控制 goroutine

上一篇我们讲了 `go-zero` 中的并发工具包 `core/syncx`。

从整体分析来看，并发组件主要通过 `channel + mutex` 控制程序中协程之间沟通。

> Do not communicate by sharing memory; instead, share memory by communicating.
>
> 不要通过共享内存来通信，而应通过通信来共享内存。

本篇来聊 `go-zero` 对 Go 中 `goroutine` 支持的并发组件。

我们回顾一下，go原生支持的 `goroutine` 控制的工具有哪些?

1. `go func()` 开启一个协程
2. `sync.WaitGroup` 控制多个协程任务编排
3. `sync.Cond` 协程唤醒或者是协程等待

那可能会问 `go-zero` 为什么还要拿出来讲这些？回到 `go-zero` 的设计理念：**工具大于约定和文档**。

那么就来看看， `go-zero` 提供哪些工具?

threading

虽然 `go func()` 已经很方便，但是有几个问题：

- 如果协程异常退出，无法追踪异常栈
- 某个异常请求触发panic，应该做故障隔离，而不是整个进程退出，容易被攻击

我们看看 `core/threading` 包提供了哪些额外选择：

Go

```go
func GoSafe(fn func()) {
    go RunSafe(fn)
}
func RunSafe(fn func()) {
    defer rescue.Recover() fn()
}
func Recover(cleanups ...func()) {
    for _, cleanup := range cleanups {
        cleanup()
    }
    if p := recover(); p != nil {
        logx.ErrorStack(p)
    }
}
```

GoSafe

`threading.GoSafe()` 就帮你解决了这个问题。开发者可以将自己在协程中需要完成逻辑，以闭包的方式传入，由 `GoSafe()` 内部 `go func()`；

当开发者的函数出现异常退出时，会在 `Recover()` 中打印异常栈，以便让开发者更快确定异常发生点和调用栈。

NewWorkerGroup

我们再看第二个：`WaitGroup`。日常开发，其实 `WaitGroup` 没什么好说的，你需要 `N` 个协程协作：`wg.Add(N)`，等待全部协程完成任务：`wg.Wait()`，同时完成一个任务需要手动 `wg.Done()`。

可以看的出来，在任务开始 -> 结束 -> 等待，整个过程需要开发者关注任务的状态然后手动修改状态。

`NewWorkerGroup` 就帮开发者减轻了负担，开发者只需要关注：

4. 任务逻辑【函数】
5. 任务数【`workers`】

然后启动 `WorkerGroup.Start()`，对应任务数就会启动：

Go

```go
func (wg WorkerGroup) Start() {
    // 包装了sync.WaitGroup
    group := NewRoutineGroup()
    for i := 0; i < wg.workers; i++ {
        // 内部维护了 wg.Add(1) wg.Done()
        // 同时也是 goroutine 安全模式下进行的
        group.RunSafe(wg.job)
    }
    group.Wait()
}
```

`worker` 的状态会自动管理，可以用来固定数量的 `worker` 来处理消息队列的任务，用法如下：

Go

```go
func main() {
    group := NewWorkerGroup(func() {
        // process tasks
    }, runtime.NumCPU())
    group.Start()
}
```

Pool

这里的 `Pool` 不是 `sync.Pool`。`sync.Pool` 有个不方便的地方是**它池化的对象可能会被垃圾回收掉**，这个就让开发者疑惑了，不知道自己创建并存入的对象什么时候就没了。

`go-zero` 中的 `pool`：

6. `pool` 中的对象会根据使用时间做懒销毁；
7. 使用 `cond` 做对象消费和生产的通知以及阻塞；
8. 开发者可以自定义自己的生产函数，销毁函数；

那我来看看生产对象，和消费对象在 `pool` 中时怎么实现的：

Go

```go
func (p *Pool) Get() interface{} {
    // 调用 cond.Wait 时必须要持有c.L的锁
    p.lock.Lock()
    defer p.lock.Unlock()
    for {
        // 1. pool中对象池是一个用链表连接的nodelist
        if p.head != nil {
            head := p.head
            p.head = head.next
            // 1.1 如果当前节点：当前时间 >= 上次使用时间+对象最大存活时间
            if p.maxAge > 0 && head.lastUsed+p.maxAge < timex.Now() {
                p.created--         // 说明当前节点已经过期了 -> 销毁节点对应的对象，然后继续寻找下一个节点
                // 【⚠：不是销毁节点，而是销毁节点对应的对象】
                p.destroy(head.item)
                continue
            } else {
                return head.item
            }
        }
        // 2. 对象池是懒加载的，get的时候才去创建对象链表
        if p.created < p.limit {
            p.created++
            // 由开发者自己传入：生产函数
            return p.create()
        }
        p.cond.Wait()
    }
}
```

Go

```go
func (p *Pool) Put(x interface{}) { if x == nil {  return } // 互斥访问 pool 中 nodelist p.lock.Lock() defer p.lock.Unlock() p.head = &node{  item:     x,  next:     p.head,  lastUsed: timex.Now(), }  // 放入head，通知其他正在get的协程【极为关键】 p.cond.Signal()}
```

上述就是 `go-zero` 对 `Cond` 的使用。可以类比 *生产者-消费者模型*，只是在这里没有使用 `channel` 做通信，而是用 `Cond` 。这里有几个特性：

- Cond和一个Locker关联，可以利用这个Locker对相关的依赖条件更改提供保护。
- Cond可以同时支持 `Signal` 和 `Broadcast` 方法，而 `Channel` 只能同时支持其中一种。