链滴

Golang 定时任务 (cron 包)

作者：K

原文链接：https://ld246.com/article/1534497226108

来源网站：链滴

cron是什么

cron 克龙(时间单位；等于百万年).

linux下的定时执行工具.

golang cron: 计划任务，定时任务。我和系统约个时间，你在几点几分几秒或者每到一个时间跑一个务(job)。

cron表达式

cron表达式是一个好东西，这个东西不仅Java的quartZ能用到，Go语言中也可以用到。我没有用过Linux的cron，但网上说Linux也是可以用crontab -e 命令来配置定时任务。Go语言和Java中都是可以确到秒的，但是Linux中不行。

cron表达式代表一个时间的集合，使用6个空格分隔的字段表示：

字段名	是否必须	允许的值	许的特定字符
秒(Seconds)	是	0-59	* / , -
分(Minute)	是	0-59	* / , -
时(Hours)	是	0-23	* / , -
日(Day of month)	是	1-31	* / , - ?
月(Month)	是	1-12 或 JAN-DEC	* / , -
星期(Day of week)	否	0-6 或 SUM-SAT	

原文链接：Golang 定时任务 (cron 包)

/ , - ?

NOTE

- 月(Month)和星期(Day of week)字段的值不区分大小写，如：SUN、Sun 和 sun 是一样的。
- 星期(Day of week)字段如果没提供，相当于是 *

```
# ┌──────────────────── min (0 - 59)
# │ ┌────────────────────── hour (0 - 23)
# │ │ ┌──────────────────────── day of month (1 - 31)
# │ │ │ ┌────────────────────────── month (1 - 12)
# │ │ │ │ ┌──────────────────────────── day of week (0 - 6) (0 to 6 are Sunday to
# │ │ │ │ │                  Saturday, or use names; 7 is also Sunday)
# │ │ │ │ │
# │ │ │ │ │
# * * * * *  command to execute
```

cron特定字符说明

1. 星号(*) 表示 cron 表达式能匹配该字段的所有值。如在第5个字段使用星号(month)，表示每个月
2. 斜线(/) 表示增长间隔，如第1个字段(minutes) 值是 3-59/15，表示每小时的第3分钟开始执行一，之后每隔 15 分钟执行一次（即 3、18、33、48 这些时间点执行），这里也可以表示为：3/15
3. 逗号(,) 用于枚举值，如第6个字段值是 MON,WED,FRI，表示 星期一、三、五 执行
4. 连字号(-) 表示一个范围，如第3个字段的值为 9-17 表示 9am 到 5pm 直接每个小时（包括9和17）
5. 问号(?) 只用于 日(Day of month) 和 星期(Day of week)，表示不指定值，可以用于代替 *
6. L，W，# Go中没有L，W，#的用法，下文作解释。

cron举例说明

- 每隔5秒执行一次：*/5 * * * *
?
- 每隔1分钟执行一次：0 */1 * * *
?
- 每天23点执行一次：0 0 23 * *
?
- 每天凌晨1点执行一次：0 0 1 * *
?
- 每月1号凌晨1点执行一次：0 0 1 1 * ?
- 在26分、29分、33分执行一次：0 26,29,33 * * *
?
- 每天的0点、13点、18点、21点都执行一次：0 0 0,13,18,21 * *
?

下载安装

控制台输入 go get github.com/robfig/cron 去下载定时任务的Go包 (注意 go 环境)

源码解析

文件目录

```
constantdelay.go      #一个最简单的秒级别定时系统。与cron无关
constantdelay_test.go #测试
cron.go               #Cron系统。管理一系列的cron定时任务（Schedule Job）
cron_test.go          #测试
doc.go                #说明文档
LICENSE               #授权书
parser.go             #解析器，解析cron格式字符串城一个具体的定时器（Schedule）
parser_test.go        #测试
README.md             #README
spec.go               #单个定时器（Schedule）结构体。如何计算自己的下一次触发时间
spec_test.go          #测试
```

cron.go结构体

```
// Cron keeps track of any number of entries, invoking the associated func as

// specified by the schedule. It may be started, stopped, and the entries may
// be inspected while running.
// Cron保持任意数量的条目的轨道，调用相关的func时间表指定。它可以被启动，停止和条目，可行的同时进行检查。
type Cron struct {
    entries  []*Entry                // 任务
    stop     chan struct{}      // 叫停止的途径
    add      chan *Entry        // 添加新任务的方式
    snapshot chan []*Entry      // 请求获取任务快照的方式
    running  bool               // 是否在运行
    ErrorLog *log.Logger        // 出错日志(新增属性)
    location *time.Location     // 所在地区(新增属性)
}

// Entry consists of a schedule and the func to execute on that schedule.

// 入口包括时间表和可在时间表上执行的func
type Entry struct {
        // 计时器
    Schedule Schedule
    // 下次执行时间
    Next time.Time
    // 上次执行时间
    Prev time.Time
    // 任务
    Job Job
}
```

