



下载APP



14 | 定时任务：如何让框架支持分布式定时脚本？

2021-10-18 叶剑峰

《手把手带你写一个Web框架》

课程介绍 >



讲述：叶剑峰

时长 22:30 大小 20.61M



你好，我是轩脉刃。

上一节课，我们改造框架让它支持命令行工具了。这样业务开发者可以使用框架定义好的命令行工具来执行框架预设的一些行为，比如启动一个 Web 服务，也可以自己定义业务需要的命令行工具来执行业务行为，非常方便。

今天继续思考如何优化，因为业务在开发过程中，不可能每个命令都要手动操作，定时执行某个命令的需求应该是非常普遍的。比如设计一个定时扫描数据命令来发送统计报告或者设计一个定时删除某些过期数据的命令。



那我们的框架是否能支持这个需求，如果要开发一个定时命令，能不能做到在业务中增加一行代码就行了？这节课我们就来挑战这个目标。

使用 timer 定时执行命令

怎么做到计时执行这个事情呢？Golang 有个标准库 `time`，里面提供一个计时器 `timer` 的结构，是否可以使用这个 `timer` 来执行呢？我们先来看看 `timer` 是怎么使用的：

[📄 复制代码](#)

```
1 func main() {  
2     timer := time.NewTimer(3 * time.Second) // 定一个计时器，3s后触发  
3     select {  
4         now <-timer.C: // 监听计时器中的事件  
5             fmt.Println("3秒执行任务，现在时间", now) //3s后执行  
6     }  
7 }
```

首先 `time.NewTimer` 会初始化一个计时器，这个计时器到定时时间后，就从 `C` 这个 channel 中返回一个时间 `Time`。逻辑很简单。所以我们的 `main` 函数只需要监听 `timer.C` 这个 channel，一旦有时间从这个 channel 中出来，就说明到计时器时间了。

知道如何定时执行了，接下来得结合实际业务场景考虑会存在什么问题了：**如果有一堆定时任务，要怎么定时执行呢？**


这个也难不倒我们：遍历所有的定时任务，计算出这些定时任务下一次要执行的时间，然后按照从近到远排序，用 `timer` 来设置一个定时器到最近的时间触发，然后等计时一到，开启一个 Goroutine 触发执行。触发之后再进行一轮计算，计算出所有定时任务下一次要执行的时间，再初始化一个定时器。

写一下大致的伪代码如下。

假设定义每个定时任务结构为 `Entry`，先计算每个 `Entry` 的下一运行时间，接着进入循环；

在循环中，根据 `Next` 排序，然后根据最近的 `Entry` 实例化一个计时器，进入阻塞等待；

等计时器时间到了，通过 `timer.C` 获取到时间，再进行遍历，把到了时间的任务进行一次执行，并且更新所有任务的下次执行时间。

 复制代码

```
1 // Entity代表每个定时任务
2 entries := []Entry
3
4 // 计算每个定时任务时间
5 for _, entry := entries {
6     entry.Next = next(entry)
7 }
8
9 for {
10     // 根据Next时间排序
11     sortByTime(entries)
12
13     // 创建计时器
14     timer = time.NewTimer(entries.Early.Sub(time.Now()))
15
16     select {
17     case now = <-timer.C:
18         for _, entry := entries {
19             // 对已经到了时间的任务，执行
20             if entry.Next.Ok() {
21                 go startJob(entry)
22             }
23
24             // 所有任务重新计算下个timer
25             entry.Next = next(entry)
26         }
27     }
28 }
```

使用 cron 包定时执行命令

其实我们刚才对定时执行任务的分析，就是开源定时执行库 [cron](#) 的核心实现。

cron 库的作者 robfig 是一名资深的 Golang 开源贡献者，他最出名的两个作品 cron 库和 Revel 框架目前已经分别有了 8.4k 和 12.4k 的 star 数。cron 库的 Licence 是基于 MIT 的，允许商用、允许私有化，目前最新的稳定版本为 v3。

cron 库可以实现多个定时任务的执行功能，核心原理正如刚才讲的那样，底层也是使用 timer 来进行每个定时任务的计算，但是细节实现远不止这些。我们看它的用法就能感受出来：

 复制代码

```
1 // 创建一个cron实例
2 c := cron.New()
```

```
3 // 每整点30分钟执行一次
4 c.AddFunc("30 * * * *", func() {
5     fmt.Println("Every hour on the half hour")
6 })
7 // 上午3-6点，下午8-11点的30分钟执行
8 c.AddFunc("30 3-6,20-23 * * *", func() {
9     fmt.Println(".. in the range 3-6am, 8-11pm")
10 })
11 // 东京时间4:30执行一次
12 c.AddFunc("CRON_TZ=Asia/Tokyo 30 04 * * *", func() {
13     fmt.Println("Runs at 04:30 Tokyo time every day")
14 })
15 // 从现在开始每小时执行一次
16 c.AddFunc("@hourly", func() {
17     fmt.Println("Every hour, starting an hour from now")
18 })
19 // 从现在开始，每一个半小时执行一次
20 c.AddFunc("@every 1h30m", func() {
21     fmt.Println("Every hour thirty, starting an hour thirty from now")
22 })
23
24 // 启动cron
25 c.Start()
26
27 ...
28 // 在cron运行过程中增加任务
29 c.AddFunc("@daily", func() { fmt.Println("Every day") })
30
31 ..
32 // 查看运行中的任务
33 inspect(c.Entries())
34
35 ..
36 // 停止cron的运行，优雅停止，所有正在运行中的任务不会停止。
37 c.Stop()
```

