第 5 章 有关 Goroutine 无限创建的分析

本章主要从操作系统层面来分析进程、线程、协程的区别，在了解协程特点的基础上，分析是否可以无限地创建 Goroutine，以及限定 Goroutine 数量的几种办法及对比。通过了解本章的内容，开发者可以清楚如何更好地控制程序 Go 语言严重泄漏而导致的内存占用过高等问题。最后本章会基于控制 Goroutine 办法的基础上，实现协程 Worker 工作池的设计。

5.1 从操作系统分析进程、线程、协程的区别

进程、线程、协程实际上都是为并发而生，但是它们各自的模样是完全不一致的，本节会分析它们各自的特点和关系。本书不重点介绍什么是进程和线程，而是提炼进程、线程、协程的主要特点及区别，并且是基于 Linux 操作系统环境来分享进程、线程。

5.1.1 进程内存

进程是一个可执行程序在运行中而形成的一个独立的内存体，这个内存体有自己独立的地址空间，Linux 操作系统会给每个进程分配一个虚拟内存空间，其中 32 位操作系统为 4GB，64 位操作系统则会更大。进程有自己的堆空间，进程直接被操作系统调度。操作系统实则也是以进程为单位，分配系统资源，如 CPU 时间片、内存等资源，所以以此特点划分进程被称作操作系统资源分配的最小单位，如图 5.1 所示。

5.1.2 线程内存

线程也可以被称为轻量级进程(Light Weight Process，LWP)，是 CPU 调度执行的最小单位。

为什么线程会被称作最小的执行单位？这是因为线程具备的一些特征。多个线程共同“寄生”在一个进程上，这些线程都拥有各自的栈空间，但其他的内存空间都和其他线程一起共享，如图 5.2 所示。由于这个特性，使线程之间的内存关联性很大，但互相通信却很简单，堆区、全局区等数据都共享，只需要加锁机制便可以完成同步通信。这种特征同时也让线程之间关联性较大，如一个线程出问题，则会导致进程也出问题，进而也可能导致其他线程也出问题。

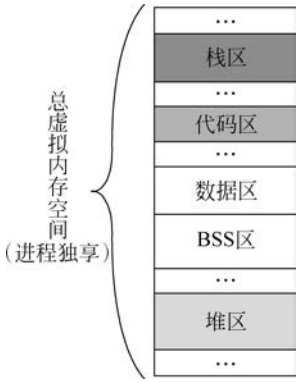

图 5.1 进程虚拟内存空间

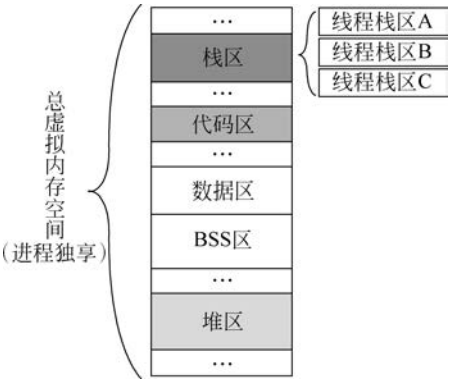

图 5.2 线程拥有独立的栈,共享进程全部其他内存资源

5.1.3 执行单元

对于 Linux 操作系统来讲,并不会去区分即将执行的单元是进程还是线程,进程和线程都是一个单独的执行单位,CPU 会一视同仁,平均分配时间片,所以开发者可以通过给一个进程提高内部线程的数量,从而增加被 CPU 分配到时间片的比例。这也是很多时候开发者发现多开一些线程就能够提高进程运行效率的原因,实则这样可以让固定的 CPU 资源能够更多地分配到自己程序上,如图 5.3 所示。

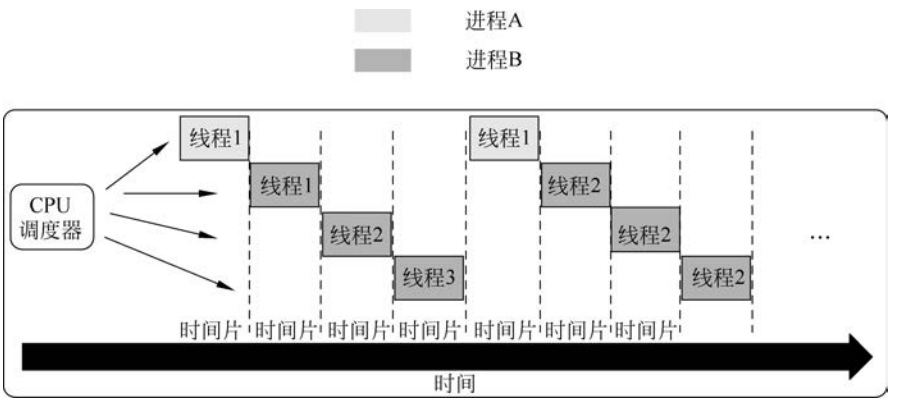

图 5.3 CPU 调度分配时间片

在图 5.3 中,进程 A 有一个线程 1,进程 B 有 3 个线程,分别为 1、2、3。通常进程 B 被分到的时间片的总和会更多,获得的 CPU 资源也就更多。

