



下载APP

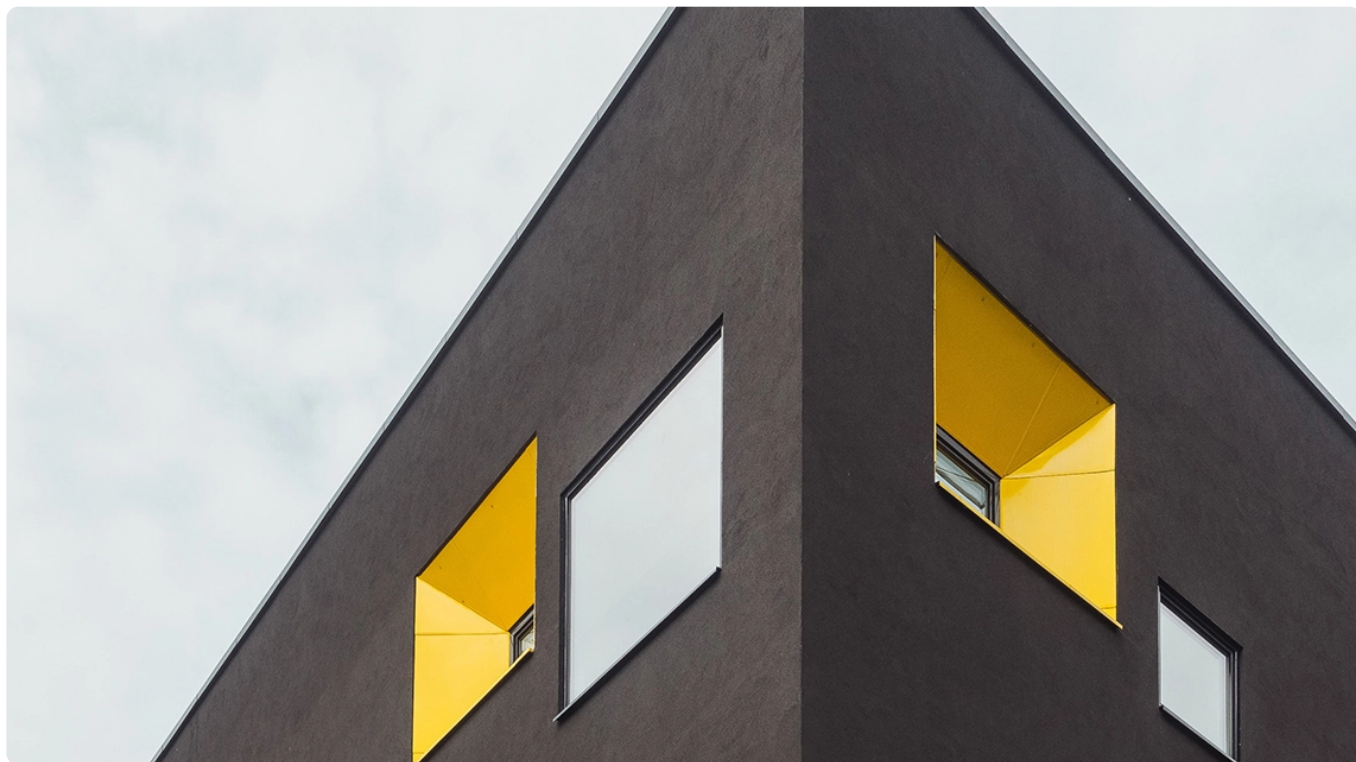


28 | SSH：如何生成发布系统让框架发布自动化？

2021-11-19 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 20:28 大小 18.74M



你好，我是轩脉刃。

在前面的课程中，我们基本上已经完成了一个能同时生成前端和后端的框架 hade，也能很方便对框架进行管理控制。下面两节课，我们来考虑框架的一些周边功能，比如部署自动化。

