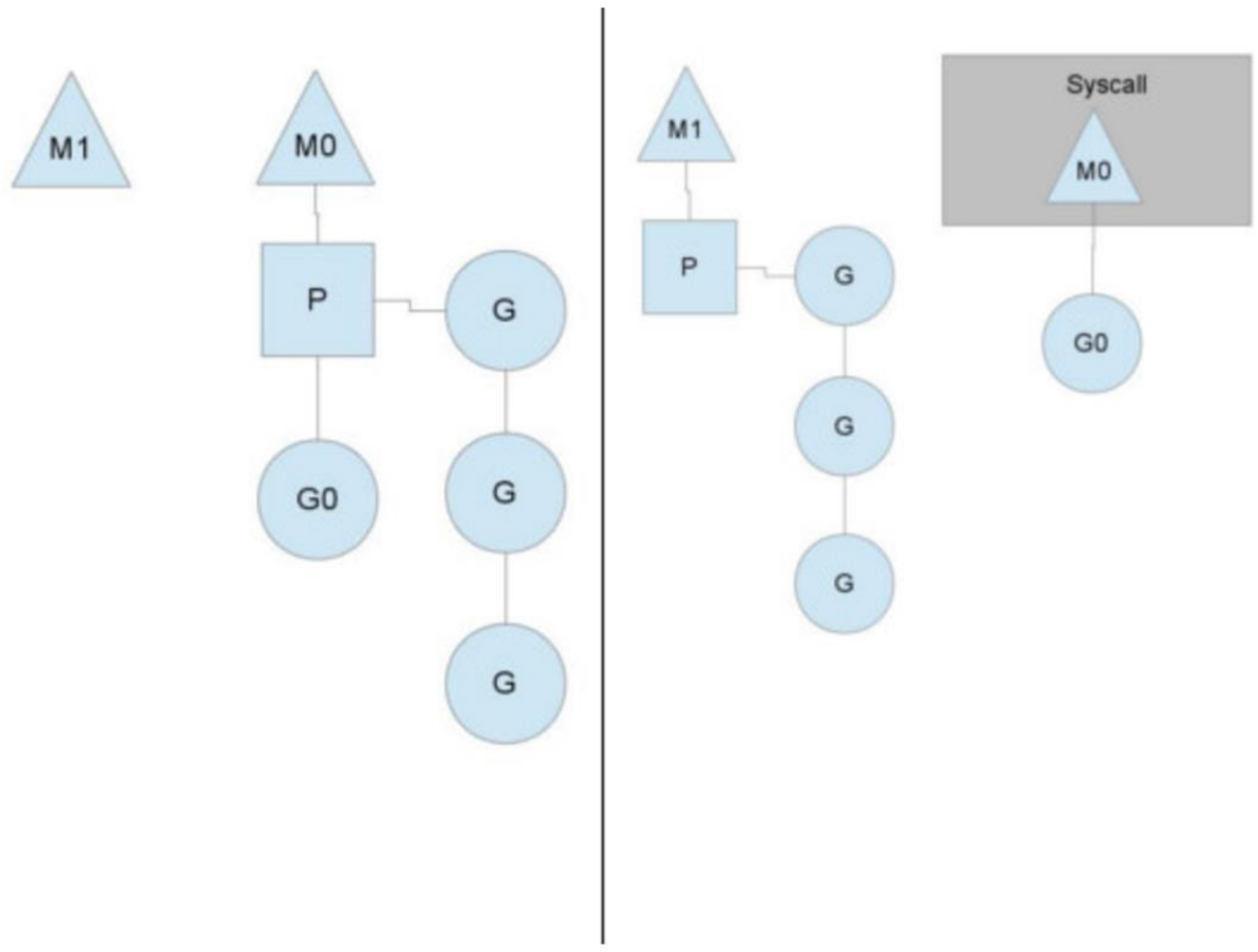
Goroutine

7. goroutine调度模型



Goroutine

8. goroutine调度模型



Goroutine

9. 如何设置golang运行的cpu核数

```
package main
import (
	"fmt"
	"runtime"
)

func main() {
	num := runtime.NumCPU()
	runtime.GOMAXPROCS(num)
	fmt.Println(num)
}
```