是不是线程可以无限制多呢?答案当然不是的,当 CPU 在内核态切换一个执行单元的时候,会有时间成本和性能开销,如图 5.4 所示。

切换内核栈和切换硬件上下文都会触发性能的开销,切换时会保存寄存器中的内容,将之前的执行流程状态保存,也会导致 CPU 高速缓存失效。

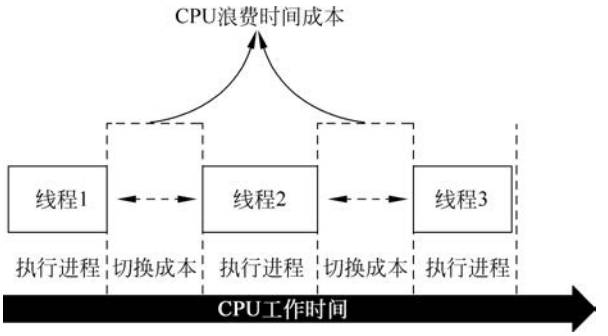

图 5.4　CPU 切换成本

这两个切换，我们没必要太深入研究，可以理解为它所带来的后果和影响是由于页表查找是一个很慢的过程，因此通常使用 Cache 来缓存常用的地址映射，这样可以加速页表查找，Cache 失效导致命中率降低，因此虚拟地址转换为物理地址就会变慢，表现出来的就是程序运行会变慢。

所以不能大量地开辟线程，因为线程执行流程越多，CPU 切换的时间成本就越大。很多编程语言就想了个解决办法，既然不能左右和优化 CPU 切换线程的开销，那么能否让 CPU 内核态不切换执行单元，而是在用户态切换执行流程。

开发者没有权限修改操作系统的内核机制，因此只能在用户态再创建一个伪执行单元，这就是协程，如图 5.5 所示。

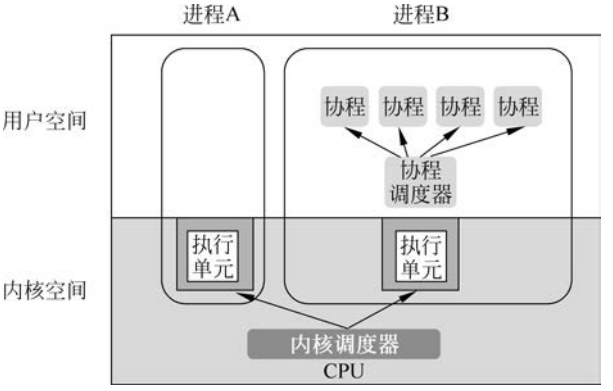

图 5.5　协程所在的空间

5.2　协程的切换成本

协程切换之所以比线程快，主要有以下两点：

(1) 协程切换完全在用户空间进行线程切换，涉及特权模式切换，需要在内核空间完成。

(2) 协程切换相比线程切换做的事情更少,线程需要有内核和用户态的切换,以及系统调用过程。

5.2.1 协程切换成本

协程切换非常简单,就是把当前协程的 CPU 寄存器的状态保存起来,然后将需要切换进来的协程的 CPU 寄存器状态加载到 CPU 寄存器上就可以了,而且完全在用户态进行,一般来讲一次协程上下文切换最多只需几十纳秒的时间。

5.2.2 线程切换成本

系统内核调度的对象是线程,因为线程是调度的基本单元(进程是资源拥有的基本单元,进程的切换需要做的事情更多,这里暂时不讨论进程切换),而线程的调度只有拥有最高权限的内核空间才可以完成,所以线程的切换涉及用户空间和内核空间的切换,也就是特权模式切换,然后需要操作系统调度模块完成线程调度,而且除了和协程基本相同的 CPU 上下文,还有线程私有的栈和寄存器等,上下文比协程多一些。

5.2.3 内存占用

进程在 32 位操作系统中占用 4GB 内存,在 64 位系统中则更多,线程跟不同的操作系统版本占用内存有所差异,查看指令如下:

```
$ulimit -s
8192
```

通过 ulimit 指令可以看到线程的大小,单位是 KB,但线程基本的量级单位为 MB,一般是 4～64MB 不等,多数维持 10MB 上下。

协程占用多少内存,下面来测试一下,这里选择的操作系统环境如下:

```
$ more /proc/cpuinfo | grep "model name"
model name  : Intel(R) Core(TM) i7-5775R CPU @ 3.30GHz
model name  : Intel(R) Core(TM) i7-5775R CPU @ 3.30GHz

(2 个 CPU )

$ grep MemTotal /proc/meminfo
MemTotal:       2017516 kB

(2GB 内存)

$ getconf LONG_BIT
```

```
64

(64 位操作系统)

 $ uname -a
Linux Ubuntu 4.15.0-91-generic #92-Ubuntu SMP Fri Feb 28 11:09:48 UTC 2020 x86_64 x86_64
x86_64 GNU/linux
```

通过下面的测试程序，来执行，代码如下：

```
//第二篇/chapter5/goroutine_size.go
package main

import (

   "time"
)

func main() {

   for i := 0; i < 200000; i++{

      go func() {

         time.Sleep(5 * time.Second)

      }()

   }

   time.Sleep(10 * time.Second)
}
```

程序运行前：

```
top - 00:16:24 up 7:08, 1 user, load average: 0.08, 0.03, 0.01
任务: 288 total, 1 running, 218 sleeping, 0 stopped, 0 zombie
%Cpu0 : 0.0 us, 0.0 sy, 0.0 ni,100.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 0.3 us, 0.3 sy, 0.0 ni, 99.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 2017516 total, 593836 free, 1163524 used, 260156 buff/cache
KiB Swap: 969960 total, 574184 free, 395776 used. 679520 avail Mem
free 的 mem 为 1163524,
```

程序运行中：

```
top - 00:17:12 up 7:09, 1 user, load average: 0.04, 0.02, 0.00
任务: 290 total, 1 running, 220 sleeping, 0 stopped, 0 zombie
%Cpu0 : 4.0 us, 1.0 sy, 0.0 ni, 95.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu1 : 8.8 us, 1.4 sy, 0.0 ni, 89.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 2017516 total, 89048 free, 1675844 used, 252624 buff/cache
KiB Swap: 969960 total, 563688 free, 406272 used. 168812 avail Mem
free 的 mem 为 1675844,
```

20 万个协程占用了约 500 000KB,平均一个协程占用约 2.5KB。既然 Go 的协程切换成本如此小,占用内存也那么小,是否可以无限开辟呢?

5.3 Go 是否可以无限创建,如何限定数量

5.2 节分析了 Go 协程的切换开销成本和内存占用,协程都非常明显地具备优势,面对如此强大的诱惑,开发者是否真的可以让 Go 协程的数量泛滥而不去特意控制呢? 本节将针对此话题继续讲解。

5.3.1 不控制 Goroutine 数量引发的问题

Goroutine 具备如下两个特点:体积轻量、优质的 GMP 调度。那么 Goroutine 是否可以无限开辟呢? 如果做一个服务器或者一些高业务的场景,能否随意地开辟 Goroutine 并且任其数量泛滥而不去主动回收 Goroutine 呢? 能否通过强大的 GC 和优质的调度算法来支撑呢?

可以先看如下一段代码:

```
//第二篇/chapter5/goroutine_max.go
package main

import (
    "fmt"
    "math"
    "runtime"
)

func main() {
    //模拟用户需求业务的数量
    task_cnt := math.MaxInt64

    for i := 0; i < task_cnt; i++{
        go func(i int) {
            //... do some busi...
```

```
            fmt.Println("go func ", i, " goroutine count = ", runtime.NumGoroutine())
        }(i)
    }
}
```

结果如下：

```
...
...
go func 73362 goroutine count = 70074
go func 73496 goroutine count = 70090
go func 1500220 goroutine count = 1132354
go func 1500513 goroutine count = 1132353
go func 1500227 goroutine count = 1132352
go func 74103 goroutine count = 70088
go func 14161 goroutine count = 14116
go func 457524 goroutine count = 503289
go func 1500307 goroutine count = 1132348
go func 1500091 goroutine count = 1132348
go func 1500519 goroutine count = 1132346
go func 1500092 goroutine count = 1132345
go func 1500886 goroutine count = 1132344
go func 1500523 goroutine count = 1132343
go func 1499968 goroutine count = 1132352
go func 1500095 goroutine count = 1132341
go func 33177 goroutine count = 31738
go func 1500419 goroutine count = 1132339
go func 1500759 goroutine count = 1132344
go func 1500531 goroutine count = 1132337
go func 73497 goroutine count = 70087
go func 1500760 goroutine count = 1132335
go func 1500420 goroutine count = 1132334
go func 1500890 goroutine count = 1132337
go func 1500535 goroutine count = 1132332
go func 161872 goroutine count = 151547
go func 456085 goroutine count = 503280
go func 456372 goroutine count = 503279
go func 73767 goroutine count = 70099
go func 1500424 goroutine count = 1132327
go func 1500538 goroutine count = 1132326
go func 160928 goroutine count = 151546
go func 73768 goroutine count = 70102
go func 1500894 goroutine count = 1132323
panic: too many concurrent operations on a single file or socket (max 1048575)
```

```
goroutine 1501390 [running]:
internal/poll.(*fdMutex).rwlock(0xc00001e120, 0x7600000000, 0xc000000076)
    /usr/local/go/src/internal/poll/fd_mutex.go:147 +0x13f
internal/poll.(*FD).writeLock(...)
    /usr/local/go/src/internal/poll/fd_mutex.go:239
internal/poll.(*FD).Write(0xc00001e120, 0xc1343fd7c0, 0x2d, 0x40, 0x0, 0x0, 0x0)
    /usr/local/go/src/internal/poll/fd_UNIX.go:255 +0x5d
os.(*File).write(...)
    /usr/local/go/src/os/file_UNIX.go:280
os.(*File).Write(0xc00000e018, 0xc1343fd7c0, 0x2d, 0x40, 0x2b, 0xc0023e8f28, 0x10090cb)
    /usr/local/go/src/os/file.go:153 +0x77
fmt.Fprintln(0x10ecac0, 0xc00000e018, 0xc0023e8f88, 0x4, 0x4, 0x2b, 0x0, 0x0)
    /usr/local/go/src/fmt/print.go:265 +0x8b
fmt.Println(...)
    /usr/local/go/src/fmt/print.go:274
main.main.func1(0x16e85a)
    /Users/Aceld/Nutstore Files/Golang/Code/第二篇/chapter11/goroutine_max.go:18 +0x10c
created by main.main
    /Users/Aceld/Nutstore Files/Golang/Code/第二篇/chapter11/goroutine_max.go:15 +0x43
panic: too many concurrent operations on a single file or socket (max 1048575)

...
```

最后被操作系统以 kill 信号或因资源紧缺遭遇 panic 错误退出，强制终结该进程。

所以，迅速地开辟 Goroutine 且不控制并发 Goroutine 的数量，会在短时间内占据操作系统的资源（CPU、内存、文件描述符等）。

在执行上述程序的过程中实际发生了三个灾难过程，即 CPU 使用率浮动上涨、内存占用不断上涨和主进程崩溃（被杀掉了）。

这些资源实际上是所有用户态程序共享的资源，所以大批的 Goroutine 最终引发的灾难不仅是自身，还会关联其他运行的程序。

因此在编写逻辑业务的时候，限制 Goroutine 是必须重视的问题。

5.3.2 一些简单方法控制 Goroutine 的数量

方法一，只用有 buffer 的 channel 来限制，代码如下：

```
//第二篇/chapter5/limit_goroutine_1.go
package main

import (
   "fmt"
   "math"
```

```
    "runtime"
)

func busi(ch chan bool, i int) {

    fmt.Println("go func ", i, " goroutine count = ", runtime.NumGoroutine())
    <- ch
}

func main() {
    //模拟用户需求业务的数量
    task_cnt := math.MaxInt64
    //task_cnt := 10

    ch := make(chan bool, 3)

    for i := 0; i < task_cnt; i++{

        ch <- true

        go busi(ch, i)
    }

}
```

结果如下：

```
...
go func 352277 goroutine count = 4
go func 352278 goroutine count = 4
go func 352279 goroutine count = 4
go func 352280 goroutine count = 4
go func 352281 goroutine count = 4
go func 352282 goroutine count = 4
go func 352283 goroutine count = 4
go func 352284 goroutine count = 4
go func 352285 goroutine count = 4
go func 352286 goroutine count = 4
go func 352287 goroutine count = 4
go func 352288 goroutine count = 4
go func 352289 goroutine count = 4
go func 352290 goroutine count = 4
go func 352291 goroutine count = 4
go func 352292 goroutine count = 4
```

```
go  func  352293  goroutine  count  =  4
go  func  352294  goroutine  count  =  4
go  func  352295  goroutine  count  =  4
go  func  352296  goroutine  count  =  4
go  func  352297  goroutine  count  =  4
go  func  352298  goroutine  count  =  4
go  func  352299  goroutine  count  =  4
go  func  352300  goroutine  count  =  4
go  func  352301  goroutine  count  =  4
go  func  352302  goroutine  count  =  4
...
```

