

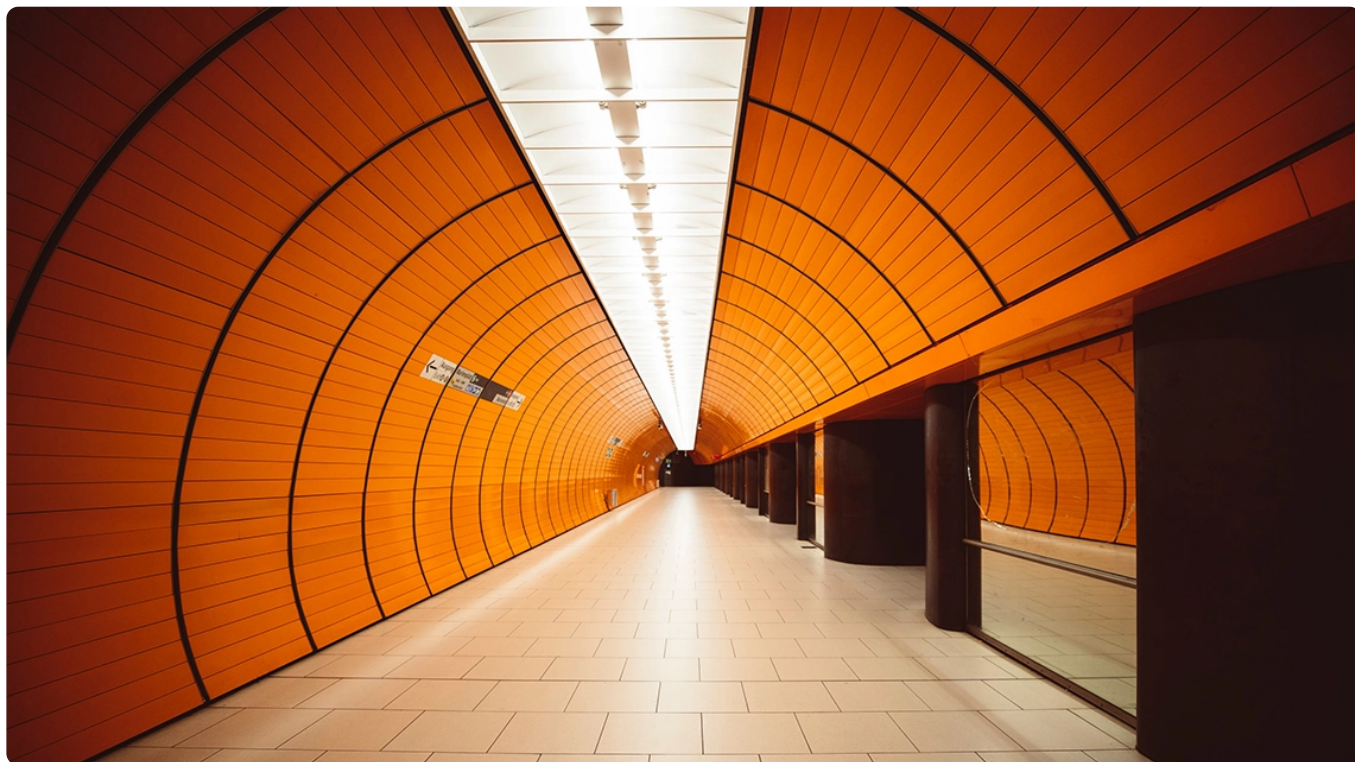


## 24 | 管理进程：如何设计完善的运行命令？

2021-11-10 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 15:38 大小 14.33M



你好，我是轩脉刃。

在 [第 13 章](#) 我们引入命令行的时候，将 Web 启动方式改成了一个命令行。但是当时只完成了一个最简单的启动 Web 服务的命令，这节课，我们要做的是完善这个 Web 服务运行命令，让 Web 服务的运行有完整的启动、停止、重启、查询的进程管理功能。

这套完整的进程管理功能，能让应用管理者非常方便地通过一套命令来统一管控一个应用，降低应用管理者的管理成本，后续也能为实现应用自动化部署到远端服务的工具提<sup>144</sup>了基础。下面我们来具体看下如何设计这套命令并且实现它吧。



### 运行命令的设计

首先照惯例需要设计一下运行命令，一级命令为 app，二级命令设计如下：

```
./hade app start 二级命令，启动一个 app 服务  
./hade app state 二级命令，获取启动的 app 的信息  
./hade app stop 二级命令，停止已经启动的 app 服务  
./hade app restart 二级命令，重新启动一个 app 服务
```


