Go 语言从入门到项目实战

第9章 并发、并行与协程

9.1 概念

- 进程（Process）：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。可以把一个独立正在运行的软件实例看作一个进程；
- 线程（Thread）：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。一个独立正在运行的软件实例里可以产生多个线程；
- 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。

9.1 概念

- 并发（Concurrency）：并发是指多线程程序在单核心的CPU上运行；
- 并行（Parallelism）：并行是指多线程程序在多核心的CPU上运行；
- 并发与并行并不相同，并发主要由切换时间片来实现“同时”运行，并行则是直接利用多核实现多线程的运行。

9.1 概念

- 协程：协程是指内存中独立的栈空间，共享堆空间。调度由用户自己控制，可以看作是“用户级”线程；
- 线程：一个线程上可以跑多个协程，协程是轻量级的线程。

9.2 Go语言协程Goroutine

- Goroutine是由Go语言官方实现的超级“线程池”；
- 每个实例仅占用4-5KB的栈内存空间；
- Goroutine可以看作是线程（本质上是协程），它由Go语言运行时自动完成调度和管理。

9.2 Go语言协程Goroutine

- 示例

```
var syncWait sync.WaitGroup
func main() {
        syncWait.Add(1)
        go hello()
        fmt.Println("程序运行结束")
        syncWait.Wait()
}
func hello(){
        defer syncWait.Done()
        fmt.Println("你好，三酷猫")
}
```

9.2 Go语言协程Goroutine

- 示例

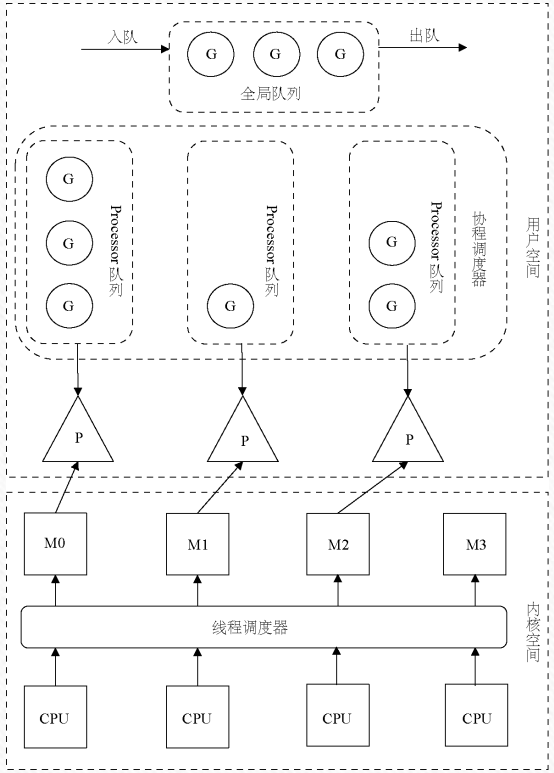
```
var syncWait sync.WaitGroup
func main() {
        syncWait.Add(1)
        go func() {
                defer syncWait.Done()
                fmt.Println("你好，三酷猫")
        }()
        syncWait.Wait()
        fmt.Println("程序运行结束")
}
```

9.2 Go语言协程Goroutine

- 示例

```
var syncWait sync.WaitGroup
func main() {
        for i := 0; i < 10; i++ {
                syncWait.Add(1)
                go hello(i)
        }
        syncWait.Wait()
        fmt.Println("程序运行结束")
}
func hello(index int)  {
        defer syncWait.Done()
        fmt.Println("你好，三酷猫",index)
}
```

9.3 Go语言调度模型 GPM



- 全局队列中包含所有排队中的Goroutine任务；
- G指单个Goroutine任务；
- Processor队列是分配好的排队中的Goroutine任务；
- P指Processor，所有的Processor在程序启动时创建，并保存在数组中；
- M0-M3指Machine，是内核线程。

9.4 runtime 包

- 获取当前操作系统信息；
- 获取CPU架构类型；
- 获取/设置CPU核心数量；
- 让出系统资源；
- 终止当前协程。

9.5 在协程任务间传递数据Channel

- 通道的声明格式

  var channel_name chan value_type

  make(chan value_type,[buffer_size])

- 示例

  var intChan chan int

  var stringChan chan string

  var arrayChan chan []bool

  var intChan=make(chan int)

9.5 在协程任务间传递数据Channel

- 示例

```
var intChan=make(chan int)                          //声明int型通道变量intChan
fmt.Println(intChan,reflect.TypeOf(intChan))        //输出intChan变量的值和类型
intChan<-100                                        //将整型值100通过intChan通道发送
intValue:=<-intChan                                 //读取intChan通道中的值，并将结果赋值
给intValue变量
fmt.Println(intValue,reflect.TypeOf(intValue))      //输出intValue变量的值和类型
close(intChan)                                      //关闭intChan通道
```

9.5 在协程任务间传递数据Channel

- 示例

```
func main() {
    var intChan=make(chan int)                          //声明int型通道变量intChan
    fmt.Println(intChan,reflect.TypeOf(intChan))        //输出intChan变量的值和类型
    go intChanRecvListener(intChan
    intChan<-100 )                                      //将整型值100通过intChan通道发送
    close(intChan)                                      //关闭intChan通道
}
```

9.5 在协程任务间传递数据Channel

- 示例

```
func intChanRecvListener(intChan chan int){
    //读取intChan通道中的值，并将结果赋值给intValue变量
    intValue:=<-intChan
    //输出intValue变量的值和类型
    fmt.Println(intValue,reflect.TypeOf(intValue))
}
```

9.5 在协程任务间传递数据Channel

- 示例

```
func main() {
        var intChan=make(chan int,2)                                    //声明int型通道变量intChan，缓冲区大小为2
        fmt.Println(intChan,reflect.TypeOf(intChan),len(intChan),cap(intChan))    //输出intChan变量的值和类型
        intChan<-100                                                    //将整型值100通过intChan通道发送
        fmt.Println(len(intChan),cap(intChan))
        intChan<-200
        fmt.Println(len(intChan),cap(intChan))
        go intChanRecvListener(intChan)
        time.Sleep(2*time.Second)
        close(intChan)                                                  //关闭intChan通道
}
```

9.5 在协程任务间传递数据Channel

- 示例

```
func intChanRecvListener(intChan chan int){
        intValue:=<-intChan                          //读取intChan通道中的值，并将结果赋值给intValue变量
        fmt.Println(intValue,reflect.TypeOf(intValue))          //输出intValue变量的值和类型
        fmt.Println("成功接收第1个值",len(intChan),cap(intChan))
        intValue=<-intChan
        fmt.Println(intValue,reflect.TypeOf(intValue))          //输出intValue变量的值和类型
        fmt.Println("成功接收第2个值",len(intChan),cap(intChan))
}
```

9.5 在协程任务间传递数据Channel

- 示例

```
intChan := make(chan int, 10)
intChan <- 1
intChan <- 2
intChan <- 3
close(intChan)
for {
        intValue, isOpen := <-intChan
        if !isOpen {
                fmt.Println("通道已关闭")
                break
        }
        fmt.Println(intValue)
}
```

9.5 在协程任务间传递数据Channel

- 单向通道的声明格式

  var chan_name chan<- value_type // 只能发送数据的通道

  var chan_name <-chan value_type // 只能读取数据的通道

- 示例

  intChan := make(chan int)

  var sendOnlyIntChan chan<- int=intChan //声明变量sendOnlyIntChan，将intChan限制为只能发送的通道

  Var recvOnlyIntChan <-chan int=intChan //声明变量recvOnlyIntChan，将intChan限制为只能接收的通道

9.6 Select 结构

```
func main() {
	chan1 := make(chan int, 5)
	chan2 := make(chan int, 5)
	go recvFunc(chan1,chan2)
	go sendFunc1(chan1)
	go sendFunc2(chan2)
	time.Sleep(5 * time.Second)
	fmt.Println("main()函数结束")
}
```

```
func sendFunc1(chan1 chan int){
	for i := 0; i < 5; i++ {
		chan1 <- i
		time.Sleep(1 * time.Second)
	}
}
func sendFunc2(chan2 chan int){
	for i := 10; i >= 5; i-- {
		chan2 <- i
		time.Sleep(1 * time.Second)
	}
}
```

9.6 Select 结构

```
func recvFunc(chan1 chan int,chan2 chan int){
	for {
		select {
		case intValue1 := <-chan1:
			fmt.Println("接收到chan1通道的值：", intValue1)
		case intValue2 := <-chan2:
			fmt.Println("接收到chan2通道的值：", intValue2)
		}
	}
}
```

9.7 锁和原子操作

```
var count int
var locker sync.Mutex
func main() {
        go countPlus(10000)
        go countPlus(10000)
        time.Sleep(2*time.Second)
        fmt.Println(count)
}
```

```
func countPlus(times int){
        for i:=0;i<times;i++{
                locker.Lock()
                count++
                locker.Unlock()
        }
}
```

9.7 锁和原子操作

```
var count int
var locker sync.RWMutex
func main() {
        for i := 1; i <= 3; i++ {
                go write(i)
        }
        for i := 1; i <= 3; i++ {
                go read(i)
        }
        time.Sleep(10 * time.Second)
        fmt.Println("count值为：", count)
}
```

```
func read(i int) {
        fmt.Println("读操作",i)
        locker.RLock()
        fmt.Println(i,"读count的值为",count)
        time.Sleep(1 * time.Second)
        locker.RUnlock()
}
func write(i int) {
        fmt.Println("写操作",i)
        locker.Lock()
        count++
        fmt.Println(i,"写count的值为",count)
        time.Sleep(1 * time.Second)
        locker.Unlock()
}
```

9.7 锁和原子操作

- 原子操作
  - 使用 atomic 包

操作分类	函数声明
读操作	func LoadInt32(addr int32) (val int32) func LoadInt64(addr int64) (val int64) func LoadUint32(addr uint32) (val uint32) func LoadUint64(addr uint64) (val uint64) func LoadUintptr(addr uintptr) (val uintptr) func LoadPointer(addr unsafe.Pointer) (val unsafe.Pointer)

9.7 锁和原子操作

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

9.7 锁和原子操作

交换操作	**func SwapInt32(addr *int32, new int32) (old int32)** **func SwapInt64(addr *int64, new int64) (old int64)** **func SwapUint32(addr *uint32, new uint32) (old uint32)** **func SwapUint64(addr *uint64, new uint64) (old uint64)** **func SwapUintptr(addr *uintptr, new uintptr) (old uintptr)** **func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)**
比较并交换操作	func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool) func CompareAndSwapInt64(addr *int64, old, new int64) (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) func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)

9.8 定时器

- Timer

```
timer1 := time.NewTimer(2 * time.Second)
t1 := time.Now()
fmt.Printf("t1:%v\n", t1)
t2 := <-timer1.C
fmt.Printf("t2:%v\n", t2)
```

9.8 定时器

- Timer

```
func main() {
        fmt.Println(time.Now())
        time.AfterFunc(2*time.Second,sayHello)
        time.Sleep(3*time.Second)
}
func sayHello(){
        fmt.Println("三酷猫打招呼",time.Now())
}
```

9.8 定时器

- Ticker

```
func main() {
        ticker := time.NewTicker(1 * time.Second)
        defer ticker.Stop()
        for range ticker.C {
                sayHello()
        }
}
func sayHello(){
        fmt.Println("三酷猫打招呼",time.Now())
}
```