这个例子充分展示了这个库的能力。

首先，它具有**极其丰富的“时间描述语言”**。我们可以通过和 Linux 的 crontab 一样的语法，定义五个星号来表示在一天中的某个时间点，时间点的维度可以是分钟、小时、日、月、星期，也可以将这五个星号替换为不同维度的值或者范围。前两个例子就很好地展示了在每小时 30 分的时候，和在上午 3-6 点、下午 8-11 点的每 30 分钟的时候执行的方法。

而第三个例子，它定义了日本东京时区的某个时间点来执行，这个已经超出了 Linux 的 crontab 能力了。看第四、五个例子，它还能通过符号 @ 和非常语义化的 hourly、every

1h30m 的描述语句，来表示从现在开始的时间计算。

或许你会好奇这种神奇的时间描述语言是怎么实现的？可以看 cron 源码的 [parser.go](#) 文件。它的本质就是将字符串解析成为一个包含秒、分、小时等时间数据的结构 SpecSchedule。看下面的部分源码：（不是今天的重点，就不解释具体的字符串解析步骤了，有兴趣的同学可以自行查看完整源码）

[复制代码](#)

```
1 // 用来表示调度时间的一个结构
2 type SpecSchedule struct {
3     // Dom表示dayOfMonth, 每个月的第几天
4     // Dow表示dayOfWeek, 每个星期的第几天
5     Second, Minute, Hour, Dom, Month, Dow uint64
6
7     // 时区
8     Location *time.Location
9 }
```

你会发现 SpecSchedule 中有秒的时间字段，是的，这是我想介绍这个库的第二个强大能力：**支持秒级别的定时**。看下面的例子：

[复制代码](#)

```
1 // 创建一个cron实例
2 c := cron.New(cron.WithParser(cron.NewParser(cron.SecondOptional | cron.Minute
3
4 // 每秒执行一次
5 c.AddFunc("* * * * *", func() {
6     fmt.Println("Every hour on the half hour")
7 })
```


在初始化实例的时候使用 NewParser，设置了允许定时字符串支持秒级别的选项 cron.SecondOptional。这样，我们可以使用 6 个星号来表示在秒级别的时候调用一次。这个功能是很多第三方库，甚至 Linux 的 crontab 都做不到的。

除了这两个强大的能力之外，cron 还支持动态增加定时任务、任务调用链等功能。不过这些功能暂时用不到，就不在这里介绍了。有兴趣的同学可以自己课后研究 cron 库的源码。

定时命令的思路与实现

我们下一步就是要考虑如何将 cron 库应用在 hade 框架中了。

从使用角度开始思考，今天一开始我们说了，挑战只增加一行代码就能定时执行某个命令。那这个代码应该有两个参数，一是设置定时时间，二是定制执行的命令。就像这样：

 复制代码

```
1 // 每秒调用一次Foo命令
2 rootCmd.AddCronCommand("* * * * *", demo.FooCommand)
```

第一个参数对应 cron 库中的“时间描述语言”，第二个参数对应我们的 Command 结构。

从刚才的两个使用小例子能看到，在 cron 库中，增加一个定时任务的 AddFunc 方法，有两个参数：时间描述语言、匿名函数。那么明显，**AddCronCommand 函数中核心要做的，就是将 Command 结构的执行封装成一个匿名函数，再调用 cron 的 AddFunc 方法就可以了。**

这里还有一个需要先解决的问题，cron 库需要初始化一个 Cron 对象，就是上面库例子中的 cron.New 方法创建的对象，这个 Cron 对象存放在哪里呢？

记得上一节课，我们将服务容器 container 存放在根 Command 中么，让所有的命令都可以通过 Root() 方法获取到根 Command，再获取到 container。那这里也可以如法炮制，将初始化的 Cron 对象放在根 Command 中。

所以，在框架目录的 framework/cobra/Command.go 中，我们对 Command 结构进行修改：

 复制代码


```
1 type Command struct {
2     // Command支持cron，只在RootCommand中有这个值
3     Cron *cron.Cron
4     // 对应Cron命令的信息
5     CronSpecs []CronSpec
6     ...
7 }
```

除了在根 Command 结构中放入 Cron 实例，我还放入了一个 CronSpecs 的数组，这个数组用来保存所有 Cron 命令的信息，为后续查看所有定时任务而准备。

思考清楚了这些，我们就能开始动手写 AddCronCommand 这个函数了。

cron 的初始化和回调

先获取根 Command，判断是否已经初始化了 Cron 实例，如果没有，就要初始化。接着，补充 Cron 命令的信息到 CronSpecs 数组中。最后封装一个匿名函数，在这个匿名函数中，我们将要执行的 Command 进行封装。

 复制代码

```
1 // AddCronCommand 是用来创建一个Cron任务的
2 func (c *Command) AddCronCommand(spec string, cmd *Command) {
3     // cron结构是挂载在根Command上的
4     root := c.Root()
5     if root.Cron == nil {
6         // 初始化cron
7         root.Cron = cron.New(cron.WithParser(cron.NewParser(cron.SecondOptional
8             root.CronSpecs = []CronSpec{}
9     }
10    // 增加说明信息
11    root.CronSpecs = append(root.CronSpecs, CronSpec{
12        Type: "normal-cron",
13        Cmd:  cmd,
14        Spec: spec,
15    })
16    // 制作一个rootCommand
17    var cronCmd Command
18    ctx := root.Context()
19    cronCmd = *cmd
20    cronCmd.args = []string{}
21    cronCmd.SetParantNull()
22    cronCmd.SetContainer(root.GetContainer())
23    // 增加调用函数
24    root.Cron.AddFunc(spec, func() {
25        // 如果后续的command出现panic，这里要捕获
26        defer func() {
27            if err := recover(); err != nil {
28                log.Println(err)
29            }
30        }()
31        err := cronCmd.ExecuteContext(ctx)
32        if err != nil {
33            // 打印出err信息
34            log.Println(err)
35        }
36    })
37}
```

```
36     }  
37     })  
38 }
```

这里有几个细节需要注意一下。

封装匿名函数的时候，要记得保证这个匿名函数不会抛出异常，因为在 cron 中，匿名函数是开启一个 Goroutine 来执行的，而在 Golang 中，每个 Goroutine 都是平等的，任何一个 Goroutine 出现 panic，都会导致整个进程中止。

另外在匿名函数中，封装的并不是传递进来的 Command，而是**把这个 Command 做了一个副本，并且将其父节点设置为空，让它自身就是一个新的根节点；然后调用这个 Command 的 Execute 方法。**


这么做主要是因为我在调试过程中发现，cobra 库在执行任意一个 Command 的 Execute 系列方法时，都会从根 Command 开始，根据参数进行遍历查询。这是因为我们是通过定时器进行调用的，这个定时器调用并没有真正的控制台，如果希望找到这个 cronCmd，直接调用其 Execute 命令就行。

所以在上面的代码里，复制了一个当前 Command 结构的 cronCmd，并且将其设置为根 Command，就是 SetParentNull() 函数，将其上游设置为空，就可以直接在定时器启动的时候调用这个 cronCmd 的 ExecuteContext 了。

启动 cron 的思路与实现

好了，现在完成了 cron 的初始化和回调，下面要解决启动 cron 的时机了。

上一节课我们将 main 函数修改为一个命令行调用，还将启动 Web 服务设计为一个子命令。那么类比到这里，启动 cron 服务也应该改造成为一个子命令。所以定义一个二级命令 cron，五个三级命令 start、stop、restart、list、state。

 复制代码

```
1  ~: ./hade cron  
2  定时任务相关命令  
3  
4  Usage:  
5    hade cron [flags]
```



```
6   hade cron [command]
7
8 Available Commands:
9   list      列出所有的定时任务
10  restart   重启cron常驻进程
11  start     启动cron常驻进程
12  state     cron常驻进程状态
13  stop      停止cron常驻进程
14
15 Flags:
16   -h, --help   help for cron
17
18 Use "hade cron [command] --help" for more information about a command.
19
```


这五个三级命令中最核心的就是启动 cron 的命令，我们就详细讲解这个命令，其他的大同小异，你可以课后对照 GitHub 补充。

在 `./hade cron start` 这个命令中，必须要考虑的有两个问题：

首先，**我们希望使用 cron 的三级命令对某个进程进行管理**。想达到这个目的，获取这个被管理进程的 PID，并放入指定文件中非常重要，只有这样，才能在不同命令之间，通过获取这个 PID，来查询或者停止这个进程。

解决方案倒不难。获取当前进程，我们可以使用标准库 `osos.GetPid()`。而 PID 的存放目录，之前在第 12 节课中定义服务目录的时候，有一个 `runtimeFolder` 是存放当前运行状态文件的，可以把 cron 进程的 PID 存放在这个目录中。

其次，在启动常驻进程的时候，我们可能以直接挂起进程的方式启动，也可能以后台 `daemon` 的方式启动，所以对于 **start 的命令最好能支持两种形态**。这两种形态可以通过一个参数进行区分，在文件 `framework/command/cron.go` 中增加 `daemon` 参数：

 复制代码

```
1 // start命令有一个daemon参数，简写为d
2 cronStartCommand.Flags().BoolVarP(&cronDaemon, "daemon", "d", false, "start se
```

直接挂起的逻辑比较简单，就是直接调用 cron 的 Run 函数：

```
1 c.Root().Cron.Run()
```

那么如何以 daemon 方式后台运行一个命令呢？或许你的第一反应是直接进行系统调用 fork 行不行？但是要告诉你，不行。

因为在 Golang 中，fork 可以启动一个子进程，但是这个子进程是无法继承父进程的调度模型的。**Golang 的调度模型是在用户态的 runtime 中自己进行调度的，而系统调用 fork 出来的子进程默认只会有单线程。**所以在 Golang 中尽量不要使用 fork 的方式来复制启动当前进程。

这里，实际上希望能 fork 出一个同样有运行到当前代码 Go 环境的进程。

另一个办法是使用 os.StartProcess 来启动一个进程，执行当前进程相同的二进制文件以及当前进程相同的参数。同时，由于二进制文件相同，还要为启动的子进程单独配置一个环境变量，这样在生成二进制文件的代码中，就能根据环境变量区分是主进程还是子进程。

这里整个启动子进程的方法在开源社区已经有封装好的了——开源库 [go-deamon](#)。这个开源库目前 star 数有 1.5k，采用 MIT 的 licence，是现在比较流行的将进程 daemon 化的库了。它的原理和上面分析的是一样的，我们看下使用思路。


这个库会初始化一个 daemon.Context 结构，在这个结构中可以设置子进程的参数、环境变量、PID 生成的文件、输出日志生成的文件、权限等。然后调用一个 Reborn 方法启动一个子进程，这个 Reborn 除了 error 之外，还有一个返回值 os.Process，如果非空，代表当前为父进程，能从这个方法获取子进程信息，如果为空，代表当前为子进程。

启动 cron 的实现

现在启动 cron 命令的两个实现关键点都考虑清楚了。

结合我们当前的需求，要启动一个子进程，命令为：`./hade cron start --daemon=true`，这个子进程会运行 `cron.Run`，子进程 PID 写入 `runtimeFolder` 目录下，子进程日志打印在 `logFolder` 目录下。而父进程打印成功信息，直接返回，不做任何操作。

代码 framework/command/cron.go 中的 cronStartCommand 核心逻辑如下：


 复制代码

```
1 // daemon 模式
2 if cronDaemon {
3     // 创建一个Context
4     cntxt := &daemon.Context{
5         // 设置pid文件
6         PidFileName: serverPidFile,
7         PidFilePerm: 0664,
8         // 设置日志文件
9         LogFileName: serverLogFile,
10        LogFilePerm: 0640,
11        // 设置工作路径
12        WorkDir: currentFolder,
13        // 设置所有设置文件的mask，默认为750
14        Umask: 027,
15        // 子进程的参数，按照这个参数设置，子进程的命令为 ./hade cron start --daemon=true
16        Args: []string{"", "cron", "start", "--daemon=true"},
17    }
18    // 启动子进程，d不为空表示当前是父进程，d为空表示当前是子进程
19    d, err := cntxt.Reborn()
20    if err != nil {
21        return err
22    }
23    if d != nil {
24        // 父进程直接打印启动成功信息，不做任何操作
25        fmt.Println("cron serve started, pid:", d.Pid)
26        fmt.Println("log file:", serverLogFile)
27        return nil
28    }
29    // 子进程执行Cron.Run
30    defer cntxt.Release()
31    fmt.Println("daemon started")
32    gspt.SetProcTitle("hade cron")
33    c.Root().Cron.Run()
34    return nil
35 }
```

整体实现可以参考 GitHub 中当前章节的分支。


下面验证一下前面对定时任务的改造是否成功。先在业务

app/console/command/demo/foo.go 中创建一个 Foo 命令，它的执行行为就是打印一行字符 “execute foo command”：

 复制代码

```
1 // FooCommand 代表Foo命令
2 var FooCommand = &cobra.Command{
3     Use:      "foo",
4     Short:    "foo的简要说明",
5     Long:     "foo的长说明",
6     Aliases: []string{"fo", "f"},
7     Example:  "foo命令的例子",
8     RunE: func(c *cobra.Command, args []string) error {
9         log.Println("execute foo command")
10        return nil
11    },
12 }
```

然后将这个 FooCommand，通过 AddCronCommand 绑定到根 Command 中，并且设置其调用时间为每秒调用一次。

 复制代码

```
1 // 绑定业务的命令
2 func AddAppCommand(rootCmd *cobra.Command) {
3     // 每秒调用一次Foo命令
4     rootCmd.AddCronCommand("* * * * *", demo.FooCommand)
5 }
```

启动三级命令行工具 ./hade cron start -d，看到控制台打印出了子进程 PID 信息，和日志存储地址。

```
~/Documents/UGit/coredemo > geekbang/14 ● ./hade cron start -d
cron serve started, pid: 32426
log file: /Users/yejianfeng/Documents/UGit/coredemo/storage/log/cron.log
```

使用 tail 命令查看日志，可以看到确实是每秒执行打印一次字符串：execute foo command

```
~/Documents/UGit/coredemo > geekbang/14 ● tail -f /Users/yejianfeng/Documents/UGit/coredemo/storage/log/cron.log
2021/09/12 00:28:09 execute foo command
2021/09/12 00:28:10 execute foo command
2021/09/12 00:28:11 execute foo command
2021/09/12 00:28:12 execute foo command
2021/09/12 00:28:13 execute foo command
2021/09/12 00:28:14 execute foo command
```

到这里，我们对框架的定时命令改造就完成了。

如何实现分布式定时器

但现在的定时器还是单机版本的定时器，容灾性很低，如果有很多定时任务都挂载在一个进程中，一旦这个进程或者这个机器出现灾难性不可恢复，那么定时任务就直接无法运行了。


容灾性更高的是分布式定时器。也就是很多机器都同时挂载定时任务，在同一时间都启动任务，只有一台机器能抢占到这个定时任务并且执行，其他机器由于抢占不到定时任务，不执行任何操作。

我们的框架也能做到这个分布式定时器的功能。

这种分布式定时器如何实现呢？你第一个想到的是不是使用 Redis？我们用 Redis 做一个分布式锁，然后所有进程去抢占这个锁。

还是这样思考问题就比较低层次了。在第十、十一章，我们花了大量的篇幅讲了面向接口编程的思想，这里就可以用到这个思想。**先定义接口，这个接口的功能是一个分布式的选择器，当有很多节点要执行某个服务的时候，只选择出其中一个节点。**这样不管底层是否用 Redis 实现分布式选择器，在业务层我们都可以不用关心。

在文件 `framework/contract/distributed.go` 文件中。我们先定义接口 `Distributed`。其中有一个分布式选举方法 `Select`。它的参数有三个，`serviceName` 代表服务名字、`appId` 代表节点的 ID、`holdTime` 表示这个选择结果持续多久，也就是在选举出来之后多久内有效。它返回两个值，`selectAppID` 表示选举的结果，即最终哪个节点被选举出来了，另一个返回值 `error` 表示异常。

 复制代码

```
1 package contract
2
3 import "time"
4
5 // DistributedKey 定义字符串凭证
6 const DistributedKey = "hade:distributed"
7
8 // Distributed 分布式服务
9 type Distributed interface {
```

```

10 // Select 分布式选择器，所有节点对某个服务进行抢占，只选择其中一个节点
11 // ServiceName 服务名字
12 // appID 当前的AppID
13 // holdTime 分布式选择器hold住的时间
14 // 返回值
15 // selectAppID 分布式选择器最终选择的App
16 // err 异常才返回，如果没有被选择，不返回err
17 Select(serviceName string, appID string, holdTime time.Duration) (selectApp
18 }
19

```

这里提到了一个 appID 是我们之前没有见过的，简单说明一下。appID 是为每个当前应用起的唯一标识，它用 Google 的 [uuid 生成库](#) 就可以很方便生成。它怎么通过服务容器获取呢？我们可以将 appID 作为服务容器中 App 服务的一个方法，这样就能从服务容器中获取 App 服务，再获取到 appID 了。

所以这里插一步来修改对应的 App 接口和其对应实现 HadeApp，详细的修改点都以注释的形式写在下面代码中了。

在 framework/contract/framework/contract/app.go 中增加 AppID 的接口函数：

[复制代码](#)

```

1 // App 定义接口
2 type App interface {
3     // AppID 表示当前这个app的唯一id，可以用于分布式锁等
4     AppID() string
5     ...
6 }

```

在 framework/provider/app/service.go 中也增加对 AppID 的实现：

[复制代码](#)

```

1 // HadeApp 代表hade框架的App实现
2 type HadeApp struct {
3     ...
4     appId string // 表示当前这个app的唯一id，可以用于分布式锁等
5 }
6
7 // NewHadeApp 初始化HadeApp
8 func NewHadeApp(params ...interface{}) (interface{}, error) {
9     ...

```



```
10     appId := uuid.New().String()
11     return &HadeApp{baseFolder: baseFolder, container: container, appId: appId}
12 }
13
14 // AppID 表示这个App的唯一ID
15 func (h HadeApp) AppID() string {
16     return h.appId
17 }
18
```

分布式服务 Distributed 的接口定义好，我们再回到它的具体实现。

这个具体实现就有很多方式了，Redis 只是其中一种而已，而且 Redis 要到后面章节才能引入。这里就实现一个本地文件锁。当一个服务器上有多个进程需要进行抢锁操作，文件锁是一种单机多进程抢占的很简易的实现方式。在 Golang 中，其使用方法也比较简单。

多个进程同时使用 `os.OpenFile` 打开一个文件，并使用 `syscall.Flock` 带上 `syscall.LOCK_EX` 参数来对这个文件加文件锁，这里只会有一个进程抢占到文件锁，而其他抢占不到的进程从 `syscall.Flock` 函数中获取到的就是 `error`。**根据这个 `error` 是否为空，我们就能判断是否抢占到了文件锁。**

释放文件锁有两种方式，一种方式是调用 `syscall.Flock` 带上 `syscall.LOCK_UN` 的参数，另外一种方式是抢占到锁的进程结束，也会自动释放文件锁。

来看分布式选择器的本地文件锁的具体实现，在代码 `framework/provider/distributed/service_local.go` 文件中：

```
1 // Select 为分布式选择器
2 func (s LocalDistributedService) Select(serviceName string, appId string, hold
3     appService := s.container.MustMake(contract.AppKey).(contract.App)
4     runtimeFolder := appService.RuntimeFolder()
5     lockFile := filepath.Join(runtimeFolder, "disribute_"+serviceName)
6     // 打开文件锁
7     lock, err := os.OpenFile(lockFile, os.O_RDWR|os.O_CREATE, 0666)
8     if err != nil {
9         return "", err
10    }
11    // 尝试独占文件锁
12    err = syscall.Flock(int(lock.Fd()), syscall.LOCK_EX|syscall.LOCK_NB)
13    // 抢不到文件锁
14    if err != nil {
```

[复制代码](#)

```
15 // 读取被选择的appid
16 selectAppIDByt, err := ioutil.ReadAll(lock)
17 if err != nil {
18     return "", err
19 }
20 return string(selectAppIDByt), err
21 }
22 // 在一段时间内，选举有效，其他节点在这段时间不能再进行抢占
23 go func() {
24     defer func() {
25         // 释放文件锁
26         syscall.Flock(int(lock.Fd()), syscall.LOCK_UN)
27         // 释放文件
28         lock.Close()
29         // 删除文件锁对应的文件
30         os.Remove(lockFile)
31     }()
32     // 创建选举结果有效的计时器
33     timer := time.NewTimer(holdTime)
34     // 等待计时器结束
35     <-timer.C
36 }()
37 // 这里已经是抢占到了，将抢占到的appID写入文件
38 if _, err := lock.WriteString(appID); err != nil {
39     return "", err
40 }
41 return appID, nil
42 }
```

大概逻辑是先打开或者创建文件锁对应的文件，然后在这个文件上加上文件锁，加上锁的过程就是抢占的过程。对于没有抢占到的，文件中内容为抢占到的那个应用的 ID，将应用 ID 返回；抢占到的，就写入自己的应用 ID 到文件中，并且通过一个新的 Goroutine 开启计时器，等待计时器结束后解开文件锁，并且删除文件锁对应的文件。

而这个分布式服务 Distributed 的服务提供者

framework/provider/distributed/provider_local.go 的逻辑就比较简单了，就是正常的实现 serviceProvider 的 5 个方法，这里就不赘述了，可以参考 GitHub 上的代码分支。

现在有了分布式选择器，就可以实现分布式调度了。我们为 Command 结构增加一个方法 **AddDistributedCronCommand**，这个方法有四个参数：


serviceName 这个服务的唯一名字，不允许带有空格

spec 具体的执行时间

cmd 具体的执行命令

holdTime 表示如果我选择上了，这次选择持续的时间也就是锁释放的时间

它的实现和 AddCronCommand 差不多，唯一的区别就是在封装 cron.AddFunc 的匿名函数中，运行目标命令之前，要做一次分布式选举，如果被选举上了，才执行目标命令。所以来增加一个文件，在 framework/cobra/hade_command_distributed.go 中，定义 AddDistributedCronCommand 方法：

 复制代码

```

1 // AddDistributedCronCommand 实现一个分布式定时器
2 // serviceName 这个服务的唯一名字，不允许带有空格
3 // spec 具体的执行时间
4 // cmd 具体的执行命令
5 // holdTime 表示如果我选择上了，这次选择持续的时间，也就是锁释放的时间
6 func (c *Command) AddDistributedCronCommand(serviceName string, spec string, c
7     root := c.Root()
8     // 初始化cron
9     if root.Cron == nil {
10         root.Cron = cron.New(cron.WithParser(cron.NewParser(cron.SecondOptional
11             root.CronSpecs = []CronSpec{})
12     }
13     // cron命令的注释，这里注意Type为distributed-cron，ServiceName需要填写
14     root.CronSpecs = append(root.CronSpecs, CronSpec{
15         Type:         "distributed-cron",
16         Cmd:          cmd,
17         Spec:         spec,
18         ServiceName:  serviceName,
19     })
20     appService := root.GetContainer().MustMake(contract.AppKey).(contract.App)
21     distributeService := root.GetContainer().MustMake(contract.DistributedKey).(
22     appID := appService.AppID()
23     // 复制要执行的command为cronCmd，并且设置为rootCmd
24     var cronCmd Command
25     ctx := root.Context()
26     cronCmd = *cmd
27     cronCmd.args = []string{}
28     cronCmd.SetParentNull()
29     // cron增加匿名函数
30     root.Cron.AddFunc(spec, func() {
31         // 防止panic
32         defer func() {
33             if err := recover(); err != nil {
34                 log.Println(err)
35             }
36         }()
37         // 节点进行选举，返回选举结果
38         selectedAppID, err := distributeService.Select(serviceName, appID, holdTi

```

```
39     if err != nil {
40         return
41     }
42     // 如果自己没有被选择到，直接返回
43     if selectedAppID != appID {
44         return
45     }
46     // 如果自己被选择到了，执行这个定时任务
47     err = cronCmd.ExecuteContext(ctx)
48     if err != nil {
49         log.Println(err)
50     }
51 })
52 }
53
```

