Golang定时任务


ourlang LV.4
2021年11月05日 22:21 · 阅读 1585

Golang定时任务

1 安装依赖

```sh
go get github.com/robfig/cron/v3@v3.0.0
```

2 定时任务HelloWorld

```go
package main

import (
        "fmt"
        "github.com/robfig/cron/v3"
        "time"
)

func main() {
        // 新建一个定时任务对象
        // 根据cron表达式进行时间调度，cron可以精确到秒，大部分表达式格式也是从秒开始。
        //crontab := cron.New()  默认从分开始进行时间调度
        crontab := cron.New(cron.WithSeconds()) //精确到秒
        //定义定时器调用的任务函数
        task := func() {
                fmt.Println("hello world", time.Now())
        }
        //定时任务
        spec := "*/5 * * * * ?" //cron表达式，每五秒一次
        // 添加定时任务,
        crontab.AddFunc(spec, task)
        // 启动定时器
        crontab.Start()
        // 定时任务是另起协程执行的,这里使用 select 简答阻塞.实际开发中需要
        // 根据实际情况进行控制
        select {} //阻塞主线程停止
}
----------------------------------------输出结果----------------------------------
hello world 2020-03-18 11:13:00.0241639 +0800 CST m=+3.113746301
hello world 2020-03-18 11:13:05.0007375 +0800 CST m=+8.090319901
hello world 2020-03-18 11:13:10.0004232 +0800 CST m=+13.090005601
hello world 2020-03-18 11:13:15.0003857 +0800 CST m=+18.089968101
hello world 2020-03-18 11:13:20.0003788 +0800 CST m=+23.089961201
```

3 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 中都是可以精确到秒的，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	允许的特定字符
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	* / , - ?

3.1 Corn表达式说明

- 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 *

3.2 Corn表达式示例说明

> 如果我们使用 `crontab := cron.New(cron.WithSeconds())` ，比如我们传递了一个字符串是： `"* * * * * *"` 在 `crontab.AddFunc()` 的第一个参数，这六个 `*` 是指什么呢？ 如果是用 `crontab := cron.New()` 则只需要五个 `*` ,如 `* * * * *`

```go
┌──────────────second 范围 (0 - 60)
│ ┌──────────── 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)
│ │ │ │ │ │
│ │ │ │ │ │
* * * * * *
```

3.3 cron特定字符说明

符号	说明
(*)	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
(/)	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分钟开始执行一次，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
(,)	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
(-)	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）
(?)	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 *

3.4 常用cron举例

```go
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 * ?

每周一和周三晚上22:30: 00 30 22 * * 1,3

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 * * ?

每年三月的星期四的下午14:10和14:40: 00 10,40 14 ? 3 4

3.5 预定义的时间格式

您可以使用几个预定义的表达式来代替上表的表达式,使用如下

输入	描述	等式
`@yearly (or @annually)`	每年1月1日午夜跑步一次	0 0 0 1 1 *
`@monthly`	每个月第一天的午夜跑一次	0 0 0 1 * *
`@weekly`	每周周六的午夜运行一次	0 0 0 * * 0
`@daily (or @midnight)`	每天午夜跑一次	0 0 0 * * *
`@hourly`	每小时运行一次	0 0 * * * *
`@every <duration>`	every duration	

```go
c := cron.New()
c.AddFunc("@every 1h30m", func() { fmt.Println("Every hour thirty") })
```

定时任务的三种模式

1. 单实例本地定时任务
2. 多实例本地定时任务（需要解决多实例并行执行任务的问题）
3. 第三方调度（例如单独部署一套定时任务调度系统）

cron是一个定时任务管理框架，可以将本地服务中所有的定时任务统一管理起来。

cron代码库：https://github.com/robfig/cron

可管理多个定时任务

多任务代码示例：

```
c := cron.New(cron.WithSeconds())
spec := "*/5 * * * * *" // 每隔5s执行一次，cron格式（秒，分，时，天，月，周）
// 添加一个任务
c.AddFunc(spec, func() {
    log.Printf("111 time = %d\n", time.Now().Unix())
})
// 添加一个任务
```

```
c.AddFunc("*/1 * * * * *", func() { // 可以随时添加多个定时任务
    log.Printf("222")
})
c.Start()
```

默认上次任务没运行完，下次任务依然会运行（任务运行在goroutine里相互不干扰）

代码示例：