从结果看，程序并没有出现崩溃的现象，而是按部就班地执行，并且 Go 的数量控制在 3[①]，从数字来看，是不是在运行的 Goroutine 有几十万个呢？下面用一张图来表示上述代码的 Go 数量控制的结构关系，如图 5.6 所示。

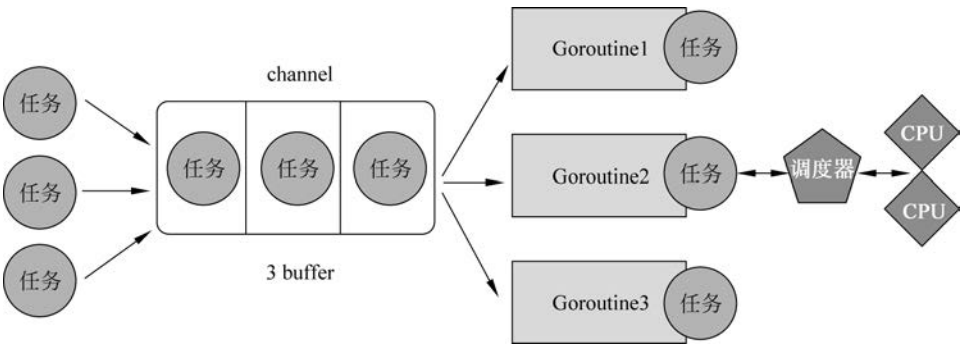

图 5.6　channel 限定 Goroutine 的数量

这里用了 buffer 为 3 的 channel，在写的过程中，实际上限制了速度，代码如下：

```
for i := 0; i < go_cnt; i++{   //循环速度

      ch <- true

      go busi(ch, i)
}
```

for 循环的速度，因为这个速度决定了 Go 的创建速度，而 Go 的结束速度取决于 busi() 函数的执行速度。这样实际上就能够保证同一时间内运行的 Goroutine 的数量与 buffer 的数量一致，从而达到限定的效果。

但是这段代码有一个小问题，就是如果把 go_cnt 的数量变得小一些，会出现输出的结

① 结果中 Goroutine 的数量为 4 的原因是因为还有一个 Main Goroutine。

果不正确，代码如下：

```
//第二篇/chapter5/limit_goroutine_1_wrong.go
package main

import (
    "fmt"
    //"math"
    "runtime"
)

func busi(ch chan bool, i int) {

    fmt.Println("go func ", i, " goroutine count = ", runtime.NumGoroutine())
    <- ch
}

func main() {
    //模拟用户需求业务的数量
    //task_cnt := math.MaxInt64
    task_cnt := 10

    ch := make(chan bool, 3)

    for i := 0; i < task_cnt; i++{

        ch <- true

        go busi(ch, i)
    }

}
```

结果如下：

```
go func 2 goroutine count = 4
go func 3 goroutine count = 4
go func 4 goroutine count = 4
go func 5 goroutine count = 4
go func 6 goroutine count = 4
go func 1 goroutine count = 4
go func 8 goroutine count = 4
```

因为 main 将全部的 Go 开辟完之后，就立刻退出进程了，所以想让全部的 Go 都执行，需要在 main 的最后进行阻塞操作。

方法二，只使用 sync 同步机制。如果不用 channel 来限定，则可以通过 sync 的同步机制来对 Go 的数量进行限定，代码如下：

```
//第二篇/chapter5/limit_goroutine_2.go
package main

import (
   "fmt"
   "math"
   "sync"
   "runtime"
)

var wg = sync.WaitGroup{}

func busi(i int) {

   fmt.Println("go func ", i, " goroutine count = ", runtime.NumGoroutine())
   wg.Done()
}

func main() {
   //模拟用户需求业务的数量
   task_cnt := math.MaxInt64


   for i := 0; i < task_cnt; i++{
    wg.Add(1)
      go busi(i)
   }

   wg.Wait()
}
```

从运行效果来看，单纯地使用 sync 依然无法控制 Goroutine 的数量，因为最终程序的结果依然是崩溃，运行程序一段时间，结果如下：

```
...
go  func  7562   goroutine  count   =   7582
go  func  24819  goroutine  count   =   17985
go  func  7685   goroutine  count   =   7582
go  func  24701  goroutine  count   =   17984
go  func  7563   goroutine  count   =   7582
go  func  24821  goroutine  count   =   17983
```

```
go  func  24822  goroutine  count  =  17983
go  func  7686   goroutine  count  =  7582
go  func  24703  goroutine  count  =  17982
go  func  7564   goroutine  count  =  7582
go  func  24824  goroutine  count  =  17981
go  func  7687   goroutine  count  =  7582
go  func  24705  goroutine  count  =  17980
go  func  24706  goroutine  count  =  17980
go  func  24707  goroutine  count  =  17979
go  func  7688   goroutine  count  =  7582
go  func  24826  goroutine  count  =  17978
go  func  7566   goroutine  count  =  7582
go  func  24709  goroutine  count  =  17977
go  func  7689   goroutine  count  =  7582
go  func  24828  goroutine  count  =  17976
go  func  24829  goroutine  count  =  17976
go  func  7567   goroutine  count  =  7582
go  func  24711  goroutine  count  =  17975
//操作系统停止响应
```