关键方法

```
//  开始任务
```

```
// Start the cron scheduler in its own go-routine, or no-op if already started.
func (c *Cron) Start() {
    if c.running {
        return
    }
    c.running = true
    go c.run()
}
// 结束任务
// Stop stops the cron scheduler if it is running; otherwise it does nothing.
func (c *Cron) Stop() {
    if !c.running {
        return
    }
    c.stop <- struct{}{}
    c.running = false
}

// 执行定时任务
// Run the scheduler.. this is private just due to the need to synchronize
// access to the 'running' state variable.
func (c *Cron) run() {
    // Figure out the next activation times for each entry.
    now := time.Now().In(c.location)
    for _, entry := range c.entries {
        entry.Next = entry.Schedule.Next(now)
    }
        // 无限循环
    for {
            //通过对下一个执行时间进行排序，判断那些任务是下一次被执行的，防在队列的前面.sort
用来做排序的
        sort.Sort(byTime(c.entries))

        var effective time.Time
        if len(c.entries) == 0 || c.entries[0].Next.IsZero() {
            // If there are no entries yet, just sleep - it still handles new entries
            // and stop requests.
            effective = now.AddDate(10, 0, 0)
        } else {
            effective = c.entries[0].Next
        }

        timer := time.NewTimer(effective.Sub(now))
        select {
        case now = <-timer.C:  // 执行当前任务
            now = now.In(c.location)
            // Run every entry whose next time was this effective time.
            for _, e := range c.entries {
                if e.Next != effective {
                    break
                }
                go c.runWithRecovery(e.Job)
                e.Prev = e.Next
                e.Next = e.Schedule.Next(now)
```

原文链接：Golang 定时任务 (cron 包)

```
            }
            continue

        case newEntry := <-c.add:  // 添加新的任务
            c.entries = append(c.entries, newEntry)
            newEntry.Next = newEntry.Schedule.Next(time.Now().In(c.location))

        case <-c.snapshot:  // 获取快照
            c.snapshot <- c.entrySnapshot()

        case <-c.stop:   // 停止任务
            timer.Stop()
            return
        }

        // 'now' should be updated after newEntry and snapshot cases.
        now = time.Now().In(c.location)
        timer.Stop()
    }
}
```

spec.go

```
// SpecSchedule specifies a duty cycle (to the second granularity), based on a

// traditional crontab specification. It is computed initially and stored as bit sets.
type SpecSchedule struct {
    // 表达式中锁表明的，秒，分，时，日，月，周，每个都是uint64
    // Dom:Day of Month,Dow:Day of week
    Second, Minute, Hour, Dom, Month, Dow uint64
}

// bounds provides a range of acceptable values (plus a map of name to value).
// 定义了表达式的结构体
type bounds struct {
    min, max uint
    names    map[string]uint
}


// The bounds for each field.
// 这样就能看出各个表达式的范围
var (
      seconds = bounds{0, 59, nil}
      minutes = bounds{0, 59, nil}
      hours   = bounds{0, 23, nil}
      dom     = bounds{1, 31, nil}
      months  = bounds{1, 12, map[string]uint{
            "jan": 1,
            "feb": 2,
            "mar": 3,
            "apr": 4,
            "may": 5,
            "jun": 6,
```

```
            "jul": 7,
            "aug": 8,
            "sep": 9,
            "oct": 10,
            "nov": 11,
            "dec": 12,
        }}
        dow = bounds{0, 6, map[string]uint{
            "sun": 0,
            "mon": 1,
            "tue": 2,
            "wed": 3,
            "thu": 4,
            "fri": 5,
            "sat": 6,
        }}
)

const (
        // Set the top bit if a star was included in the expression.
        starBit = 1 << 63
)
```

看了上面的东西肯定有人疑惑为什么秒分时这些都是定义了unit64,以及定义了一个常量starBit = 1 << 63这种写法，这是逻辑运算符。表示二进制1向左移动63位。原因如下：

cron表达式是用来表示一系列时间的，而时间是无法逃脱自己的区间的， 分，秒 0 - 59， 时 0 - 23， 天/月 0 - 31， 天/周 0 - 6， 月0 - 11 。 这些本质上都是一个点集合，或者说是一个整数区间。那么对于任意的整数区间， 可以描述cron的如下部分规则。