这四个二级命令，有 app 服务的启动、停止、重启、查询，基本上已经把 app 服务启动的状态变更都包含了，能基本满足后面我们对于一个应用的管理需求。下面来讨论下每个命令的功能和设计。

## 启动命令

首先是 start 这个命令，写在 framework/command/app.go 中。我们先分析下参数。

想要启动 app 服务，至少需要一个参数，就是**启动服务的监听地址**。如何获取呢？首先可以直接从默认配置获取，另外因为这是一个控制台命令，也一定可以直接从命令行获取。除了这两种方式，我们回顾下之前的配置项获取方法，还有环境变量和配置项。

所以总结起来，环境变量这个参数我们设计为有四个方式可以获取，一个是直接从命令行参数获取 address 参数，二是从环境变量 ADDRESS 中获取，然后是从配置文件中获取配置项 app.address，最后如果以上三个方式都没有设置，就使用默认值:8888。关键的代码逻辑如下：

 复制代码

```
1 if appAddress == "" {  
2     envService := container.MustMake(contract.EnvKey).(contract.Env)  
3     if envService.Get("ADDRESS") != "" {  
4         appAddress = envService.Get("ADDRESS")  
5     } else {  
6         configService := container.MustMake(contract.ConfigKey).(contract.ConfigService)  
7         if configService.IsExist("app.address") {  
8             appAddress = configService.GetString("app.address")  
9         } else {  
10            appAddress = ":8888"  
11        }  
12    }  
13 }
```

除了监听地址的参数，回忆之前 cron 命令运行的时候，启动 app 服务，我们是两种启动方式的，一种是启动后直接挂在控制台，这种启动方式适合调试开发使用；而另外一种，以守护进程 daemon 的方式启动，直接挂载在后台。所以，对于这两种启动方式，我们也需要有一个参数 daemon，标记是使用哪种方式启动。

有了 appAddress、daemon 这两个参数，我们顺着继续想**启动服务时需要的记录文件**。

不管是使用挂载方式，还是 daemon 方式启动进程，都能获取到一个进程 PID，启动 app 服务的时候，要将这个 PID 记录在一个文件中，这里我们就存储在 app/storage/runtime/app.pid 文件中。在运行时候，需要保证这个目录和文件是存在的。

同时也会产生日志，日志存放在 app/storage/log/app.log 中，所以我们要确认这个目录是否存在。

关于 app.pid 和 app.log 对应的代码：


[复制代码](#)

```
1 appService := container.MustMake(contract.AppKey).(contract.App)
2
3 pidFolder := appService.RuntimeFolder()
4 if !util.Exists(pidFolder) {
5     if err := os.MkdirAll(pidFolder, os.ModePerm); err != nil {
6         return err
7     }
8 }
9 serverPidFile := filepath.Join(pidFolder, "app.pid")
10 logFolder := appService.LogFolder()
11 if !util.Exists(logFolder) {
12     if err := os.MkdirAll(logFolder, os.ModePerm); err != nil {
13         return err
14     }
15 }
16 // 应用日志
17 serverLogFile := filepath.Join(logFolder, "app.log")
18 currentFolder := util.GetExecDirectory()
```

好到这里，准备工作都做好了，我们看看 Web 服务的启动，逻辑和之前设计的基本上没有什么区别，使用 net/http 来启动一个 Web 服务。

**重点是启动的时候注意设置优雅关闭机制。**先使用 [第六章](#)实现的优雅关闭机制：开启一个 Goroutine 启动服务，主 Goroutine 监听信号，当获取到信号之后，等待所有请求都结束或者超过最长等待时长，就结束信号。当然，这里的最长等待时长可以设置为配置项，从 app.close\_wait 配置项中获取，如果没有配置项，我们默认使用 5s 的最长等待时长。

启动相关代码：

 复制代码

```
1 // 启动AppServer，这个函数会将当前goroutine阻塞
2 func startAppServe(server *http.Server, c framework.Container) error {
3     // 这个goroutine是启动服务的goroutine
4     go func() {
5         server.ListenAndServe()
6     }()
7
8     // 当前的goroutine等待信号量
9     quit := make(chan os.Signal)
10    // 监控信号：SIGINT, SIGTERM, SIGQUIT
11    signal.Notify(quit, syscall.SIGINT, syscall.SIGTERM, syscall.SIGQUIT)
12    // 这里会阻塞当前goroutine等待信号
13    <-quit
14
15    // 调用Server.Shutdown graceful结束
16    closeWait := 5
17    configService := c.MustMake(contract.ConfigKey).(contract.Config)
18    if configService.IsExist("app.close_wait") {
19        closeWait = configService.GetInt("app.close_wait")
20    }
21    timeoutCtx, cancel := context.WithTimeout(context.Background(), time.Duration(closeWait)*time.Second)
22    defer cancel()
23
24    if err := server.Shutdown(timeoutCtx); err != nil {
25        return err
26    }
27    return nil
28 }
```