Goroutine

11. goroute练习

Channel

1. 不同goroutine之间如何进行通讯?

    a. 全局变量和锁同步

    b. Channel

Channel

2. channel概念

a. 类似unix中管道（pipe）

b. 先进先出

c. 线程安全，多个goroutine同时访问，不需要加锁

d. channel是有类型的，一个整数的channel只能存放整数

Channel

3. channel声明

var 变量名 chan 类型

var test chan int

var test chan string

var test chan map[string]string

var test chan stu

var test chan *stu

4. channel初始化

使用make进行初始化，比如：

```
var test chan int
test = make(chan int, 10)

var test chan string
test = make(chan string, 10)
```

Channel

5. channel基本操作

1. 从channel读取数据:

```
var testChan chan int
testChan = make(chan int, 10)
var a int
a = <- testChan
```

2. 从channel写入数据:

```
var testChan chan int
testChan = make(chan int, 10)
var a int  = 10
testChan <- a
```

6. goroutine和channel相结合

```
package main

import (
	"fmt"
	"time"
)

func main() {
	ch := make(chan string)
	go sendData(ch)
	go getData(ch)
	time.Sleep(100 * time.Second)
}

func sendData(ch chan string) {
	ch <- "Washington"
	ch <- "Tripoli"
	ch <- "London"
	ch <- "Beijing"
	ch <- "Tokio"
}

func getData(ch chan string) {
	var input string
	for {
		input = <-ch
		fmt.Println(input)
	}
}
```

7. channel阻塞

```
package main

import (
	"fmt"
	"time"
)

func main() {
	ch := make(chan string)
	go sendData(ch)
	time.Sleep(100 * time.Second)
}

func sendData(ch chan string) {
	var i int
	for {
		var str string
		str = fmt.Sprintf("stu %d", i)
		fmt.Println("write:", str)
		ch <- str
		i++
	}
}
```

8. 带缓冲区的channel

1. 如下所示，testChan长度为0：

```
var testChan chan int
testChan = make(chan int)
var a int
a = <- testChan
```

2. 如下所示，testChan是带缓冲区的chan，一次可以放10个元素:

```
var testChan chan int
testChan = make(chan int, 10)
var a int  = 10
testChan <- a
```

9. chan之间的同步

```
package main

import (
	"fmt"
	"time"
)

func main() {
	ch := make(chan string)
	go sendData(ch)
	go getData(ch)
	time.Sleep(100 * time.Second)
}

func sendData(ch chan string) {
	ch <- "Washington"
	ch <- "Tripoli"
	ch <- "London"
	ch <- "Beijing"
	ch <- "Tokio"
}

func getData(ch chan string) {
	var input string
	for {
		input = <-ch
		fmt.Println(input)
	}
}
```

10. for range遍历chan

```
package main

import (
	"fmt"
	"time"
)

func main() {
	ch := make(chan string)
	go sendData(ch)
	go getData(ch)
	time.Sleep(100 * time.Second)
}

func sendData(ch chan string) {
	ch <- "Washington"
	ch <- "Tripoli"
	ch <- "London"
	ch <- "Beijing"
	ch <- "Tokio"
}

func getData(ch chan string) {

	for input  := range ch {
		fmt.Println(input)
	}
}
```

11. chan的关闭

    1. 使用内置函数close进行关闭，chan关闭之后，for range遍历chan中已经存在的元素后结束

    2. 使用内置函数close进行关闭，chan关闭之后，没有使用for range的写法需要使用，v, ok := <- ch进行判断chan是否关闭

12. chan的只读和只写

a. 只读chan的声明

Var 变量的名字 <-chan int

Var readChan <- chan int

b. 只写chan的声明

Var 变量的名字 chan<- int

Var writeChan chan<- int

12. 对chan进行select操作

```
Select {
    case u := <- ch1:
    case e := <- ch2:
    default:
}
```

