Go semaphore

part.1

Go 并发设计的一个惯用法就是将带缓冲 channel 用作计数信号量（counting semaphore）。

带缓冲 channel 中的当前数据个数代表的是当前同时处于活动状态（处理业务）的 goroutine 的数量，而带缓冲 channel 的容量（capacity）就代表了允许同时处于活动状态的 goroutine 的最大数量。

向带缓冲 channel 的一个发送操作表示获取一个信号量，而从 channel 的一个接收操作则表示释放一个信号量。

计数信号量经常被使用于限制最大并发数。

e.g.

```
package main

import (
        "log"
        "sync"
        "time"
)

func main() {
        active := make(chan struct{}, 3)
        jobs := make(chan int, 10)
```

```
        go func() {
                for i := 0; i < 8; i++ {
                        jobs <- i + 1
                }
                close(jobs)
        }()

        var wg sync.WaitGroup
        for j := range jobs {
                wg.Add(1)
                active <- struct{}{}
                go func(j int) {
                        defer func() { <-active }()
                        log.Printf("handle job: %d\n", j)
                        time.Sleep(2 * time.Second)
                        wg.Done()
                }(j)
        }
        wg.Wait()
}
```

上面的示例创建了一组 goroutines 来处理 job，同一时间允许的最多 3 个 goroutine 处于活动状态。为达成这一目标，我们看到示例使用了一个容量 (capacity) 为 3 的带缓冲 channel:**active** 作为计数信号量，这意味着允许同时处于**活动状态**的最大 goroutine 数量为 3。

运行一下该示例：

```
2021/07/14 23:15:17 handle job: 3
2021/07/14 23:15:17 handle job: 8
```

```
2021/07/14 23:15:17 handle job: 6
2021/07/14 23:15:19 handle job: 1
2021/07/14 23:15:19 handle job: 4
2021/07/14 23:15:19 handle job: 7
2021/07/14 23:15:21 handle job: 2
2021/07/14 23:15:21 handle job: 5
```

从示例运行结果中的时间戳我们可以看到：虽然我们创建了很多 goroutine，但由于计数信号量的存在，同一时间内处理活动状态(正在处理 job)的 goroutine 的数量最多为 3 个。

e.g.

```go
package main

import (
        "log"
        "math/rand"
        "time"
)

type Customer struct{ id int }
type Bar chan Customer

func (bar Bar) ServeCustomer(c Customer) {
        log.Print("++ 顾客#", c.id, "开始饮酒")
        time.Sleep(time.Second * time.Duration(3+rand.Intn(16)))
        log.Print("-- 顾客#", c.id, "离开酒吧")
        <-bar // 离开酒吧，腾出位子
}
```

```
func main() {
        rand.Seed(time.Now().UnixNano())

        bar24x7 := make(Bar, 10) // 最对同时服务10位顾客
        for customerId := 0; ; customerId++ {
                time.Sleep(time.Second * 2)
                customer := Customer{customerId}
                bar24x7 <- customer // 等待进入酒吧
                go bar24x7.ServeCustomer(customer)
        }
}
```

---

part.2

Go 在它的扩展包中提供了信号量 ***semaphore • pkg.go.dev***

- type Weighted
  - func NewWeighted(n int64) *Weighted
  - func (s *Weighted) Acquire(ctx context.Context, n int64) error
  - func (s *Weighted) Release(n int64)
  - func (s *Weighted) TryAcquire(n int64) bool

- **Acquire** | 你可以一次获取多个资源，如果没有足够多的资源，调用者就会被阻塞。它的第一个参数是 Context，这就意味着，你可以通过 Context 增加超时或者 cancel 的机制。如果是正常获取了资源，就返回 nil；否则，就返回 ctx.Err()，信号量不改变。
- **Release** | 可以将 n 个资源释放，返还给信号量。
- **TryAcquire** | 尝试获取 n 个资源，但是它不会阻塞，要么成功获取 n 个资源，返回 true，要么一个也不获取，返回 false。

e.g.

```
package main

import (
        "context"
        "log"
        "sync"
        "time"

        "golang.org/x/sync/semaphore"
)

func main() {
        sema := semaphore.NewWeighted(3)
        jobs := make(chan int, 10)
        go func() {
                for i := 0; i < 8; i++ {
                        jobs <- i + 1
                }
                close(jobs)
        }()
```

```
        ctx := context.Background()
        var wg sync.WaitGroup
        for j := range jobs {
                wg.Add(1)
                sema.Acquire(ctx, 1)
                go func(j int) {
                        defer sema.Release(1)
                        log.Printf("handle job: %d\n", j)
                        time.Sleep(2 * time.Second)
                        wg.Done()
                }(j)
        }
        wg.Wait()
}
```

e.g.

创建和 CPU 核数一样多的 Worker，让它们去处理一个 4 倍数量的整数 slice。每个 Worker 一次只能处理一个整数，处理完之后，才能处理下一个。当然，这个问题的解决方案有很多种，这一次我们使用信号量，代码如下：

```
package main

import (
        "context"
        "fmt"
        "log"
        "runtime"
        "time"
```

```
        "golang.org/x/sync/semaphore"
)

var (
        maxWorkers = runtime.GOMAXPROCS(0)                    // worker数量
        sema       = semaphore.NewWeighted(int64(maxWorkers)) // 信号量
        task       = make([]int, maxWorkers*4)                // 任务数，是worker的四倍
)

func main() {
        ctx := context.Background()

        for i := range task {
                // 如果没有worker可用，会阻塞在这里，直到某个worker被释放
                if err := sema.Acquire(ctx, 1); err != nil {
                        break
                }

                // 启动worker goroutine
                go func(i int) {
                        defer sema.Release(1)
                        time.Sleep(100 * time.Millisecond) // 模拟一个耗时操作
                        task[i] = i + 1
                }(i)
        }

        // 请求所有的worker,这样能确保前面的worker都执行完
        if err := sema.Acquire(ctx, int64(maxWorkers)); err != nil {
                log.Printf("获取所有的worker失败: %v", err)
        }
```

```
        fmt.Println(task)
}
```

如果在实际应用中，你想等所有的 Worker 都执行完，就可以获取最大计数值的信号量。

你可能会问，相比 channel 信号量的实现看起来非常简单，而且也能应对大部分的信号量的场景，为什么官方扩展库的信号量的实现不采用这种方法呢？其实，具体是什么原因，我也不知道，但是我必须要强调的是，官方的实现方式有这样一个功能：它可以一次请求多个资源，这是通过 Channel 实现的信号量所不具备的。

发布于 2021-07-14 23:24 · IP 属地北京

Golang 最佳实践