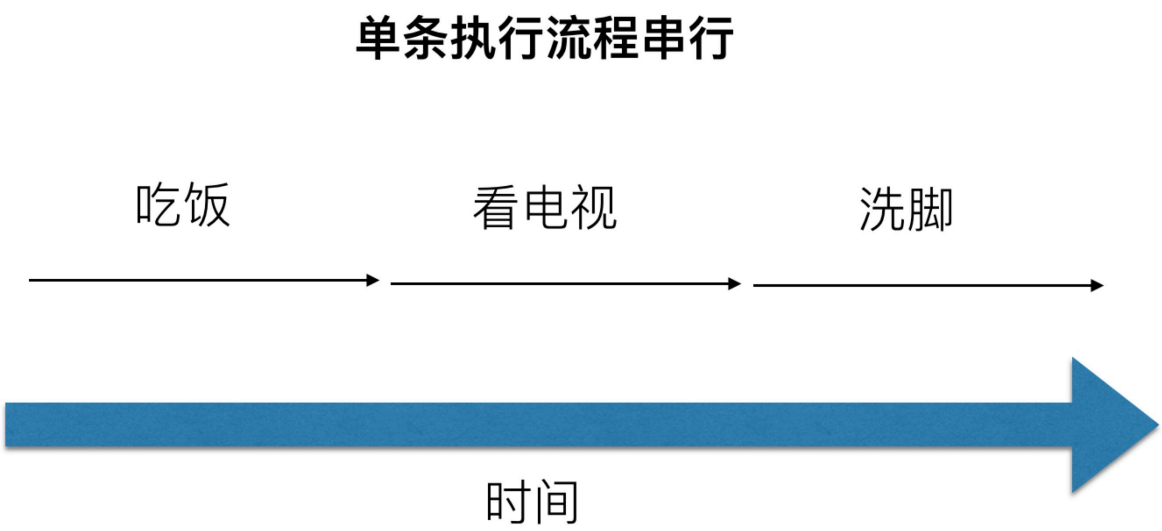
《Go的并发协程池设计》

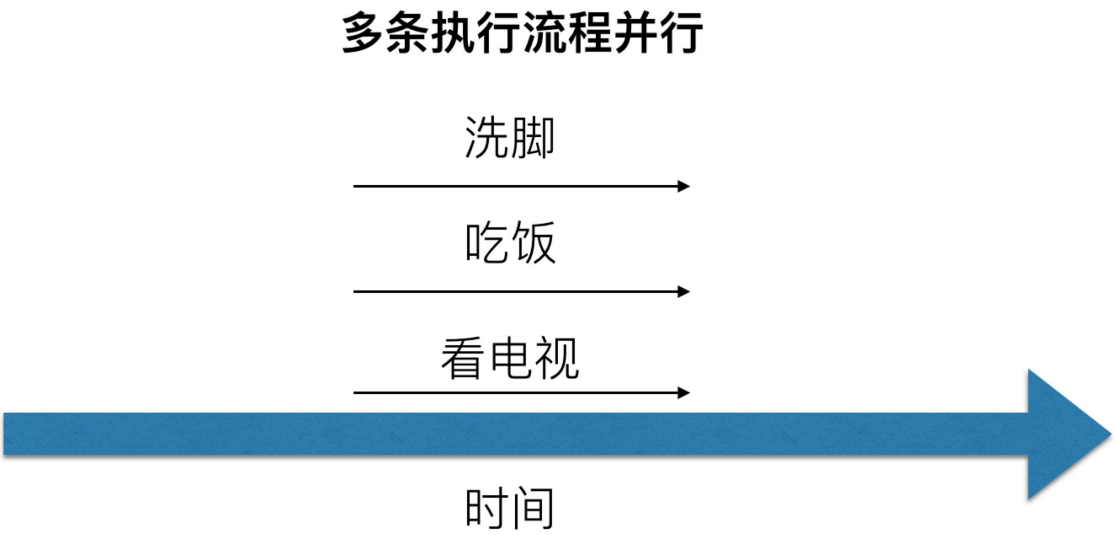
教程制作： 无崖子(刘丹冰)

教程简介： 本教程主要针对具有一定编程基础的学员，懂得基本的编程语法。

> 使用Go语言实现并发的协程调度池阉割版，本文主要介绍协程池的基本设计思路，目的为深入浅出快速了解协程池工作原理，与真实的企业协程池还有很大差距，本文仅供学习参考。

一、何为并发，Go又是如何实现并发?