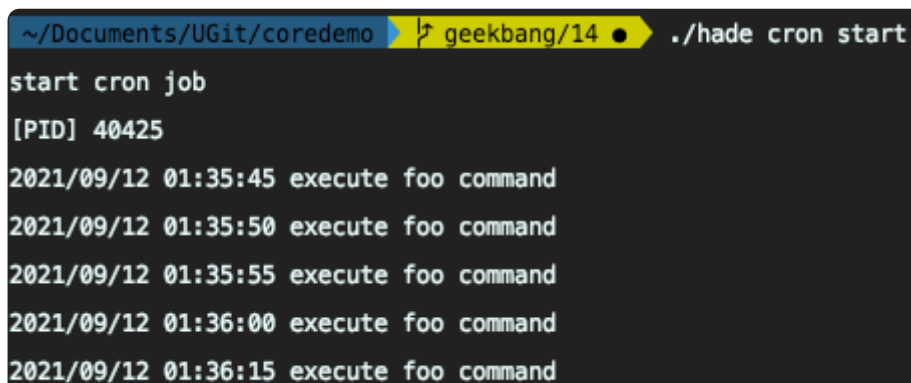
完成了实现逻辑，下面照例做验证。

我们将打印字符 “execute foo command” 的 FooCommand 命令执行的服务名设置为 “foo_func_for_test”。设置为每 5 秒进行一次选举并产生结果，每次选举结果保留 2 秒钟：

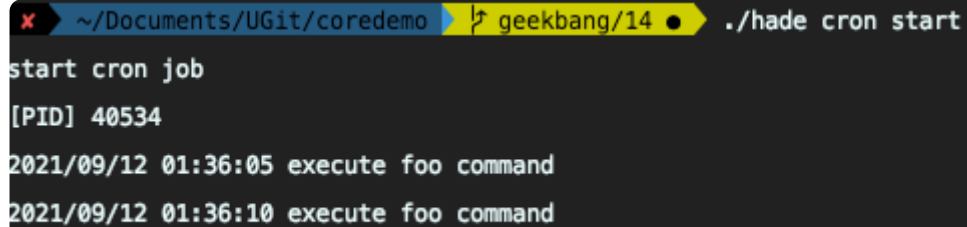
[复制代码](#)

```
1 // 绑定业务的命令
2 func AddAppCommand(rootCmd *cobra.Command) {
3
4     // 启动一个分布式任务调度，调度的服务名称为init_func_for_test，每个节点每5s调用一次Fo
5     rootCmd.AddDistributedCronCommand("foo_func_for_test", "* /5 * * * *", dem
6 }
```

开启两个控制台，启动两个进程 `./hade cron start`。



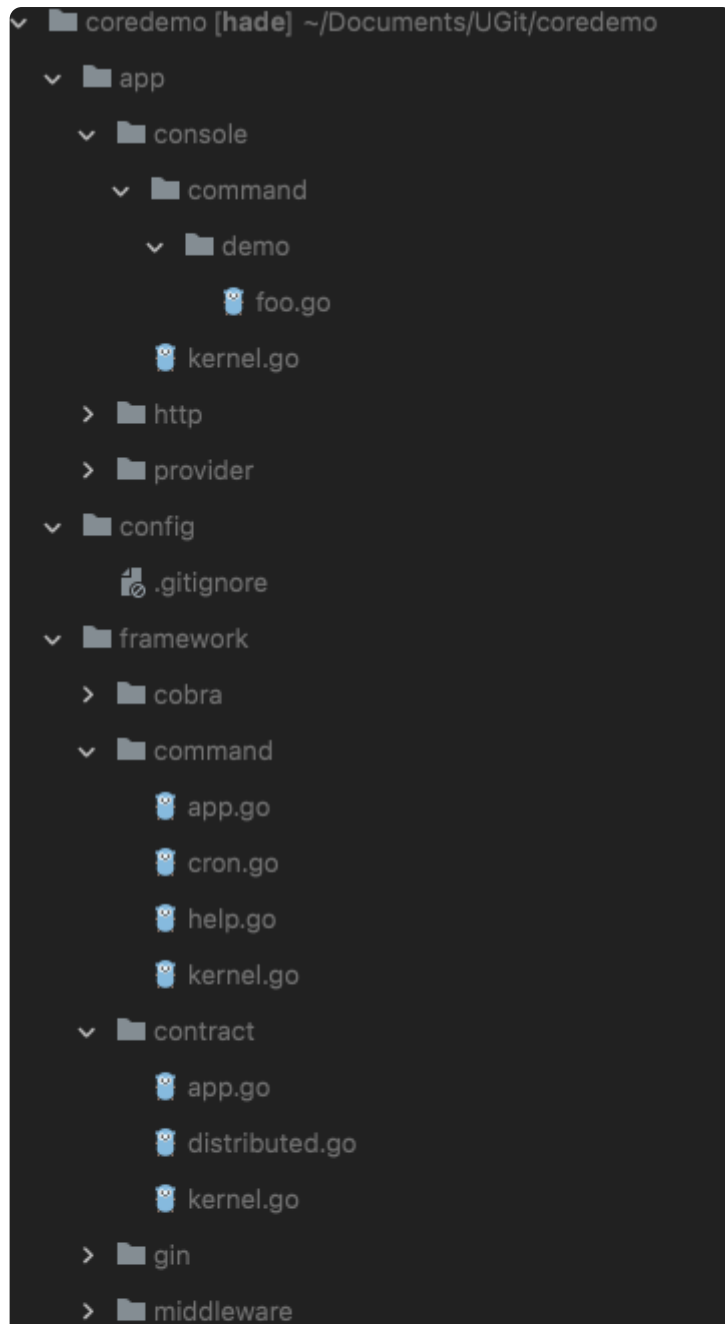
```
~/Documents/UGit/coredemo geekbang/14 ● ./hade cron start
start cron job
[PID] 40425
2021/09/12 01:35:45 execute foo command
2021/09/12 01:35:50 execute foo command
2021/09/12 01:35:55 execute foo command
2021/09/12 01:36:00 execute foo command
2021/09/12 01:36:15 execute foo command
```