但是这里还出现了一个问题，挂在控制台的启动，比较简单，直接调用封装好的 startAppServe 就行了。但 daemon 方式如何启动呢？它是不能直接在主进程中调用

startAppServe 方法的，会把主进程给阻塞挂起来了，怎么办呢？

这个其实在 [第十四章](#) 定时任务中有说到，我们可以使用和定时任务一样的实现机制，使用开源库 [go-daemon](#)。比较重要，所以这里再啰嗦一下，**理解 go-daemon 库的使用，要理解最核心的 daemon.Context 结构。**

在我们框架这个需求中，daemon 方式启动命令为 `./hade app start --daemon=true`。所以在 `daemon.Context` 结构中的 `Args` 参数填写如下：

[复制代码](#)

```
1 // 创建一个Context
2 cntxt := &daemon.Context{
3     ...
4     // 子进程的参数，按照这个参数设置，子进程的命令为 ./hade app start --daemon=true
5     Args: []string{"", "app", "start", "--daemon=true"},
6 }
7 // 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
8 d, err := cntxt.Reborn()
9
10 if d != nil {
11     // 父进程直接打印启动成功信息，不做任何操作
12     fmt.Println("app启动成功, pid:", d.Pid)
13     fmt.Println("日志文件:", serverLogFile)
14     return nil
15 }
16 ...
```

有的同学对这个启动子进程的 `Reborn` 可能有些疑惑。

我们把 `Reborn` 理解成 `fork`，当调用这个函数的时候，父进程会继续往下走，但是返回值 `d` 不为空，它的信息是子进程的进程号等信息。而子进程会重新运行对应的命令，再次进入到 `Reborn` 函数的时候，返回的 `d` 就为 `nil`。所以在 **`Reborn` 的后面，我们让父进程直接 `return`，而让子进程继续往后进行操作，这样就达到了 `fork` 一个子进程的效果了。**

理解了这一点，对应的代码就很简单了：

[复制代码](#)

```
1 // daemon 模式
2 if appDaemon {
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 app start --daemon=tr
16        Args: []string{"", "app", "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("app启动成功, pid:", d.Pid)
26        fmt.Println("日志文件:", serverLogFile)
27        return nil
28    }
29    defer cntxt.Release()
30    // 子进程执行真正的app启动操作
31    fmt.Println("daemon started")
32    gspt.SetProcTitle("hade app")
33    if err := startAppServe(server, container); err != nil {
34        fmt.Println(err)
35    }
36    return nil
37 }
```

到这里服务的进程启动成功，最后还有一点细节，对于启动的进程，我们一般都希望能自定义它的进程名称。

这里可以使用一个第三方库 [gspt](#)。它使用 MIT 协议，虽然 star 数不多，但是我个人亲测是功能齐全且有效的。在 Golang 中没有现成的设置进程名称的方法，只能调用 C 的设置进程名称的方法 `setproctitle`。所以这个库使用的方式是，使用 `cgo` 从 Go 中调用 C 的方法来实现进程名称的修改。

它的使用非常简单，就是一个函数 `SetProcTitle` 方法：



```
1 gspt.SetProcTitle("hade app")
```

现在，进程的启动就基本完成了。当然最后还有非常重要的关闭逻辑也记得加上。

好了，以上我们讨论了 start 的关键设计，再回头梳理一遍这个命令的实现步骤：

从四个方式获取参数 appAddress

获取参数 daemon

确认 runtime 目录和 PID 文件存在

确认 log 目录的 log 文件存在

判断是否是 daemon 方式。如果是，就使用 go-daemon 来启动一个子进程；如果不是，直接进行后续调用

使用 gspt 来设置当前进程名称

启动 app 服务

具体的实现步骤相信你已经很清楚了，完整代码我们写在 [framework/command/app.go](#) 中了。

## 获取进程


已经完成了启动进程的命令，那么第二个获取进程 PID 的命令就非常简单了。因为启动命令的时候创建了一个 PID 文件，app/storage/runtime/app.pid，读取这个文件就可以获取到进程的 PID 信息了。

但是这里我们可以更谨慎一些加一步，获取到 PID 之后，去操作系统中查询这个 PID 的进程是否存在，存在的话，就确定这个 PID 是可行的。

如何根据 PID 查询一个进程是否存在呢？常用的比如 Linux 的 ps 和 grep 命令，基本上都是通过 Linux 的其他命令来检查输出，**但最为可靠的方式是直接使用信号对接要查询的进程：通过给进程发送信号来检测，这个信号就是信号 0。**

给进程发送信号 0 之后什么都不会操作，如果进程存在，不返回错误信息；如果进程不存在，会返回不存在进程的错误信息。在 Golang 中，我们可以用 os 库的 Process 结构来发送信号。

代码在 framework/util/exec.go 中，逻辑也很清晰，先用 os.FindProcess 来获取这个 PID 对应的进程，然后给进程发送 signal 0，如果返回 nil，代表进程存在，否则进程不存在。

 复制代码

```
1 // CheckProcessExist 检查进程pid是否存在，如果存在的话，返回true
2 func CheckProcessExist(pid int) bool {
3     // 查询这个pid
4     process, err := os.FindProcess(pid)
5     if err != nil {
6         return false
7     }
8
9     // 给进程发送signal 0，如果返回nil，代表进程存在，否则进程不存在
10    err = process.Signal(syscall.Signal(0))
11    if err != nil {
12        return false
13    }
14    return true
15 }
```


这个关键函数实现之后，其他的就很容易了。

这里我们也简单说一下进程获取的具体步骤：获取 PID 文件内容之后，做判断，如果有 PID 文件且有内容就继续，否则返回无进程；然后：

将内容转换为 PID 的 int 类型，转换失败视为无进程；

**使用 signal 0 确认这个进程是否存在，存在返回结果有进程，不存在返回结构无进程。**

具体代码如下，存放在 framework/command/app.go 文件中：

 复制代码

```
1 // 获取启动的app的pid
2 var appStateCommand = &cobra.Command{
3     Use:    "state",
```



```
4 Short: "获取启动的app的pid",
5 RunE: func(c *cobra.Command, args []string) error {
6     container := c.GetContainer()
7     appService := container.MustMake(contract.AppKey).(contract.App)
8
9     // 获取pid
10    serverPidFile := filepath.Join(appService.RuntimeFolder(), "app.pid")
11
12    content, err := ioutil.ReadFile(serverPidFile)
13    if err != nil {
14        return err
15    }
16
17    if content != nil && len(content) > 0 {
18        pid, err := strconv.Atoi(string(content))
19        if err != nil {
20            return err
21        }
22        if util.CheckProcessExist(pid) {
23            fmt.Println("app服务已经启动, pid:", pid)
24            return nil
25        }
26    }
27    fmt.Println("没有app服务存在")
28    return nil
29 },
30 }
```

## 停止命令

命令的启动和获取完成了，就到了第三个停止命令了。既然有了进程号，需要停止一个进程，我们还是可以使用第六章说的信号量方法，回顾下当时说的四个关闭信号：

信号	操作	是否可处理
SIGINT	Ctrl+C	可以
SIGQUIT	Ctrl+\	可以
SIGTERM	kill	可以
SIGKILL	kill -9	不可以



由于启动进程监听了 SIGINT、SIGQUIT、SIGTERM 这三个信号，所以我们在这三个信号中选取一个发送给 PID 所在的进程即可，这里就选择更符合“关闭”语义的 SIGTERM 信号。

同样实现步骤也很清晰，获取 PID 文件内容之后，判断如果有 PID 文件且有内容再继续，否则什么都不做，之后就是：

将内容转换为 PID 的 int 类型，转换失败则什么都不做

直接给这个 PID 进程发送 SIGTERM 信号

将 PID 文件内容清空

对应代码同样在 framework/command/app.go 中：

复制代码

```
1 // 停止一个已经启动的app服务
2 var appStopCommand = &cobra.Command{
3     Use:     "stop",
4     Short:   "停止一个已经启动的app服务",
5     RunE: func(c *cobra.Command, args []string) error {
6         container := c.GetContainer()
7         appService := container.MustMake(contract.AppKey).(contract.App)
8
9         // GetPid
```

```
10     serverPidFile := filepath.Join(appService.RuntimeFolder(), "app.pid")
11
12     content, err := ioutil.ReadFile(serverPidFile)
13     if err != nil {
14         return err
15     }
16
17     if content != nil && len(content) != 0 {
18         pid, err := strconv.Atoi(string(content))
19         if err != nil {
20             return err
21         }
22         // 发送SIGTERM命令
23         if err := syscall.Kill(pid, syscall.SIGTERM); err != nil {
24             return err
25         }
26         if err := ioutil.WriteFile(serverPidFile, []byte{}, 0644); err != nil {
27             return err
28         }
29         fmt.Println("停止进程:", pid)
30     }
31     return nil
32 },
33 }
```

## 重启命令

最后我们要完成重启命令，还是在 framework/command/app.go 中。大致逻辑也很清晰，读取 PID 文件之后判断，如果 PID 文件中没有 PID，说明没有进程在运行，直接启动新进程；如果 PID 文件中有 PID，检查旧进程是否存在，如果不存在，直接启动新进程，如果存在，这里就有一些需要注意的了。

[复制代码](#)

```
1 //获取pid
2 ...
3
4 if content != nil && len(content) != 0 {
5     // 解析pid是否存在
6     if util.CheckProcessExist(pid) {
7         // 关闭旧的pid进程
8         ...
9     }
10 }
11
12 appDaemon = true
13 // 启动新的进程
14 return appStartCommand.RunE(c, args)
```

因为重启的逻辑是先结束旧进程，再启动新进程。结束进程和停止命令一样，使用 SIGTERM 信号就能保证进程的优雅关闭了。但是**由于新、旧进程都是使用同一个端口，所以必须保证旧进程结束，才能启动新的进程。**

而怎么保证旧进程确实结束了呢？

这里可以使用前面定义的 CheckProcessExist 方法，每秒做一次轮询，检测 PID 对应的进程是否已经关闭。那么轮询多少次呢？

我们知道在启动进程的时候，设置了一个优雅关闭的最大超时时间 closeWait，这个 closeWait 的时间设置为秒。那么**为了轮询检查旧进程是否关闭，我们只需要设置次数超过 closeWait 的轮询时间即可。**考虑到 net/http 在 closeWait 之后还有一些程序运行的逻辑，这里我们可以设置为  $2 * \text{closeWait}$ ，时间是非常充裕的。关键代码如下：

[复制代码](#)

```
1 // 确认进程已经关闭,每秒检测一次, 最多检测closeWait * 2秒
2 for i := 0; i < closeWait*2; i++ {
3     if util.CheckProcessExist(pid) == false {
4         break
5     }
6     time.Sleep(1 * time.Second)
7 }
```

再严谨一些，可以这么设置，如果在  $2 * \text{closeWait}$  时间内，旧进程还未关闭，那么就不能启动新进程了，需要直接返回错误。所以，在  $2 * \text{closeWait}$  轮询之后，我们还需要再做一次检查，检查进程是否关闭，如果没有关闭的话，直接返回 error：

[复制代码](#)

```
1 // 确认进程已经关闭,每秒检测一次, 最多检测closeWait * 2秒
2 for i := 0; i < closeWait*2; i++ {
3     if util.CheckProcessExist(pid) == false {
4         break
5     }
6     time.Sleep(1 * time.Second)
7 }
8
9 // 如果进程等待了2*closeWait之后还没结束,返回错误,不进行后续的操作
10 if util.CheckProcessExist(pid) == true {
```

```
11     fmt.Println("结束进程失败:"+strconv.Itoa(pid), "请查看原因")
12     return errors.New("结束进程失败")
13 }
```

在确认旧进程结束后，记得把 PID 文件清空，再启动一个新进程。启动进程的逻辑还是比较复杂的，就不重复写了，我们直接调用 `appStartCommand` 的 `RunE` 方法来实现，会更优雅一些。

同其他命令一样，这里再梳理一下判断旧进程存在之后详细的实现步骤，如果存在：

发送 `SIGTERM` 信号

循环 `2*closeWait` 次数，每秒执行一次查询进程是否已经结束


如果某次查询进程已经结束，或者等待 `2*closeWait` 循环结束之后，再次查询一次进程

如果还未结束，返回进程结束失败

如果已经结束，将 PID 文件清空，启动新进程

在 `framework/command/app.go` 中，整体代码如下：

```
1 // 重新启动一个app服务
2 var appRestartCommand = &cobra.Command{
3     Use:     "restart",
4     Short:   "重新启动一个app服务",
5     RunE: func(c *cobra.Command, args []string) error {
6         container := c.GetContainer()
7         appService := container.MustMake(contract.AppKey).(contract.App)
8
9         // GetPid
10        serverPidFile := filepath.Join(appService.RuntimeFolder(), "app.pid")
11
12        content, err := ioutil.ReadFile(serverPidFile)
13        if err != nil {
14            return err
15        }
16
17        if content != nil && len(content) != 0 {
18            pid, err := strconv.Atoi(string(content))
19            if err != nil {
20                return err
21            }
22            if util.CheckProcessExist(pid) {
```