上述虽然可以达到同步效果，但是如果耗费的速度跟不上 Go 生产的速度，则最终还是会导致 Go 的数量泛滥而造成系统资源被占满，最后程序只能以崩溃收场。

方法三，channel 与 sync 同步组合方式。了解方法二的问题之后，可以让生产方和消耗方的速度达成一致，这里就需要用一个 channel 来保证二者速度一致，从而可以达到限制 Goroutine 的数量，代码如下：

```
//第二篇/chapter5/limit_goroutine_3.go
package main

import (
   "fmt"
   "math"
   "sync"
   "runtime"
)

var wg = sync.WaitGroup{}

func busi(ch chan bool, i int) {

   fmt.Println("go func ", i, " goroutine count = ", runtime.NumGoroutine())

   <- ch
```

```
    wg.Done()
}

func main() {
    //模拟用户需求 Go 业务的数量
    task_cnt := math.MaxInt64

    ch := make(chan bool, 3)

    for i := 0; i < task_cnt; i++{
        wg.Add(1)

        ch <- true

        go busi(ch, i)
    }

     wg.Wait()
}
```

结果如下：

```
...
go func 228851 goroutine count = 4
go func 228852 goroutine count = 4
go func 228853 goroutine count = 4
go func 228854 goroutine count = 4
go func 228855 goroutine count = 4
go func 228856 goroutine count = 4
go func 228857 goroutine count = 4
go func 228858 goroutine count = 4
go func 228859 goroutine count = 4
go func 228860 goroutine count = 4
go func 228861 goroutine count = 4
go func 228862 goroutine count = 4
go func 228863 goroutine count = 4
go func 228864 goroutine count = 4
go func 228865 goroutine count = 4
go func 228866 goroutine count = 4
go func 228867 goroutine count = 4
...
```

这样程序就不会再造成资源占满而崩溃的问题了，而且运行 Go 的数量控制住了，即在 buffer 为 3 的这个范围内。

方法四，利用无缓冲 channel 与任务发送/执行分离方式。发送和执行分离方式的代码如下：

```
//第二篇/chapter5/limit_goroutine_4.go
package main

import (
   "fmt"
   "math"
   "sync"
   "runtime"
)

var wg = sync.WaitGroup{}

func busi(ch chan int) {

   for t := range ch {
      fmt.Println("go task = ", t, ", goroutine count = ", runtime.NumGoroutine())
      wg.Done()
   }
}

func sendTask(task int, ch chan int) {
   wg.Add(1)
   ch <- task
}

func main() {

   ch := make(chan int)                          //无 buffer channel

goCnt := 3                                       //启动 Goroutine 的数量

   for i := 0; i < goCnt; i++{
      //启动 go
      go busi(ch)
   }

taskCnt := math.MaxInt64                         //模拟用户需求业务的数量

   for t := 0; t < taskCnt; t++{
      //发送任务
      sendTask(t, ch)
   }

    wg.Wait()
}
```

结果如下：

```
...
go task = 130069 , goroutine count = 4
go task = 130070 , goroutine count = 4
go task = 130071 , goroutine count = 4
go task = 130072 , goroutine count = 4
go task = 130073 , goroutine count = 4
go task = 130074 , goroutine count = 4
go task = 130075 , goroutine count = 4
go task = 130076 , goroutine count = 4
go task = 130077 , goroutine count = 4
go task = 130078 , goroutine count = 4
go task = 130079 , goroutine count = 4
go task = 130080 , goroutine count = 4
go task = 130081 , goroutine count = 4
go task = 130082 , goroutine count = 4
go task = 130083 , goroutine count = 4
go task = 130084 , goroutine count = 4
go task = 130085 , goroutine count = 4
go task = 130086 , goroutine count = 4
go task = 130087 , goroutine count = 4
go task = 130088 , goroutine count = 4
go task = 130089 , goroutine count = 4
go task = 130090 , goroutine count = 4
go task = 130091 , goroutine count = 4
go task = 130092 , goroutine count = 4
go task = 130093 , goroutine count = 4
...
```

执行流程大致如下，这里实际上是将任务的发送和执行做了业务上的分离。使消息出去，输入 SendTask 的频率可设置，并且执行 Goroutine 的数量也可设置。也就是既可控制输入(生产)，又可控制输出(消费)，使可控更加灵活。这也是很多 Go 框架的 Worker 工作池的最初设计理念，如图 5.7 所示。

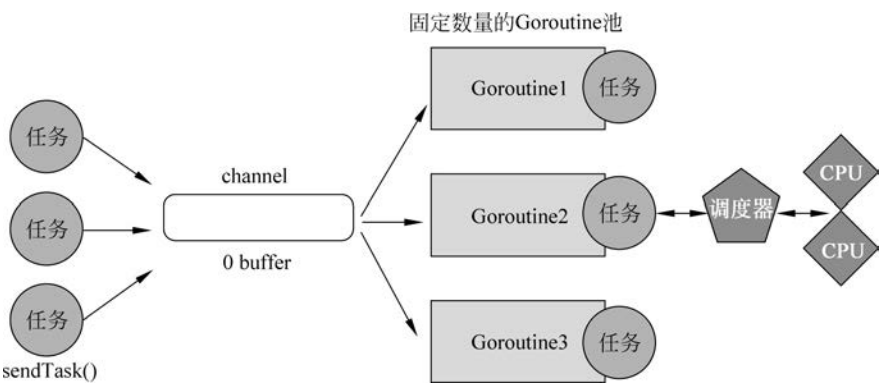

图 5.7 发送/执行分离方式限定 Goroutine 的数量

以上几种方法便是目前有关限定 Goroutine 的基础设计思路。

5.4　动态保活 Worker 工作池设计

至此，了解了如何限定 Goroutine 的数量，接下来有必要了解一下如何创建一个 Worker 工作池来实现一个通用的限定 Goroutine 的解决办法。这里的 Worker 的概念实则是每个正在被调度且执行业务任务的 Goroutine。本节将简单地设计一个 Worker 工作池。

5.4.1　如何确定一个 Goroutine 已经死亡

实际上，Go 语言并没有给开发者暴露如何知道一个 Goroutine 是否存在接口，如果要证明一个 Go 是否存在，则可以在子 Goroutine 的业务中，定期地写一个 Keeplive 的 Channel，然后由主 Goroutine 来发现当前子 Go 的状态。Go 语言在 Go 和 Go 之间没有像进程和线程那样有强烈的父子、兄弟等关系，每个 Go 实际上对于调度器都是一个独立的平等的执行流程。

注意　如果要监控子线程、子进程的死亡状态，就没有这么简单了，这里也要感谢 Go 的调度器给开发者提供的方便，既然用 Go，就要基于 Go 的调度器实现该模式。

那么，如何做到得知一个 Goroutine 已经死亡了呢？子 Goroutine 和主 Goroutine 需要做如下动作。

1. 子 Goroutine

可以通过给一个被监控的 Goroutine 添加一个 defer，当 recover() 捕获到当前 Goroutine 的异常状态时通过 channel 给主 Goroutine 发送一个死亡信号。

2. 主 Goroutine

在主 Goroutine 上，从这个 channel 读取内容，当读到内容时，就重启这个子 Goroutine，当然主 Goroutine 需要记录子 Goroutine 的 ID，这样就可以有针对性地启动了。

5.4.2　Worker 工作池的设计

这里以一个工作池的场景来对上述方式进行实现，WorkerManager 作为主 Goroutine，Worker 作为子 Goroutine。

WorkerManager 的实现代码如下：

```
//第二篇/chapter5/worker_pool.go

/*
   WorkerManager
*/

type WorkerManager struct {
```

```
    //用来监控 Worker 是否已经死亡的缓冲 channel
    workerChan chan * worker
    //一共要监控的 Worker 的数量
    nWorkers int
}

//创建一个 WorkerManager 对象
func NewWorkerManager(nworkers int) * WorkerManager {
    return &WorkerManager{
        nWorkers:nworkers,
        workerChan: make(chan * worker, nworkers),
    }
}

//启动 worker 池,并为每个 Worker 分配一个 ID,让每个 Worker 进行工作
func (wm * WorkerManager)StartWorkerPool() {
    //开启一定数量的 Worker
    for i := 0; i < wm.nWorkers; i++{
        i := i
        wk := &worker{id: i}
        go wk.work(wm.workerChan)
    }

    //启动保活监控
    wm.KeepLiveWorkers()
}

//保活监控 Workers
func (wm * WorkerManager) KeepLiveWorkers() {
    //如果有 Worker 已经死亡,则 workChan 会得到具体死亡的 Worker,然后输出异常,最后重启
    for wk := range wm.workerChan {
        //log the error
        fmt.Printf("Worker %d stopped with err: [ %v] \n", wk.id, wk.err)
        //reset err
        wk.err = nil
        //当前这个 wk 已经死亡了,需要重新启动它的业务
        go wk.work(wm.workerChan)
    }
}
```