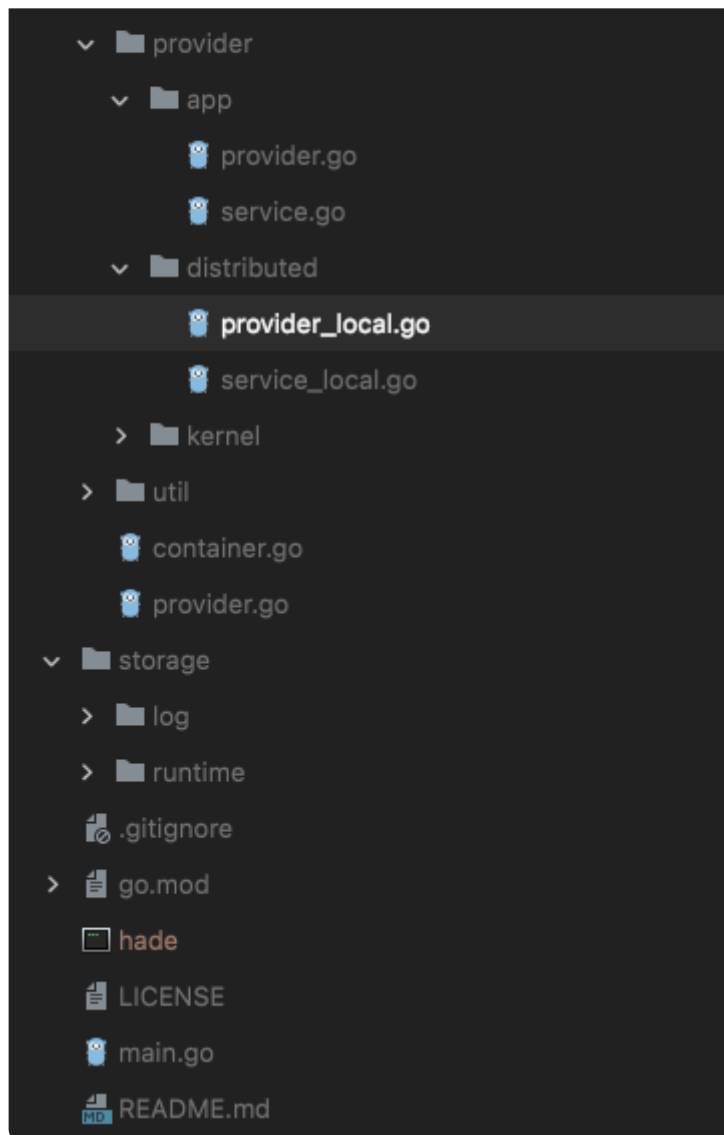


```
~/Documents/UGit/coredemo geekbang/14 ● ./hade cron start
start cron job
[PID] 40534
2021/09/12 01:36:05 execute foo command
2021/09/12 01:36:10 execute foo command
```

能看到 36 分 00 秒的任务是在 PID 为 40425 的进程执行，而 36 分 05 秒的任务是在 PID 为 40653 的进程执行。而且仿造容灾演练，关闭掉其中任何一个进程，剩余每 5 秒钟执行的任务，都会落在存活的那个进程中。

到这里，我们就为 hade 框架提供了分布式定时器的功能。这节课的所有代码都在 GitHub 上的 [geekbang/14](#) 分支。对应的目录结构截图如下：





小结

今天内容有点多，对不太清楚的地方，你可以多看几遍慢慢理解。

我们先使用 cron 包来为框架增加了定时执行命令的功能。在实现过程中，介绍了在 Golang 中，如何启动一个当前二进制文件的子进程。这个启动子进程的方式非常重要，后续在多个命令行工具中还会使用到，得熟练掌握。

接着一起实现了分布式定时器的功能。在实现分布式定时器的功能中，你也能更深一步感受到前几节课说的**面向接口编程和服务容器设计的好处**。比如说我们实现分布式定时器需要 appID，就去 app 服务中增加一个获取 AppID 的接口；需要分布式选举服务，就创建了一个分布式服务接口。非常方便。

思考题

本节课实现了 cron 的五个三级命令其中最重要的一个 cron start。你可以思考并尝试写一下其他几个命令：

cron stop 停止启动的进程

cron state 判断当前 cron 进程是否已经启动，返回 PID

cron restart 重新启动 cron 进程

cron list 列出挂载的所有 cron 任务

欢迎在留言区分享你的思考。如果你觉得有收获，也欢迎把今天的内容分享给身边的朋友，邀他一起学习。我们下节课见。

分享给需要的人，Ta订阅后你可得 **20** 元现金奖励



生成海报并分享

👍 赞 0

💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | 交互：可以执行命令行的框架才是好框架

技术管理案例课

踩坑复盘+案例分析+精进攻略=高效管理

许健

eBay 基础架构工程研发总监



新版升级：点击「🔗 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言

💬 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。