> ### 并行的好处：
>
> 1. 同一时刻可以处理多个事务
> 2. 更加节省时间，效率更高

具有并行处理能力的程序我们称之为“并发程序”

并发程序的处理能力优势体现在哪里？

二、Go语言如何实现并发？

```
package main

import "fmt"
import "time"

func go_worker(name string) {
        for i := 0; i < 10; i++ {
                fmt.Println("我是一个go协程，我的名字是 ", name, "----")
                time.Sleep(1 * time.Second)
        }
        fmt.Println(name, " 执行完毕!")
}
```

```
func main() {
        go go_worker("小黑")  //创建一个goroutine协程去执行 go_worker("小黑")
        go go_worker("小白")  //创建一个goroutine协程去执行 go_worker("小白")

        //防止main函数执行完毕,程序退出
        for {
                time.Sleep(1 * time.Second)
        }
}
```

那么多个goroutine之前如何通信呢?

```
package main

import "fmt"

func worker(c chan int) {
        //从channel中去读数据
        num := <-c
        fmt.Println("foo recv channel ", num)
}

func main() {
        //创建一个channel
        c := make(chan int)

        go worker(c)

        //main协程 向一个channel中写数据
        c <- 1

        fmt.Println("send 1 -> channel over")
}
```

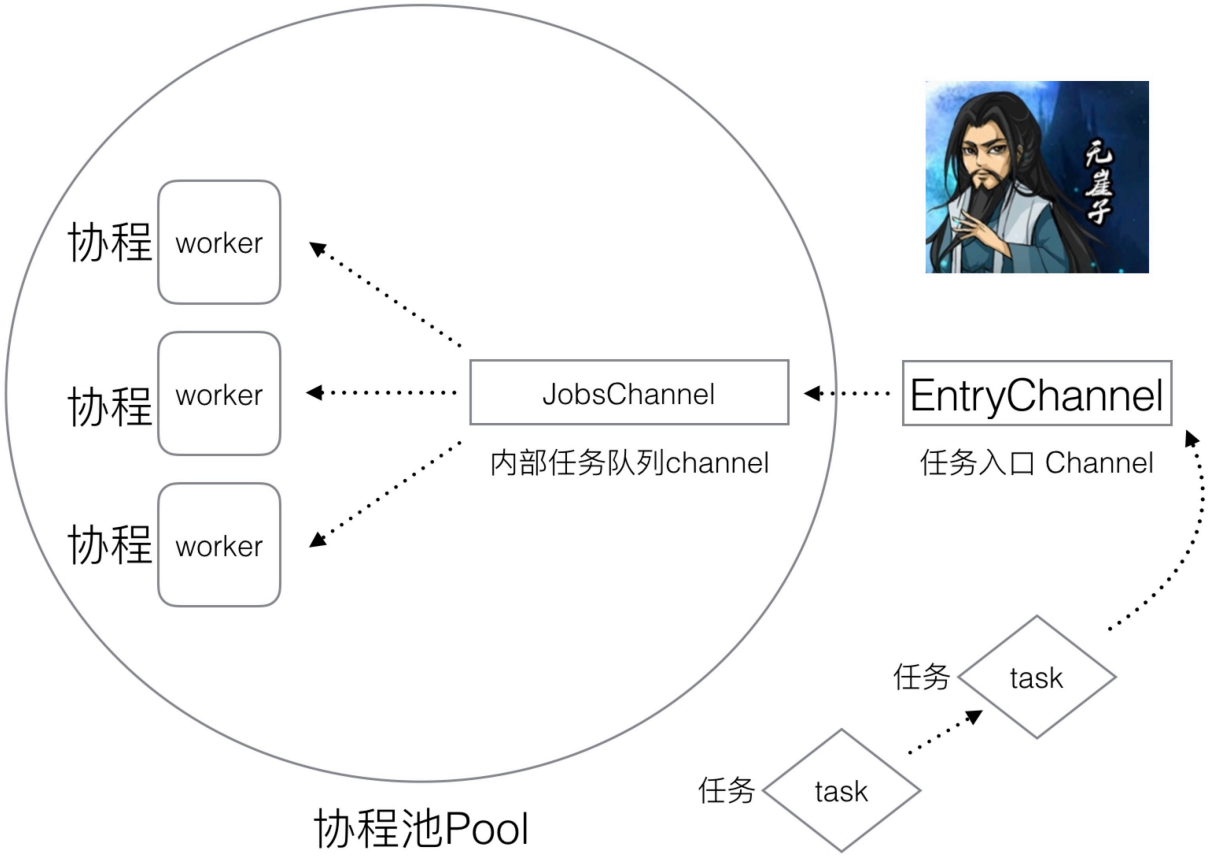
三、协程池的设计思路

为什么需要协程池?

虽然go语言在调度Goroutine已经优化的非常完成，并且Goroutine作为轻量级执行流程，也不需要CPU调度器的切换，我们一般在使用的时候，如果想处理一个分支流程，直接`go`一下即可。

但是，如果无休止的开辟Goroutine依然会出现高频率的调度Groutine，那么依然会浪费很多上下文切换的资源，导致做无用功。所以设计一个Goroutine池限制Goroutine的开辟个数在大型并发场景还是必要的。

四、快速实现并发协程通讯池




```
package main

import (
        "fmt"
        "time"
)

/* 有关Task任务相关定义及操作 */
//定义任务Task类型,每一个任务Task都可以抽象成一个函数
type Task struct {
        f func() error //一个无参的函数类型
}

//通过NewTask来创建一个Task
func NewTask(f func() error) *Task {
        t := Task{
```

```
                f: f,
        }

        return &t
}

//执行Task任务的方法
func (t *Task) Execute() {
        t.f() //调用任务所绑定的函数
}