```
c := cron.New(cron.WithSeconds())
c.AddFunc("*/1 * * * * *", func() { // 每隔一秒执行一次
   unix := time.Now().Unix()
   fmt.Println("111--start, time=%d", unix)
   time.Sleep(2 * time.Second) // 任务执行耗时，超过定时间隔
   fmt.Println("111--end, time=%d", unix)
})
c.Start()
```

输出如下：

```
多个任务在并行执行
111--start, time=%d 1600431216
111--start, time=%d 1600431217
111--start, time=%d 1600431218
111--end, time=%d 1600431216
111--start, time=%d 1600431219
111--end, time=%d 1600431217
111--end, time=%d 1600431218
111--start, time=%d 1600431220
```

支持上次任务未执行完，下次任务不启动

代码示例：

```
c := cron.New(cron.WithSeconds(),cron.WithChain(cron.SkipIfStillRunning(cron.VerbosePrintfLogger(log.New(os.Stdout, "cron: ", log.LstdFlags)))))
c.AddFunc("*/1 * * * * *", func() {
   unix := time.Now().Unix()
   fmt.Println("111--start, time=%d", unix)
   time.Sleep(2 * time.Second)
   fmt.Println("111--end, time=%d", unix)
})
c.Start()
```

输出如下：

```
任务按顺序执行，没有交叉重叠
111--start, time=%d 1600431474
cron: 2020/09/18 20:17:55 skip
cron: 2020/09/18 20:17:56 skip
111--end, time=%d 1600431474
111--start, time=%d 1600431477
cron: 2020/09/18 20:17:58 skip
```

4 多个定时任务

go 复制代码

```
package main

import (
```

```
        "fmt"
        "github.com/robfig/cron/v3"
        "time"
)

type TestTask struct {
        Name string
}

func (t *TestTask) Run() {
        fmt.Println("TestTask", t.Name)
}

type Test2Task struct {
        Name string
}

func (t *Test2Task) Run() {
        fmt.Println("Test2Task", t.Name)
}

func main() {
        // 新建一个定时任务对象
        // 根据cron表达式进行时间调度，cron可以精确到秒，大部分表达式格式也是从秒开始。
        //crontab := cron.New()  默认从分开始进行时间调度
        crontab := cron.New(cron.WithSeconds()) //精确到秒
        //定义定时器调用的任务函数
        //定时任务
        spec := "*/5 * * * * ?" //cron表达式，每五秒一次

        //定义定时器调用的任务函数
        task := func() {
                fmt.Println("hello world", time.Now())
        }
        // 添加定时任务
        crontab.AddFunc(spec, task)

        // 添加多个定时器
        crontab.AddJob(spec, &TestTask{Name: "tom"})
        crontab.AddJob(spec, &Test2Task{Name: "jeck"})
        // 启动定时器
        crontab.Start()
        //关闭着计划任务, 但是不能关闭已经在执行中的任务.
        defer crontab.Stop()
        // 定时任务是另起协程执行的,这里使用 select 简答阻塞.实际开发中需要
        // 根据实际情况进行控制
        select {} //阻塞主线程停止
}
--------------------------------------输出结果----------------------------------------
TestTask tom
Test2Task jeck
hello world 2020-03-18 13:22:10.0245997 +0800 CST m=+2.761856801
TestTask tom
Test2Task jeck
hello world 2020-03-18 13:22:15.0008245 +0800 CST m=+7.738081601
.....................................
```

5 主要类型或接口说明

5.1 Cron

> Cron：包含一系列要执行的实体；支持暂停【stop】；添加实体等

- 注意：
  - Cron 结构没有导出任何成员。

- 有一个成员 `stop`，类型是 `struct{}`，即空结构体。

```go
type Cron struct {
    entries  []*Entry
    stop     chan struct{}   // 控制 Cron 实例暂停
    add      chan *Entry     // 当 Cron 已经运行了，增加新的 Entity 是通过 add 这个 channel 实现的
    snapshot chan []*Entry   // 获取当前所有 entity 的快照
    running  bool            // 当已经运行时为true；否则为false
}
```

5.2 Entry调度实体

```go
type Entry struct {
    // The schedule on which this job should be run.
    // 负责调度当前 Entity 中的 Job 执行
    Schedule Schedule

    // The next time the job will run. This is the zero time if Cron has not been
    // started or this entry's schedule is unsatisfiable
    // Job 下一次执行的时间
    Next time.Time

    // The last time this job was run. This is the zero time if the job has never
    // been run.
    // 上一次执行时间
    Prev time.Time

    // The Job to run.
    // 要执行的 Job
    Job Job
}
```

5.3 Job

> Job：每一个实体包含一个需要运行的Job,这是一个接口，只有一个方法：`Run`

