参考文档:

https://github.com/rfyiamcool/gpool

使用示例

```
        gp, err := gpool.NewGPool(&gpool.Options{
                MaxWorker:   5,                  // 最大的协程数
                MinWorker:   2,                  // 最小的协程数
                JobBuffer:   1,                  // 缓冲队列的大小
                IdleTimeout: 120 * time.Second, // 协程的空闲超时退出时间
        })

        if err != nil {
                panic(err.Error())
        }

        for index := 0; index < 1000; index++ {
                wg.Add(1)
                idx := index
                 // 这里异步, ProcessSync是同步
                gp.ProcessAsync(func() {
                        fmt.Println(idx, time.Now())
                        time.Sleep(1 * time.Second)
                        wg.Done()
                })
        }

        wg.Done()

```

实现思路

1. 开启管理WorkerCount的数目, 这里开启关闭时候需要使用sync.Mutex
2. 实例化,就开启规定数据worker, 运行 go worker
3. 获取任务使用chan, 传递工作任务task
4. 每个worker运行Hander, 里面进行的工作是如下:

    a. 开启定时器
    b. 监听JobChan是否有任务, 有就去完成并且重置定时器时间
    c. 定时器到点,证明没有任务, 执行workerExit, 删除该worker
        i. 这里删除需要用到sync.Mutex, curWorkerCount--,特别判断worker数目不能少于minWorkerCount
    d. 监听ctx.Done() 管道,因为会退出
5. 执行Process同步或者异步方法时, 会尝试扩大worker数量大于min, 然后 实例化task, pool.jobChannel <- task 传递任务执行
6. 使用task的 res chan struct{} 来实现同步或者异步方法, 同步就是要等待res, 异步res 就是Nil
7. 最后是pool.Stop方法, 加锁, 修改isClose bool 值, 然后执行cancelCtx的cancel方法

源码解读

```
1  package gpools
2
3  import (
4      "context"
5      "errors"
6      "fmt"
7      "sync"
8      "time"
9  )
10
11 type Options struct {
12     // number of maxworkers,
13     MaxWorker int
14     MinWorker int
15     // size of job queue
16     JobBuffer      int
17     IdleTimeout    time.Duration
18     DispatchPeriod time.Duration //TODO 暂时没开发, 就是请求超时的时间
19 }
20
21 type Pool struct {
22     sync.Mutex
23     ctx     context.Context
24     cancel  context.CancelFunc
25     isClose bool
26
```

```
27      // workerCount
28      maxWorkerCount int
29      minWorkerCount int
30      curWorkerCount int
31
32      // size of job queue
33      idleTimeout    time.Duration
34      dispatchPeriod time.Duration
35      jobBuffer      int
36      jobChannel     chan taskModel
37  }
38
39  type JobFunc func()
40
41  type taskModel struct {
42      call JobFunc
43      res  chan bool
44  }
45
46  func NewPool(op *Options) *Pool {
47      pool := &Pool{}
48      pool.maxWorkerCount = op.MaxWorker
49      pool.minWorkerCount = op.MinWorker
50      pool.jobBuffer = op.JobBuffer
51      pool.idleTimeout = op.IdleTimeout
52      pool.dispatchPeriod = op.DispatchPeriod
53
54      // check Options
55      if pool.maxWorkerCount <= 1 {
56          pool.maxWorkerCount = 5
57      }
58      if pool.minWorkerCount <= 0 {
59          pool.minWorkerCount = 2
60      }
61
62      ctx, cancel := context.WithCancel(context.Background())
63      pool.ctx = ctx
64      pool.cancel = cancel
65      pool.jobChannel = make(chan taskModel, pool.jobBuffer)
```

```
66
67      // start
68      pool.spawnWorker(pool.maxWorkerCount)
69
70      return pool
71  }
72
73  func (p *Pool) spawnWorker(num int) {
74      p.Lock()
75      defer p.Unlock()
76
77      if num == p.curWorkerCount {
78          return
79      }
80
81      for i := 0; i < num; i++ {
82          if p.curWorkerCount > p.minWorkerCount {
83              return
84          }
85          // 起 go
86          p.curWorkerCount++
87          go p.handler()
88      }
89  }
90
91  // 处理任务
92  func (p *Pool) handler() {
93      // 定时idleTimeOut退出
94      // 处理任务
95      timer := time.NewTicker(p.idleTimeout)
96      defer timer.Stop()
97
98      for {
99          select {
100         case task := <-p.jobChannel: // 管道不关闭不会deadlock，会自动到下一个case匹配
101             task.call()
102             // 异步为nil
103             if task.res != nil {
104                 time.Sleep(10 * time.Second)
```

```
105                 task.res <- true
106             }
107             // 需要重置
108             timer.Reset(p.idleTimeout)
109         case <-timer.C:
110             // 超时
111             err := p.workerExit()
112             if err != nil {
113                 timer.Reset(p.idleTimeout)
114                 continue
115             }
116             return
117         case <-p.ctx.Done():
118             fmt.Println("pool ctx Done")
119             return
120         }
121
122     }
123 }
124
125 // 线程退出
126 func (p *Pool) workerExit() error {
127     p.Lock()
128     defer p.Unlock()
129
130     if p.curWorkerCount <= p.minWorkerCount {
131         return errors.New("can not less than minWorkerCount")
132     }
133
134     p.curWorkerCount--
135     return nil
136 }
137
138 // 同步运行
139 func (p *Pool) ProcessSync(f JobFunc) {
140     task := taskModel{
141         call: f,
142         res:  make(chan bool),
143     }
```

```
144     // 申请增大
145     p.trySpawnWorker()
146 
147     p.jobChannel <- task
148     //同步
149     <-task.res
150     fmt.Println("finish ProcessSync")
151 }
152 
153 // 扩大worker
154 func (p *Pool) trySpawnWorker() {
155     if len(p.jobChannel) > 0 && p.maxWorkerCount > p.curWorkerCount {
156         p.spawnWorker(p.maxWorkerCount / 5) // 自己判断算法
157     }
158 }
159 
160 // 退出
161 func (p *Pool) Stop() {
162     p.Lock()
163     defer p.Unlock()
164 
165     if p.isClose {
166         return
167     }
168     // close(p.jobChannel)  // 可以不用关闭,自己回收, 管道不关闭不会deadlock，会自动到下一个case匹配
169 
170     p.cancel()
171     p.isClose = true
172 }
173 
```