《Go语言实战》笔记(十五) | Go并发示例-Runner

原创 2017-04-17  飞雪无情

《Go语言实战》读书笔记，未完待续，欢迎扫码关注公众号flysnow\_org，第一时间看后续笔记。觉得有帮助的话，顺手分享到朋友圈吧，感谢支持。

这篇通过一个例子，演示使用通道来监控程序的执行时间，生命周期，甚至终止程序等。我们这个程序叫runner，我们可以称之为执行者，它可以在后台执行任何任务，而且我们还可以控制这个执行者，比如强制终止它等。

现在开始吧，运用我们前面十几篇连载的知识，来构建我们的Runner，使用一个结构体类型就可以。

*//一个执行者，可以执行任何任务，但是这些任务是限制完成的，//该执行者可以通过发送终止信号终止它*type Runner struct {

   tasks []func(int) *//要执行的任务*

   complete chan error *//用于通知任务全部完成*

   timeout <-chan time.Time *//这些任务在多久内完成*

   interrupt chan os.Signal *//可以控制强制终止的信号*}

示例中，我们定义了一个结构体类型Runner，这个Runner包含了要执行哪些任务tasks,然后使用complete通知任务是否全部完成，不过这个执行者是有时间限制的，这就是timeout，如果在限定的时间内没有完成，就会接收到超时的通知，如果完成了就会接收到完成的通知。注意这里的timeout是单向通道，只能接收。

complete定义为error类型的通道，是为了当执行任务出现问题时返回错误的原因，如果没有出现错误，返回的是nil。

此外，我们还定义了一个中断的信号，让我们可以随时的终止执行者。

有了结构体，我们接着再定义一个工厂函数New,用于返回我们需要的Runner。

func New(tm time.Duration) \*Runner {  
    return &Runner{

       complete:make(chan error),

       timeout:time.After(tm),

       interrupt:make(chan os.Signal,1),

   }

}

这个New函数非常简洁，可以帮我们很快的初始化一个Runnner，它只有一个参数，用来设置这个执行者的超时时间。这个超时时间被我们传递给了time.After函数，这个函数可以在tm时间后，会同伙一个time.Time类型的只能接收的单向通道，来告诉我们已经到时间了。

complete是一个无缓冲通道，也就是同步通道，因为我们要使用它来控制我们整个程序是否终止，所以它必须是同步通道，要让main goroutine等待，一致要任务完成或者被强制终止。

interrupt是一个有缓冲的通道，这样做是因为，我们可以至少接收到一个操作系统的中断信息，这样Go runtime在发送这个信号的时候不会被阻塞，如果是无缓冲的通道就会阻塞了。

系统信号是什么意思呢，比如我们在程序执行的时候按下Ctrl + C，这就是一个中断的信号，告诉程序可以强制终止了。

我们这里初始化了结构体的三个字段，而执行的任务tasks没有初始化，默认就是零值nil，因为它是一个切片。但是我们的执行者Runner不能没有任务啊，既然初始化Runner的时候没有，那我们就定义一个方法，通过方法给执行者添加需要执行的任务。

*//将需要执行的任务，添加到Runner里*func (r \*Runner) Add(tasks ...func(int)){

   r.tasks = append(r.tasks,tasks...)

}

这个没有太多可以说明的，r.tasks就是一个切片，来存储需要执行的任务。通过内置的append函数就可以追加任务了。这里使用了可变参数，可以灵活的添加一个，甚至同时多个任务，比较方便。

到了这里我们需要的执行者Runner,如何添加任务，如何获取一个执行者，都有了，下面就开始执行者如何运行任务？如何在运行的时候强制中断任务？在这些处理之前，我们先来定义两个我们的两个错误变量，以便在接下来的代码实例中使用。

var ErrTimeOut = errors.New("执行者执行超时")

var ErrInterrupt = errors.New("执行者被中断")

两种错误类型，一个表示因为超时错误，一个表示因为被中断错误。下面我们就看看如何执行一个个任务。

*//执行任务，执行的过程中接收到中断信号时，返回中断错误//如果任务全部执行完，还没有接收到中断信号，则返回nil*func (r \*Runner) run() error {  
    for id, task := range r.tasks {  
        if r.isInterrupt() {  
               return ErrInterrupt

       }

       task(id)

    }      
    return nil  
}  
      
*//检查是否接收到了中断信号*func (r \*Runner) isInterrupt() bool {  
    select {  
    case <-r.interrupt:

       signal.Stop(r.interrupt)          
        return true

   default:         
         return false

   }

}

新增的run方法也很简单，会使用for循环，不停的运行任务，在运行的每个任务之前，都会检测是否收到了中断信号，如果没有收到，则继续执行，一直到执行完毕，返回nil；如果收到了中断信号，则直接返回中断错误类型，任务执行终止。

这里注意isInterrupt函数，它在实现的时候，使用了基于select的多路复用，select和switch很像，只不过它的每个case都是一个通信操作。那么到底选择哪个case块执行呢？原则就是哪个case的通信操作可以执行就执行哪个，如果同时有多个可以执行的case，那么就随机选择一个执行。

针对我们方法中，如果r.interrupt中接受不到值，就会执行default语句块，返回false，一旦r.interrupt中可以接收值，就会通知Go Runtime停止接收中断信号，然后返回true。

这里如果没有default的话，select是会阻塞的，直到r.interrupt可以接收值为止，因为我们例子中的逻辑要求不能阻塞，所以我们使用了default。

好了，基础工作都做好了，现在开始执行我们所有的任务，并且时刻监视着任务的完成，执行事件的超时。

*//开始执行所有任务，并且监视通道事件*func (r \*Runner) Start() error {  
    *//希望接收哪些系统信号*

   signal.Notify(r.interrupt, os.Interrupt)      
    go func() {

       r.complete <- r.run()

   }()      
      
    select {  
        case err := <-r.complete:  
                return err      
        case <-r.timeout:          
                return ErrTimeOut

   }

}

signal.Notify(r.interrupt, os.Interrupt)，这个是表示，如果有系统中断的信号，发给r.interrupt即可。

任务的执行，这里开启了一个groutine，然后调用run方法，结果发送给通道r.complete。最后就是使用一个select多路复用，哪个通道可以操作，就返回哪个。

到了这时候，只有两种情况了，要么任务完成；要么到时间了，任务执行超时。从我们前面的代码看，任务完成又分两种情况，一种是没有执行完，但是收到了中断信号，中断了，这时返回中断错误；一种是顺利执行完成，这时返回nil。

现在把这些代码汇总一下，容易统一理解一下，所有代码如下

package common  
import (  
    "errors"

   "os"

   "os/signal"

   "time"  
)  
var ErrTimeOut = errors.New("执行者执行超时")  
var ErrInterrupt = errors.New("执行者被中断")  
 *//一个执行者，可以执行任何任务，但是这些任务是限制完成的，//该执行者可以通过发送终止信号终止它*type Runner struct {

   tasks     []func(int)      *//要执行的任务*

   complete  chan error       *//用于通知任务全部完成*

   timeout   <-chan time.Time *//这些任务在多久内完成*

   interrupt chan os.Signal   *//可以控制强制终止的信号*}  
func New(tm time.Duration) \*Runner {  
    return &Runner{

       complete:  make(chan error),

       timeout:   time.After(tm),

       interrupt: make(chan os.Signal, 1),

   }

}  
*//将需要执行的任务，添加到Runner里*func (r \*Runner) Add(tasks ...func(int)) {

   r.tasks = append(r.tasks, tasks...)

}  
*//执行任务，执行的过程中接收到中断信号时，返回中断错误//如果任务全部执行完，还没有接收到中断信号，则返回nil*func (r \*Runner) run() error {  
    for id, task := range r.tasks {  
            if r.isInterrupt() {  
                        return ErrInterrupt

       }

       task(id)

    }  
     return nil  
}  
*//检查是否接收到了中断信号*func (r \*Runner) isInterrupt() bool {  
    select {  
    case <-r.interrupt:

       signal.Stop(r.interrupt)          
        return true

   default:          
        return false

   }

}  
*//开始执行所有任务，并且监视通道事件*func (r \*Runner) Start() error {      
 *//希望接收哪些系统信号*

   signal.Notify(r.interrupt, os.Interrupt)  
    go func() {

       r.complete <- r.run()

   }()      
    select {  
        case err := <-r.complete:   
               return err     
        case <-r.timeout:         
               return ErrTimeOut

   }

}

这个common包里的Runner我们已经开发完了，现在我们写个例子试试它。

package main  
import (  
    "flysnow.org/hello/common"

   "log"

   "time"

   "os"  
)  
func main() {

   log.Println("...开始执行任务...")

   timeout := 3 \* time.Second

   r := common.New(timeout)

   r.Add(createTask(), createTask(), createTask())  
    if err:=r.Start();err!=nil{  
            switch err {   
            case common.ErrTimeOut:

                   log.Println(err)

                   os.Exit(1)          
            case common.ErrInterrupt:

                  log.Println(err)

                  os.Exit(2)

       }

   }

   log.Println("...任务执行结束...")

}  
func createTask() func(int) {  
    return func(id int) {

       log.Printf("正在执行任务%d", id)

       time.Sleep(time.Duration(id)\* time.Second)

   }

}

例子非常简单，定义任务超时时间为3秒，添加3个生成的任务，每个任务都是打印一个正在执行哪个任务，然后休眠一段时间。

调用r.Start()开始执行任务，如果一切都正常的话，返回nil，然后打印出...任务执行结束...，不过我们例子中，因为超时时间和任务的设定，结果是执行超时的。

2017/04/15 22:17:55 ...开始执行任务...  
2017/04/15 22:17:55 正在执行任务0  
2017/04/15 22:17:55 正在执行任务1  
2017/04/15 22:17:56 正在执行任务2  
2017/04/15 22:17:58 执行者执行超时

如果我们把超时时间改为4秒或者更多，就会打印...任务执行结束...。这里我们还可以测试另外一种系统中断情况，在终端里运行程序后，快速不停的按Ctrl + C，就可以看到执行者被中断的打印输出信息了。

到这里，这篇文章已经要收尾了，这个例子中，我们演示使用通道通信、同步等待，监控程序等。

此外这个执行者也是一个很不错的模式，比如我们写好之后，交给定时任务去执行即可，比如cron，这个模式我们还可以扩展，更高效率的并发，更多灵活的控制程序的生命周期，更高效的监控等，这个大家自己可以试试，基于自己的需求修改就可以了。

《Go语言实战》读书笔记，未完待续，欢迎扫码关注公众号flysnow\_org，第一时间看后续笔记。觉得有帮助的话，顺手分享到朋友圈吧，感谢支持。