/* 有关协程池的定义及操作 */
//定义池类型
type Pool struct {
        //对外接收Task的入口
        EntryChannel chan *Task

        //协程池最大worker数量,限定Goroutine的个数
        worker_num int

        //协程池内部的任务就绪队列
        JobsChannel chan *Task
}

//创建一个协程池
func NewPool(cap int) *Pool {
        p := Pool{
                EntryChannel: make(chan *Task),
                worker_num:   cap,
                JobsChannel:  make(chan *Task),
        }

        return &p
}

//协程池创建一个worker并且开始工作
func (p *Pool) worker(work_ID int) {
        //worker不断的从JobsChannel内部任务队列中拿任务
        for task := range p.JobsChannel {
                //如果拿到任务,则执行task任务
                task.Execute()
                fmt.Println("worker ID ", work_ID, " 执行完毕任务")
        }
}

//让协程池Pool开始工作
func (p *Pool) Run() {
        //1,首先根据协程池的worker数量限定,开启固定数量的Worker,
        //  每一个Worker用一个Goroutine承载
```

```
        for i := 0; i < p.worker_num; i++ {
                go p.worker(i)
        }

        //2, 从EntryChannel协程池入口取外界传递过来的任务
        //   并且将任务送进JobsChannel中
        for task := range p.EntryChannel {
                p.JobsChannel <- task
        }

        //3, 执行完毕需要关闭JobsChannel
        close(p.JobsChannel)

        //4, 执行完毕需要关闭EntryChannel
        close(p.EntryChannel)
}

//主函数
func main() {
        //创建一个Task
        t := NewTask(func() error {
                fmt.Println(time.Now())
                return nil
        })

        //创建一个协程池,最大开启3个协程worker
        p := NewPool(3)

        //开一个协程 不断的向 Pool 输送打印一条时间的task任务
        go func() {
                for {
                        p.EntryChannel <- t
                }
        }()

        //启动协程池p
        p.Run()

}
```

五、获取更多Go语言与区块链相关学习资料

QQ技术讨论群: