Go Goroutine 究竟可以开多少

要知道Goroutine最多可以开多少，我们要先搞清楚下面几个问题

1.Goroutine 是什么

2.开Goroutine 需要消耗什么资源

**Goroutine 是什么？ **

Go抽象出来的轻量级线程，在应用层做调度，让我们能够很方便的进行并发编程。

通过 `go` 关键字就可以启动

> 译器会通过`cmd/compile/internal/gc.state.stmt`和`cmd/compile/internal/gc.state.call`两个方法将该关键字转换成`runtime.newproc`函数调用，详细分析可以看[《Go设计与实现》](http://cxyroad.com/”https://link.zhihu.com/?target=https%3A//draveness.me/golang/docs/part3-runtime/ch06-concurrency/golang-goroutine/”)

启动一个新的 Goroutine 来执行任务时，会通过 `runtime.newproc` 初始化一个 `g` 来运行协程

**Goroutine 需要消耗多少资源？ **

内存的消耗

通过开启协程并进行阻塞，来查看前后内存的变化情况

```
func getGoroutineMemConsume() {
var c chan int
var wg sync.WaitGroup
const goroutineNum = 1000000

memConsumed := func() uint64 {
runtime.GC() //GC，排除对象影响
var memStat runtime.MemStats
runtime.ReadMemStats(&memStat)
return memStat.Sys
}

noop := func() {
wg.Done()
<-c //防止goroutine退出，内存被释放
}

wg.Add(goroutineNum)
before := memConsumed() //获取创建goroutine前内存
for i := 0; i < goroutineNum; i++ {
go noop()
}
wg.Wait()
after := memConsumed() //获取创建goroutine后内存
fmt.Println(runtime.NumGoroutine())
fmt.Printf("%.3f KB bytes\n", float64(after-before)/goroutineNum/1024)
}
```

结果分析：

每个协程至少需要消耗 2KB 的空间，那么假设计算机的内存是 2GB，那么至多允许 2GB/2KB = 1M 个协程同时存在。

CPU的消耗

一个Goroutine消耗多少CPU 实际上跟执行函数的逻辑有着很大的关系，如果执行的函数是CPU密集型的计算，并且持续的时间很长，那么这个时候CPU就会优先到达瓶颈。

衡量一段代码能开多少协程同时并发在跑，还得看程序内跑是什么内容，如果

都是非常消耗内存的网络操作，几个Goroutine就可以跑崩溃

结论
--

协程数能看多少取决于打开协程处理方法所占的CPU和内存，如果只是空的操作，那么理论上内存会首先成为瓶颈，此时2G的内存跑满之后程序会出现错误。如果是CPU密集型的话则可能两三个协程就会让程序出现异常。

Goroutine 过多常见触发的问题

1.too many open files， 这种情况下因为打开的文件socket 过多

2.out of memory

业务中的应用

**如何控制并发数？ **

> runtime.NumGoroutine() 可以监控 Goroutine的数量

1.任务只有一个协程在运行

接口需要如果打开并发去进行操作时则需要在应用层控制好并发数，比如开Goroutine初始化资源的数据只需要初始化一次，不需要多个协程同时去进行初始化，可以通过`running flag` 来判断是否正在初始化

```
// SingerConcurrencyRunner 保证任务只有一个在跑
type SingerConcurrencyRunner struct {
isRunning bool
sync.Mutex
}
```

```
func NewSingerConcurrencyRunner() *SingerConcurrencyRunner {
return &SingerConcurrencyRunner{}
}

func (c *SingerConcurrencyRunner) markRunning() (ok bool) {
c.Lock()
defer c.Unlock()
// 二次检查，避免外部检查成功后又被其他协程抢占
if c.isRunning {
return false
}
// 标记成功
c.isRunning = true
return true
}

func (c *SingerConcurrencyRunner) unmarkRunning() (ok bool) {
c.Lock()
defer c.Unlock()
if !c.isRunning {
return false
}
// unmark 成功
c.isRunning = false
return true
}

func (c *SingerConcurrencyRunner) Run(f func()) {
// 如果已经有并发在跑，则不进入，直接返回，可以防止开过多Goroutine导
致内存消耗
if c.isRunning {
return
}
if !c.markRunning() {
//抢占失败则直接返回
return
}

// 执行真正的逻辑
go func() {
defer func() {
if err := recover(); err != nil {
// log
}
}()
f()
c.unmarkRunning()
```

```
}()
}
```

可靠性测试，看是否有超过2个协程在运行

```
func TestConcurrency(t *testing.T) {
runner := NewConcurrencyRunner()
for i := 0; i < 100000; i++ {
runner.Run(f)
}
}

func f() {
        // 这里一直不会超过对应协程数
if runtime.NumGoroutine() > 3 {
fmt.Println(">3", runtime.NumGoroutine())
}
}
```

2.任务有指定协程数运行

其他协程则进入等待并设置对应的超时，或者可以用旧数据直接进行返回，则无需等待。

> [github.com/Jeffail/tun...](http://cxyroad.com/"https://link.zhihu.com/?target=https%3A//github.com/Jeffail/tunny")

通过该包可以实现协程数的控制，如果 `Worker` 已经全部被占满的话，则不会获得 WorkRequest 进行处理 ，而是写入在 `reqChan` 中进行等待

```
func (w *workerWrapper) run() {
//...
        for {
// NOTE: Blocking here will prevent the worker from closing down.
```

```
w.worker.BlockUntilReady()
select {
case w.reqChan <- workRequest{
jobChan:       jobChan,
retChan:       retChan,
interruptFunc: w.interrupt,
}:
select {

case payload := <-jobChan:
result := w.worker.Process(payload)
select {
case retChan <- result:
case <-w.interruptChan:
w.interruptChan = make(chan struct{})
}
//...
}
}
//...
}
```

这里实现的方式是通过常驻的 `Goroutine` 进行实现，当改变Size之后会新增 `Worker`来执行，另一种实现方式可以使用 `chan` 来控制是否开启协程，如果缓冲区已经被写满了数据，则不能再开启新的Goroutine 进行处理 。

```
type ProcessFunc func(ctx context.Context, param interface{})

type MultiConcurrency struct {
ch chan struct{}
f  ProcessFunc
}

func NewMultiConcurrency(size int, f ProcessFunc) *MultiConcurrency {
return &MultiConcurrency{
ch: make(chan struct{}, size),
f:  f,
}
}

func (m *MultiConcurrency) Run(ctx context.Context, param interface{}) {
// 如果缓冲区已满则不进入
m.ch <- struct{}{}
```

```
go func() {
defer func() {
// 释放缓冲区
<-m.ch
if err := recover(); err != nil {
fmt.Println(err)
}
}()
m.f(ctx, param)
}()
}
```

增加测试，协程数不超过13

```
func mockFunc(ctx context.Context, param interface{}) {
fmt.Println(param)
}

func TestNewMultiConcurrency_Run(t *testing.T) {
concurrency := NewMultiConcurrency(10, mockFunc)
for i := 0; i < 1000; i++ {
concurrency.Run(context.Background(), i)
if runtime.NumGoroutine() > 13 {
fmt.Println("goroutine", runtime.NumGoroutine())
}
}
}
```

通过这种方式可以不用让内存中常驻许多正在跑的协程，但是实际上如果常驻了100个协程也仅仅会带来2kb * 100 = 200kb 的内存消耗，所以基本可以忽略不计。

写在最后

感谢你读到这里，如果想要看更多 Go及云原生的文章可以订阅我的专栏： juejin.cn/column/7321… 。