- * | ? 任意， 对应区间上的所有点。 （ 额外注意 日/周， 日 / 月 的相互干扰。）
- 纯数字， 对应一个具体的点。
- / 分割的两个数字 a , b， 区间上符合 a + n * b 的所有点 （ n >= 0)。
- - 分割的两个数字， 对应这两个数字决定的区间内的所有点。
- L | W 需要对于特定的时间特殊判断， 无法通用的对应到区间上的点。

至此， robfig/cron为什么不支持 L | W的原因已经明了了。去除这两条规则后， 其余的规则其实完可以使用点的穷举来通用表示。 考虑到最大的区间也不过是60个点，那么使用一个uint64的整数的一位来表示一个点便很合适了。所以定义unit64不为过

下面是go中cron表达式的方法：

```
/*
  ---------------------------------------------------------
  第64位标记任意， 用于 日/周， 日 / 月 的相互干扰。
- 0 为 表示区间 [63 , 0] 的 每一个点。
  ---------------------------------------------------------

  假设区间是 0 - 63， 则有如下的例子：

  比如  0/3 的表示如下： (表示每隔两位为1)
  * / ?
  +---+-----------------------------------------------------+
```

```
  |0|1001001 ~~ ~~            1001001|
  +---+-----------------------------------------------------+
~ ~                           ~~ 0

  比如 2-5 的表示如下： (表示从右往左2-5位上都是1)
  */?
  +---+-----------------------------------------------------+
  |0|0000~ ~    ~~       ~  0001111100|
  +---+-----------------------------------------------------+
~ ~                           ~~ 0

 比如 * 的表示如下： (表示所有位置上都为1)
  */?
  +---+-----------------------------------------------------+
  |1|11111~ ~           ~  1111111111|
  +---+-----------------------------------------------------+
~ ~                           ~~ 0
*/
```

parser.go 将字符串解析为SpecSchedule的类。

```
package cron


import (
    "fmt"
    "math"
    "strconv"
    "strings"
    "time"
)

// Configuration options for creating a parser. Most options specify which
// fields should be included, while others enable features. If a field is not
// included the parser will assume a default value. These options do not change
// the order fields are parse in.
type ParseOption int

const (
    Second      ParseOption = 1 << iota // Seconds field, default 0
    Minute                              // Minutes field, default 0
    Hour                                // Hours field, default 0
    Dom                                 // Day of month field, default *
    Month                               // Month field, default *
    Dow                                 // Day of week field, default *
    DowOptional                         // Optional day of week field, default *
    Descriptor                          // Allow descriptors such as @monthly, @weekly, etc.
)

var places = []ParseOption{
    Second,
    Minute,
    Hour,
    Dom,
```

```
    Month,
    Dow,
}

var defaults = []string{
    "0",
    "0",
    "0",
    "*",
    "*",
    "*",
}

// A custom Parser that can be configured.
type Parser struct {
    options   ParseOption
    optionals int
}

// Creates a custom Parser with custom options.
//
//  // Standard parser without descriptors
//  specParser := NewParser(Minute | Hour | Dom | Month | Dow)
//  sched, err := specParser.Parse("0 0 15 */3 *")
//
//  // Same as above, just excludes time fields
//  subsParser := NewParser(Dom | Month | Dow)
//  sched, err := specParser.Parse("15 */3 *")
//
//  // Same as above, just makes Dow optional
//  subsParser := NewParser(Dom | Month | DowOptional)
//  sched, err := specParser.Parse("15 */3")
//
func NewParser(options ParseOption) Parser {
    optionals := 0
    if options&DowOptional > 0 {
        options |= Dow
        optionals++
    }
    return Parser{options, optionals}
}

// Parse returns a new crontab schedule representing the given spec.
// It returns a descriptive error if the spec is not valid.
// It accepts crontab specs and features configured by NewParser.
// 将字符串解析成为SpecSchedule 。 SpecSchedule符合Schedule接口

func (p Parser) Parse(spec string) (Schedule, error) {
      // 直接处理特殊的特殊的字符串
    if spec[0] == '@' && p.options&Descriptor > 0 {
        return parseDescriptor(spec)
    }

    // Figure out how many fields we need
```

```
    max := 0
    for _, place := range places {
        if p.options&place > 0 {
            max++
        }
    }
    min := max - p.optionals

    // cron利用空白拆解出独立的items。
    fields := strings.Fields(spec)

    // 验证表达式取值范围
    if count := len(fields); count < min || count > max {
        if min == max {
            return nil, fmt.Errorf("Expected exactly %d fields, found %d: %s", min, count, spec)
        }
        return nil, fmt.Errorf("Expected %d to %d fields, found %d: %s", min, max, count, spec)
    }

    // Fill in missing fields
    fields = expandFields(fields, p.options)

    var err error
    field := func(field string, r bounds) uint64 {
        if err != nil {
            return 0
        }
        var bits uint64
        bits, err = getField(field, r)
        return bits
    }

    var (
        second     = field(fields[0], seconds)
        minute     = field(fields[1], minutes)
        hour       = field(fields[2], hours)
        dayofmonth = field(fields[3], dom)
        month      = field(fields[4], months)
        dayofweek  = field(fields[5], dow)
    )
    if err != nil {
        return nil, err
    }
    // 返回所需要的SpecSchedule
    return &SpecSchedule{
        Second: second,
        Minute: minute,
        Hour:   hour,
        Dom:    dayofmonth,
        Month:  month,
        Dow:    dayofweek,
    }, nil
}
```

```
func expandFields(fields []string, options ParseOption) []string {
    n := 0
    count := len(fields)
    expFields := make([]string, len(places))
    copy(expFields, defaults)
    for i, place := range places {
        if options&place > 0 {
            expFields[i] = fields[n]
            n++
        }
        if n == count {
            break
        }
    }
    return expFields
}

var standardParser = NewParser(
    Minute | Hour | Dom | Month | Dow | Descriptor,
)

// ParseStandard returns a new crontab schedule representing the given standardSpec
// (https://en.wikipedia.org/wiki/Cron). It differs from Parse requiring to always
// pass 5 entries representing: minute, hour, day of month, month and day of week,
// in that order. It returns a descriptive error if the spec is not valid.
//
// It accepts
//   - Standard crontab specs, e.g. "* * * * ?"
//   - Descriptors, e.g. "@midnight", "@every 1h30m"
// 这里表示不仅可以使用cron表达式，也可以使用@midnight @every等方法

func ParseStandard(standardSpec string) (Schedule, error) {
    return standardParser.Parse(standardSpec)
}

var defaultParser = NewParser(
    Second | Minute | Hour | Dom | Month | DowOptional | Descriptor,
)

// Parse returns a new crontab schedule representing the given spec.
// It returns a descriptive error if the spec is not valid.
//
// It accepts
//   - Full crontab specs, e.g. "* * * * * ?"
//   - Descriptors, e.g. "@midnight", "@every 1h30m"
func Parse(spec string) (Schedule, error) {
    return defaultParser.Parse(spec)
}

// getField returns an Int with the bits set representing all of the times that
// the field represents or error parsing field value.  A "field" is a comma-separated
// list of "ranges".
func getField(field string, r bounds) (uint64, error) {
    var bits uint64
```

```
    ranges := strings.FieldsFunc(field, func(r rune) bool { return r == ',' })
    for _, expr := range ranges {
        bit, err := getRange(expr, r)
        if err != nil {
            return bits, err
        }
        bits |= bit
    }
    return bits, nil
}

// getRange returns the bits indicated by the given expression:
//   number | number "-" number [ "/" number ]
// or error parsing range.
func getRange(expr string, r bounds) (uint64, error) {
    var (
        start, end, step uint
        rangeAndStep     = strings.Split(expr, "/")
        lowAndHigh       = strings.Split(rangeAndStep[0], "-")
        singleDigit      = len(lowAndHigh) == 1
        err              error
    )

    var extra uint64
    if lowAndHigh[0] == "*" || lowAndHigh[0] == "?" {
        start = r.min
        end = r.max
        extra = starBit
    } else {
        start, err = parseIntOrName(lowAndHigh[0], r.names)
        if err != nil {
            return 0, err
        }
        switch len(lowAndHigh) {
        case 1:
            end = start
        case 2:
            end, err = parseIntOrName(lowAndHigh[1], r.names)
            if err != nil {
                return 0, err
            }
        default:
            return 0, fmt.Errorf("Too many hyphens: %s", expr)
        }
    }

    switch len(rangeAndStep) {
    case 1:
        step = 1
    case 2:
        step, err = mustParseInt(rangeAndStep[1])
        if err != nil {
            return 0, err
        }
```

```
            // Special handling: "N/step" means "N-max/step".
            if singleDigit {
                end = r.max
            }
        default:
            return 0, fmt.Errorf("Too many slashes: %s", expr)
        }

        if start < r.min {
            return 0, fmt.Errorf("Beginning of range (%d) below minimum (%d): %s", start, r.min, expr

        }
        if end > r.max {
            return 0, fmt.Errorf("End of range (%d) above maximum (%d): %s", end, r.max, expr)
        }
        if start > end {
            return 0, fmt.Errorf("Beginning of range (%d) beyond end of range (%d): %s", start, end,
xpr)
        }
        if step == 0 {
            return 0, fmt.Errorf("Step of range should be a positive number: %s", expr)
        }

        return getBits(start, end, step) | extra, nil
    }

    // parseIntOrName returns the (possibly-named) integer contained in expr.
    func parseIntOrName(expr string, names map[string]uint) (uint, error) {
        if names != nil {
            if namedInt, ok := names[strings.ToLower(expr)]; ok {
                return namedInt, nil
            }
        }
        return mustParseInt(expr)
    }

    // mustParseInt parses the given expression as an int or returns an error.
    func mustParseInt(expr string) (uint, error) {
        num, err := strconv.Atoi(expr)
        if err != nil {
            return 0, fmt.Errorf("Failed to parse int from %s: %s", expr, err)
        }
        if num < 0 {
            return 0, fmt.Errorf("Negative number (%d) not allowed: %s", num, expr)
        }

        return uint(num), nil
    }

    // getBits sets all bits in the range [min, max], modulo the given step size.
    func getBits(min, max, step uint) uint64 {
        var bits uint64
```

```
    // If step is 1, use shifts.
    if step == 1 {
        return ^(math.MaxUint64 << (max + 1)) & (math.MaxUint64 << min)
    }

    // Else, use a simple loop.
    for i := min; i <= max; i += step {
        bits |= 1 << i
    }
    return bits
}

// all returns all bits within the given bounds.  (plus the star bit)
func all(r bounds) uint64 {
    return getBits(r.min, r.max, 1) | starBit
}

// parseDescriptor returns a predefined schedule for the expression, or error if none matches.
func parseDescriptor(descriptor string) (Schedule, error) {
    switch descriptor {
    case "@yearly", "@annually":
        return &SpecSchedule{
            Second: 1 << seconds.min,
            Minute: 1 << minutes.min,
            Hour:   1 << hours.min,
            Dom:    1 << dom.min,
            Month:  1 << months.min,
            Dow:    all(dow),
        }, nil

    case "@monthly":
        return &SpecSchedule{
            Second: 1 << seconds.min,
            Minute: 1 << minutes.min,
            Hour:   1 << hours.min,
            Dom:    1 << dom.min,
            Month:  all(months),
            Dow:    all(dow),
        }, nil

    case "@weekly":
        return &SpecSchedule{
            Second: 1 << seconds.min,
            Minute: 1 << minutes.min,
            Hour:   1 << hours.min,
            Dom:    all(dom),
            Month:  all(months),
            Dow:    1 << dow.min,
        }, nil

    case "@daily", "@midnight":
        return &SpecSchedule{
            Second: 1 << seconds.min,
            Minute: 1 << minutes.min,
```

```
            Hour:   1 << hours.min,
            Dom:    all(dom),
            Month:  all(months),
            Dow:    all(dow),
        }, nil

    case "@hourly":
        return &SpecSchedule{
            Second: 1 << seconds.min,
            Minute: 1 << minutes.min,
            Hour:   all(hours),
            Dom:    all(dom),
            Month:  all(months),
            Dow:    all(dow),
        }, nil
    }

    const every = "@every "
    if strings.HasPrefix(descriptor, every) {
        duration, err := time.ParseDuration(descriptor[len(every):])
        if err != nil {
            return nil, fmt.Errorf("Failed to parse duration %s: %s", descriptor, err)
        }
        return Every(duration), nil
    }

    return nil, fmt.Errorf("Unrecognized descriptor: %s", descriptor)
}
```

简单的实例

运行都是以一个精确的时间运行的

(每 10 秒运行, 就是每逢10, 20, 30, 40, 50, 0秒运行.)

```
package main

import (
  "github.com/robfig/cron"
  "fmt"
  "time"
)

func main() {
  c := cron.New()
  spec := "*/10 * * * * ?"
  c.AddFunc(spec, func() {
    now := time.Now()
    fmt.Println("cron running:", now.Minute(), now.Second())
  })
  c.Start()
  select{}
}
```

```
cron running: 50 20
cron running: 50 30
cron running: 50 40
cron running: 50 50
```

源地址 稍加修改.