```go
type Job interface {
    Run()
}
```

由于 `Entity` 中需要 `Job` 类型，因此，我们希望定期运行的任务，就需要实现 `Job` 接口。同时，由于 `Job` 接口只有一个无参数无返回值的方法，为了使用方便，作者提供了一个类型

```go
type FuncJob func()
```

它通过简单的实现 `Run()` 方法来实现 `Job` 接口：

```go
func (f FuncJob) Run() { f() }
```

这样，任何无参数无返回值的函数，通过强制类型转换为 `FuncJob`，就可以当作 `Job` 来使用了，`AddFunc` 方法 就是这么做的。

5.4 Schedule

> 每个实体包含一个调度器（Schedule）负责调度 `Job` 的执行。它也是一个接口。 `Schedule` 的具体实现通过解析 `Cron` 表达式得到。库中提供了 `Schedule` 的两个具体实现，分别是 `SpecSchedule` 和 `ConstantDelaySchedule` 。

```go
type Schedule interface {
    // Return the next activation time, later than the given time.
    // Next is invoked initially, and then each time the job is run.
    // 返回同一 Entity 中的 Job 下一次执行的时间
```

```
    Next(time.Time) time.Time
}
```

5.4.1 SpecSchedule

从开始介绍的 Cron 表达式可以容易得知各个字段的意思，同时，对各种表达式的解析也会最终得到一个 SpecSchedule 的实例。库中的 Parse 返回的其实就是 SpecSchedule 的实例（当然也就实现了 Schedule 接口）

go 复制代码

```go
type SpecSchedule struct {
    Second, Minute, Hour, Dom, Month, Dow uint64
}
```

5.4.2 ConstantDelaySchedule

go 复制代码

```go
type ConstantDelaySchedule struct {
    Delay time.Duration // 循环的时间间隔
}
```

这是一个简单的循环调度器，如：每 5 分钟。注意，最小单位是秒，不能比秒还小，比如 毫秒。

通过 `Every` 函数可以获取该类型的实例，如下得到的是一个每 5 秒执行一次的调度器。：

go 复制代码

```go
constDelaySchedule := Every(5e9)
```

6 函数调用

6.1 实例化 Cron

实例化时，成员使用的基本是默认值；

go 复制代码

```go
func New() *Cron {
    return &Cron{
        entries:  nil,
        add:      make(chan *Entry),
        stop:     make(chan struct{}),
        snapshot: make(chan []*Entry),
        running:  false,
    }
}
```

6.2 成员方法

如果对每个方法调用还是不了解可以去看一下每个函数的实现源码

go 复制代码

```go
// EntryID identifies an entry within a Cron instance
type EntryID int

// 将 job 加入 Cron 中
// 如上所述，该方法只是简单的通过 FuncJob 类型强制转换 cmd，然后调用 AddJob 方法
func (c *Cron) AddFunc(spec string, cmd func()) (EntryID, error)

// 将 job 加入 Cron 中
// 通过 Parse 函数解析 cron 表达式 spec 的到调度器实例(Schedule)，之后调用 c.Schedule 方法
func (c *Cron) AddJob(spec string, cmd Job) (EntryID, error)

// 获取当前 Cron 总所有 Entities 的快照
```

```
func (c *Cron) Entries() []*Entry

// Location获取时区位置
func (c *Cron) Location() *time.Location

// Entry返回给定项的快照，如果找不到则返回nil
func (c *Cron) Entry(id EntryID) Entry

// 删除将来运行的条目。
func (c *Cron) Remove(id EntryID)

// 通过两个参数实例化一个 Entity，然后加入当前 Cron 中
// 注意：如果当前 Cron 未运行，则直接将该 entity 加入 Cron 中；
// 否则，通过 add 这个成员 channel 将 entity 加入正在运行的 Cron 中
func (c *Cron) Schedule(schedule Schedule, cmd Job) EntryID

// 新启动一个 goroutine 运行当前 Cron
func (c *Cron) Start()

// 通过给 stop 成员发送一个 struct{}{} 来停止当前 Cron，同时将 running 置为 false
// 从这里知道，stop 只是通知 Cron 停止，因此往 channel 发一个值即可，而不关心值是多少
// 所以，成员 stop 定义为空 struct
func (c *Cron) Stop()

// 运行cron调度程序，如果已经在运行，则不运行op
func (c *Cron) Run()
```

7 更多参考资料

源码地址

定时器API