 复制代码

```
23 // 杀死进程
24 if err := syscall.Kill(pid, syscall.SIGTERM); err != nil {
25     return err
26 }
27 if err := ioutil.WriteFile(serverPidFile, []byte{}, 0644); err !=
28     return err
29 }
30
31 // 获取closeWait
32 closeWait := 5
33 configService := container.MustMake(contract.ConfigKey).(contract.
34 if configService.IsExist("app.close_wait") {
35     closeWait = configService.GetInt("app.close_wait")
36 }
37
38 // 确认进程已经关闭,每秒检测一次, 最多检测closeWait * 2秒
39 for i := 0; i < closeWait*2; i++ {
40     if util.CheckProcessExist(pid) == false {
41         break
42     }
43     time.Sleep(1 * time.Second)
44 }
45
46 // 如果进程等待了2*closeWait之后还没结束,返回错误,不进行后续的操作
47 if util.CheckProcessExist(pid) == true {
48     fmt.Println("结束进程失败:" + strconv.Itoa(pid), "请查看原因")
49     return errors.New("结束进程失败")
50 }
51
52 fmt.Println("结束进程成功:" + strconv.Itoa(pid))
53 }
54 }
55
56 appDaemon = true
57 // 直接daemon方式启动apps
58 return appStartCommand.RunE(c, args)
59 },
60 }
```

## 测试

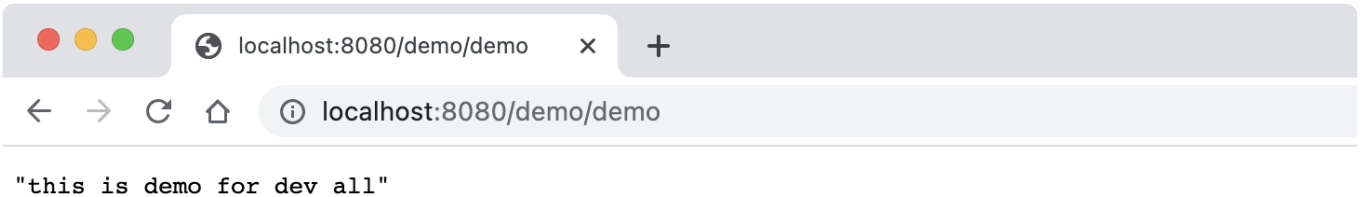
下面来测试一下。首先记得使用 `./hade build sef` 命令编译，我们设置的默认服务启动地址为 `":8888"`，这里就不用这个默认启动地址，使用环境变量 `ADDRESS=:8080` 来启动服务。这样能测试到环境变量是否能生效。

调用命令 `ADDRESS=:8080 ./hade app start --daemon=true` 以 daemon 方式启动一个 8080 端口的服务：



```
~/Documents/UGit/coredemo geekbang/22 ADDRESS=:8080 ./hade app start --daemon=true  
app启动成功, pid: 86793  
日志文件: /Users/yejianfeng/Documents/UGit/coredemo/storage/log/app.log
```

使用浏览器打开 localhost:8080/demo/demo :



A screenshot of a web browser window. The address bar shows 'localhost:8080/demo/demo'. The page content displays the text: "this is demo for dev all".

服务启动成功，且正常提供服务。

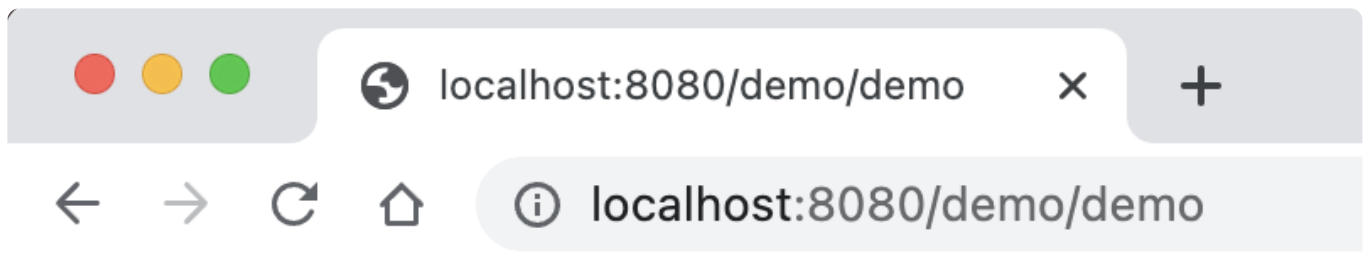
使用 `./hade app state` 查看进程状态：

```
~/Documents/UGit/coredemo geekbang/22 ./hade app state  
app服务已经启动, pid: 86793
```

使用命令 `ADDRESS=:8080 ./hade app restart` 重新启动进程：

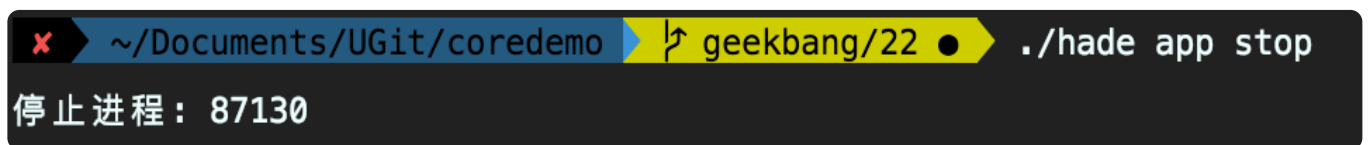
```
~/Documents/UGit/coredemo geekbang/22 ADDRESS=:8080 ./hade app restart  
结束进程成功:86793  
app启动成功, pid: 87130  
日志文件: /Users/yejianfeng/Documents/UGit/coredemo/storage/log/app.log
```

再次访问浏览器 localhost:8080/demo/demo，正常提供服务：



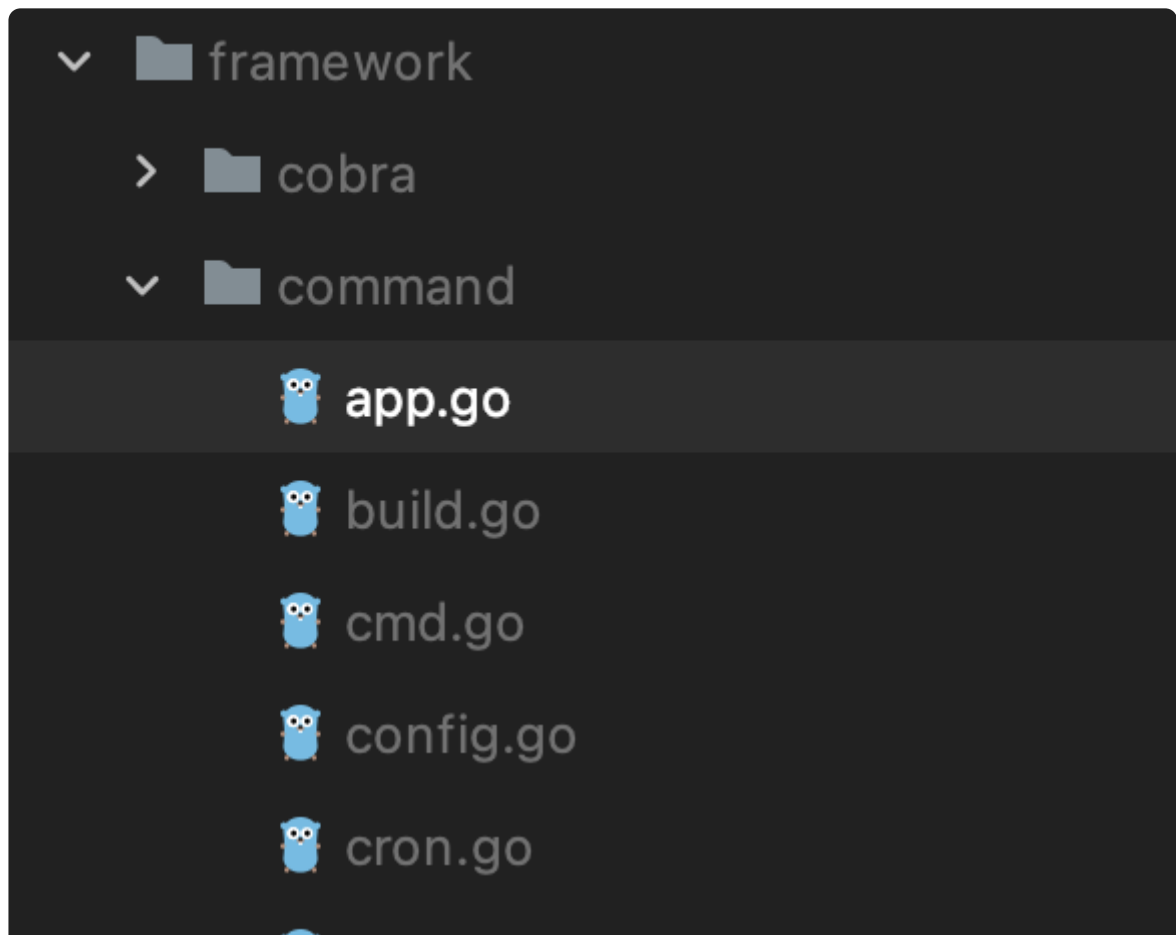
"this is demo for dev all"