13. 练习

```
Select {
    case u := <- ch1:
    case e := <- ch2:
    default:
}
```

14. 定时器的使用

```
package main

import (
	"fmt"
	"time"
)

func main() {
	t := time.NewTicker(time.Second)
	for v := range t.C {
		fmt.Println("hello, ", v)
	}
}
```

15. 一次定时器

```
package main

import (
        "fmt"
        "time"
)

func main() {
        select {
         Case <- time.After(time.Second):
          fmt.Println(“after”)
          }
}
```

16. 超时控制

```
package main

import (
	"fmt"
	"time"
)
func queryDb(ch chan int) {

	time.Sleep(time.Second)
	ch <- 100
}
func main() {
	ch := make(chan int)
	go queryDb(ch)
	t := time.NewTicker(time.Second)

	select {
	case v := <-ch:
		fmt.Println("result", v)
	case <-t.C:
		fmt.Println("timeout")
	}
}
```

17. goroutine中使用recover

应用场景，如果某个goroutine panic了，而且这个goroutine里面没有捕获(recover)，那么整个进程就会挂掉。所以，好的习惯是每当go产生一个goroutine，就需要写下recover

线程安全介绍

7. 现实例子

   A. 多个goroutine同时操作一个资源，这个资源又叫临界区

   B. 现实生活中的十字路口，通过红路灯实现线程安全

   C. 火车上的厕所，通过互斥锁来实现线程安全

线程安全介绍

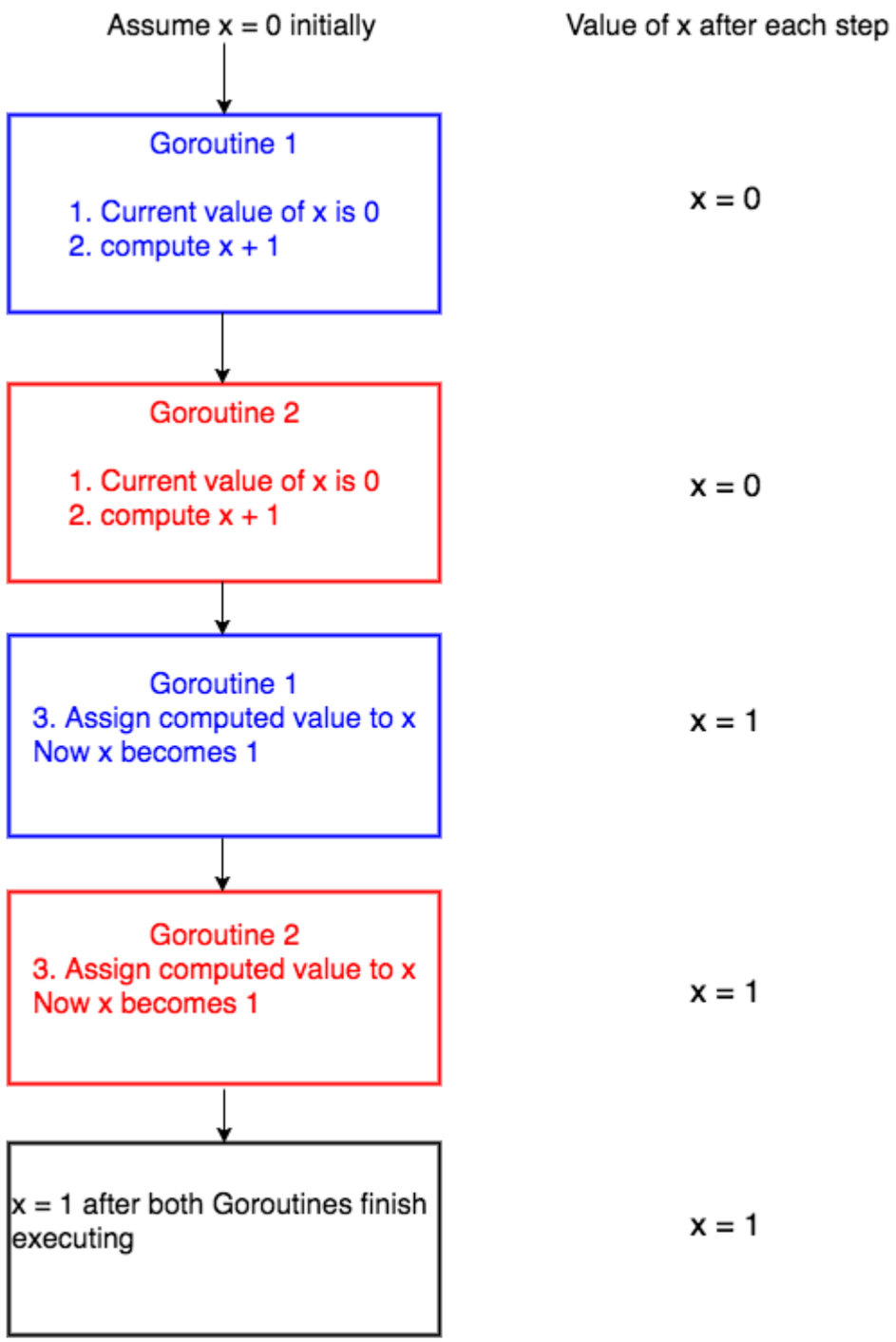
8. 实际例子， x = x +1

   A. 先从内存中取出x的值

   B. CPU进行计算，x+1

   C. 然后把x+1的结果存储在内存中

线程安全介绍


Assume x = 0 initially
Value of x after each step
Goroutine 1
1. Current value of x is 0
2. compute x + 1
x = 0
Goroutine 2
1. Current value of x is 0