Worker 的实现代码如下：

```
//第二篇/chapter5/worker_pool.go

/*
```

```
  Worker
*/

type worker struct {
  id int
  err error
}

func (wk *worker) work(workerChan chan<- *worker) (err error) {
  //任何 Goroutine 只要异常退出或者正常退出都会调用 defer 函数,所以在 defer 中向
  //WorkerManager 的 WorkChan 发送通知
  defer func() {
    //捕获异常信息,防止 panic 直接退出
    if r := recover(); r != nil {
      if err, ok := r.(error); ok {
        wk.err = err
      } else {
        wk.err = fmt.Errorf("Panic happened with [%v]", r)
      }
    } else {
      wk.err = err
    }

   //通知主 Goroutine,当前子 Goroutine 已经死亡
    workerChan <- wk
  }()

  //do something
  fmt.Println("Start Worker...ID = ", wk.id)

  //每个 Worker 睡眠一定时间之后,panic 退出或者 Goexit()退出
  for i := 0; i < 5; i++{
    time.Sleep(time.Second * 1)
  }

  panic("worker panic..")
  //runtime.Goexit()

  return err
}
```

5.4.3　测试 Worker 工作池

main()函数的代码如下：

```
//第二篇/chapter5/worker_pool.go

/*
  main
*/
func main() {
  wm := NewWorkerManager(10)

  wm.StartWorkerPool()
}
```

结果如下：

```
$ go run workmanager.go
Start  Worker...ID   =   2
Start  Worker...ID   =   1
Start  Worker...ID   =   3
Start  Worker...ID   =   4
Start  Worker...ID   =   7
Start  Worker...ID   =   6
Start  Worker...ID   =   8W
Start  Worker...ID   =   9
Start  Worker...ID   =   5
Start  Worker...ID   =   0
Worker  9  stopped with err: [Panic happened with [worker panic..]]
Worker  1  stopped with err: [Panic happened with [worker panic..]]
Worker  0  stopped with err: [Panic happened with [worker panic..]]
Start  Worker...ID   =   9
Start  Worker...ID   =   1
Worker  2  stopped with err: [Panic happened with [worker panic..]]
Worker  5  stopped with err: [Panic happened with [worker panic..]]
Worker  4  stopped with err: [Panic happened with [worker panic..]]
Start  Worker...ID   =   0
Start  Worker...ID   =   2
Start  Worker...ID   =   4
Start  Worker...ID   =   5
Worker  7  stopped with err: [Panic happened with [worker panic..]]
Worker  8  stopped with err: [Panic happened with [worker panic..]]
Worker  6  stopped with err: [Panic happened with [worker panic..]]
Worker  3  stopped with err: [Panic happened with [worker panic..]]
Start  Worker...ID   =   3
Start  Worker...ID   =   6
Start  Worker...ID   =   8
Start  Worker...ID   =   7
...
...
```

从结果可以看出，无论子 Goroutine 是因为 panic()异常退出，还是因为 Goexit()退出，都会被主 Goroutine 监听到并且重启。这样就能够起到保活的功能了，但如果线程死亡，则又该如何保证呢？Go 开发实际上是基于 Go 的调度器来开发的，进程和线程级别的死亡会导致调度器死亡，此种情况全部基础框架都将会塌陷，所以就要依赖线程和进程的保活机制了，这将不再涉及 Go 设计保活机制的范畴。读者可以关注一些有关进程和线程的保活机制方案。

5.5　小结

本章主要介绍了有关 Goroutine 的限制数量问题，分别从进程、线程及协程的区别讲起，最终得到协程的优势是占用内存空间小且切换成本也小等优点，但是如果任意地开辟协程也会带来一定的系统灾难。通过案例等分析，提供了几种可以限定协程数量的方法，这里主要是针对 Goroutine 的限定方法，当然限定协程未必是一定要做的，如果开发者已经开发好了程序，并且能够从逻辑上做到合理地退出协程，或者显示调用 runtime.Goexit()函数来终止当前协程，则可以不做限定协程数量的处理，因为这样可以避免协程的数量泄漏。

Go 语言虽然提供了良好的 GC 垃圾回收机制，但是对于 Goroutine 的数量控制，GC 并不能做得非常智能，这一部分的回收控制还需要开发者自己从代码上进行优化和控制，如果开发者开辟的协程过多，则说明是一种内存泄漏的表现形式。