最后调用停止进程命令 `./hade app stop`：



到这里，对进程的启动、关闭、查询和重启的命令就验证完成了。

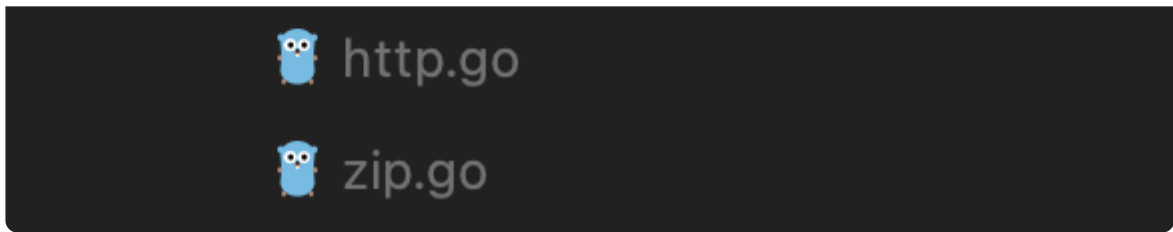
今天我们的所有代码都保存在 GitHub 上的 [@geekbang/24](#) 分支了。只修改了 `framework/command/app.go` 和 `framework/util/exec.go` 文件，其他保持不变。



```
🦉 dev.go
🦉 env.go
🦉 go_cmd.go
🦉 go_cmd_test.go
🦉 help.go
🦉 kernel.go
🦉 middleware.go
🦉 new.go
🦉 npm.go
🦉 provider.go

> 📁 contract
> 📁 gin
> 📁 middleware
> 📁 provider
✓ 📁 util

    🦉 console.go
    🦉 console_test.go
    🦉 exec.go
    🦉 file.go
```



## 小结

今天我们完成了运行 app 相关的命令，包括 app 一级命令和四个二级命令，启动 app 服务、停止 app 服务、重启 app 服务、查询 app 服务。基本上已经把 app 服务启动的状态变更都包含了。有了这些命令，我们对 app 的控制就方便很多了。特别是 daemon 运行模式，为线上运行提供了不少方便。

在实现这四个命令的过程中，我们使用了不少第三方库，gspt、go-daemon，这些库的使用你要能熟练掌握，特别是 go-daemon 库，我们已经不止一次使用到它了。确认一个进程是否已经结束，我们使用每秒做一次轮询的 CheckProcessExist 方法实现了检查机制，并仔细考虑了轮训的次数和效果，你可以多多体会这么设计的好处。

## 思考题

我们在启动应用的时候，使用的地址格式为 “:8080”，其实这里也可以为 “localhost:8080”、“127.0.0.1:8080” 或者 “10.11.22.33:8080”（10.11.22.33 为本机绑定的 IP）。你了解 localhost、127.0.0.1、10.11.22.33 以及不填写 IP 的区别么？

欢迎在留言区分享你的思考。感谢你的收听，如果你觉得今天的内容对你有所帮助，也欢迎分享给你身边的朋友，邀请他一起学习。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](#)[下一篇](#) 25 | GORM：数据库的使用必不可少（上）

## 训练营推荐

# Java 学习包免费领 NEW

面试题答案均由大厂工程师整理

阿里、美团等  
大厂真题18 大知识点  
专项练习大厂面试  
流程解析可复用的  
面试方法面试前  
要做的准备

## 精选留言 (2)

[写留言](#)

kngxue

2021-11-18

使用了syscall包，windows上就没法编译了吧

展开 ∨



宙斯

2021-11-14

为什么不把closeWait直接定义为2\*closeWait值呢

展开 ∨

共 2 条评论 &gt;