2. compute x + 1
x = 0
Goroutine 1
3. Assign computed value to x
Now x becomes 1
x = 1
Goroutine 2
3. Assign computed value to x
Now x becomes 1
x = 1
x = 1 after both Goroutines finish executing
x = 1

线程安全介绍

9. 互斥锁介绍

A. 同时有且只有一个线程进入临界区，其他的线程则在等待锁

B. 当互斥锁释放之后，等待锁的线程才可以获取锁进入临界区

C. 多个线程同时等待同一个锁，唤醒的策略是随机的

线程安全介绍

有问题的代码!

```
package main
import (
    "fmt"
    "sync"
    )
var x  = 0
func increment(wg *sync.WaitGroup) {
    x = x + 1
    wg.Done()
}
func main() {
    var w sync.WaitGroup
    for i := 0; i < 1000; i++ {
        w.Add(1)
        go increment(&w)
    }
    w.Wait()
    fmt.Println("final value of x", x)
}
```

线程安全介绍

使用互斥锁fix

```
package main
import (
    "fmt"
    "sync"
    )
var x  = 0
func increment(wg *sync.WaitGroup, m *sync.Mutex) {
    m.Lock()
    x = x + 1
    m.Unlock()
    wg.Done()
}
func main() {
    var w sync.WaitGroup
    var m sync.Mutex
    for i := 0; i < 1000; i++ {
        w.Add(1)
        go increment(&w, &m)
    }
    w.Wait()
    fmt.Println("final value of x", x)
}
```

读写锁介绍

10. 读写锁使用场景

    A. 读多写少的场景

    B. 分为两种角色，读锁和写锁

    C. 当一个goroutine获取写锁之后，其他的goroutine获取写锁或读锁都会等待

    D. 当一个goroutine获取读锁之后，其他的goroutine获取写锁都会等待, 但其他goroutine获取读锁时，都会继续获得锁.

读写锁介绍

11.读写锁案例演示

读写锁介绍

12. 读写锁和互斥锁性能比较

Waitgroup介绍

23. 如何等待一组goroutine结束?

A. 方法一，使用**不带缓冲区**的channel实现

```
package main
import (
    "fmt"
    "time"
)
func process(i int, ch chan bool) {
    fmt.Println("started Goroutine ", i)
    time.Sleep(2 * time.Second)
    fmt.Printf("Goroutine %d ended\n", i)
    ch <- true
}
func main() {
    no := 3
    exitChan := make(chan bool, no)
    for i := 0; i < no; i++ {
        go process(i, exitChan)
    }
    for i := 0; I < no;i++{
        <-exitChan
    }
    fmt.Println("All go routines finished executing")
}
```

Waitgroup介绍

24. 如何等待一组goroutine结束?

B. 方法二，使用sync.WaitGroup实现

```
package main

import (
    "fmt"
    "sync"
    "time"
)
func process(i int, wg *sync.WaitGroup) {
    fmt.Println("started Goroutine ", i)
    time.Sleep(2 * time.Second)
    fmt.Printf("Goroutine %d ended\n", i)
    wg.Done()
}
func main() {
    no := 3
    var wg sync.WaitGroup
    for i := 0; i < no; i++ {
        wg.Add(1)
        go process(i, &wg)
    }
    wg.Wait()
    fmt.Println("All go routines finished executing")
}
```

原子操作

13. 原子操作

A. 加锁代价比较耗时，需要上下文切换

B. 针对基本数据类型，可以使用原子操作保证线程安全

C. 原子操作在用户态就可以完成，因此性能比互斥锁要高

原子操作

14. 原子操作介绍

```
func AddInt32(addr *int32, delta int32) (new int32)
func AddInt64(addr *int64, delta int64) (new int64)
func AddUint32(addr *uint32, delta uint32) (new uint32)
func AddUint64(addr *uint64, delta uint64) (new uint64)
func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)
```

加减操作

```
func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool)
func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)
func CompareAndSwapUint32(addr *uint32, old, new uint32) (swapped bool)
func CompareAndSwapUint64(addr *uint64, old, new uint64) (swapped bool)
func CompareAndSwapUintptr(addr *uintptr, old, new uintptr) (swapped bool)
```

比较并交换

```
func LoadInt32(addr *int32) (val int32)
func LoadInt64(addr *int64) (val int64)
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
func LoadUint32(addr *uint32) (val uint32)
func LoadUint64(addr *uint64) (val uint64)
func LoadUintptr(addr *uintptr) (val uintptr)
```

读取操作

```
func StoreInt32(addr *int32, val int32)
func StoreInt64(addr *int64, val int64)
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
func StoreUint32(addr *uint32, val uint32)
func StoreUint64(addr *uint64, val uint64)
func StoreUintptr(addr *uintptr, val uintptr)
```

写入操作

```
func SwapInt32(addr *int32, new int32) (old int32)
func SwapInt64(addr *int64, new int64) (old int64)
func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)
func SwapUint32(addr *uint32, new uint32) (old uint32)
func SwapUint64(addr *uint64, new uint64) (old uint64)
func SwapUintptr(addr *uintptr, new uintptr) (old uintptr)
```

交换操作

课后作业

1. 缩略图批量转换

   A. 用户指定一个图片目录，目录中包括许多图片。

   B. 写一个程序，遍历该目录的所有图片，为所有图片生成缩略图。

   C. 缩略图的命名方式如下：原文件名_thumb.png，比如原文件: a.png，则生成的缩略图为：a_thumb.png

   D. 生成缩略图的操作是一个耗时的操作，本程序需要采用多个goroutine完成.

   E. 参考程序: https://github.com/sherlockhua/gostudy03/blob/master/image_thumb/image_thumb.go

QA