部署自动化其实不是一个框架的刚需，有很多方式可以将一个服务进行自动化部署，比如现在比较流行的 Docker 化或者 CI/CD 流程。



但是是一些比较个人比较小的项目，比如一个博客、一个官网网站，**这些部署流程往往都太庞大了，更需要一个服务，能快速将在开发机器上写好、调试好的程序上传到目标服务器，并且更新应用程序。**这就是我们今天要实现的框架发布自动化。

所有的部署自动化工具，基本都依赖本地与远端服务器的连接，这个连接可以是 FTP，可以是 HTTP，但是更经常的连接是 SSH 连接。因为一旦我们购买了一个 Web 服务器，服务器提供商就会提供一个有 SSH 登录账号的服务器，我们可以通过这个账号登录到服务器上，来进行各种软件的安装，比如 FTP、HTTP 服务等。

基本上，SSH 账号是我们拿到 Web 服务器的首要凭证，所以要设计的自动化发布系统也是依赖 SSH 的。


SSH 服务

那么在 Golang 中如何 SSH 连接远端的服务器呢？有一个 [ssh](#) 库能完成 SSH 的远端连接。

这里介绍一个小知识，你可以看下这个 ssh 库的 git：[golang.org/x/crypto/ssh](https://github.com/golang/crypto/ssh)。它是在官网 golang.org 下的，但是又不是官方的标准库，因为子目录是 x。

这种库其实也是经过官方认证的，属于实验性的库，我们可以这么理解：**以 golang.org/x/ 开头的库，都是官方认为这些库后续有可能成为标准库的一部份**，但是由于种种原因，现在还没有计划放进标准库中，需要更多时间打磨。但是这种库的维护者和开发者一般已经是 Golang 官方组的人员了。比如现在今年讨论热度很大的 Golang 泛型，据说也会先以实验库的形式出现。

不管怎么样，这种以 golang.org/x/ 开头的库，成熟度已经非常高了，我们是可以放心使用的。来了解一下这个 ssh 库：

 复制代码

```
1 package main
2
3 import (
4     "bytes"
5     "fmt"
6     "log"
7
8     "golang.org/x/crypto/ssh"
9 )
10
11 func main() {
12     var hostKey ssh.PublicKey
13 }
```

```
14 // ssh相关配置
15 config := &ssh.ClientConfig{
16     User: "username",
17     Auth: []ssh.AuthMethod{
18         ssh.Password("yourpassword"),
19     },
20     HostKeyCallback: ssh.FixedHostKey(hostKey),
21 }
22 // 创建client
23 client, err := ssh.Dial("tcp", "yourserver.com:22", config)
24 if err != nil {
25     log.Fatal("Failed to dial: ", err)
26 }
27 defer client.Close()
28
29 // 使用client做各种操作
30
31 session, err := client.NewSession()
32 if err != nil {
33     log.Fatal("Failed to create session: ", err)
34 }
35 defer session.Close()
36
37 var b bytes.Buffer
38 session.Stdout = &b
39 if err := session.Run("/usr/bin/whoami"); err != nil {
40     log.Fatal("Failed to run: " + err.Error())
41 }
42 fmt.Println(b.String())
43 }
```


在这个官方示例中，我们可以看到 ssh 库作为客户端连接，最重要的是创建 ssh.Client 这个数据结构，而这个数据结构使用 ssh.Dial 能进行创建，创建的时候依赖 ssh.ClientConfig 这么一个配置结构。

是不是非常熟悉？和前面的 Gorm、Redis 一样，将 SSH 的连接部分封装成为 hade 框架的 SSH 服务，这样我们就能很方便地初始化一个 ssh.Client 了。

经过前面几节课，相信你已经非常熟悉这种套路了，我们就简要说明下 ssh service 的封装和实现思路。这节课的重点在后面对自动化发布系统的实现上。

ssh service 的封装一样有三个部分，服务协议、服务提供者、服务实现。

服务协议我们提供 GetClient 方法：


 复制代码

```

1 // SSHService 表示一个ssh服务
2 type SSHService interface {
3     // GetClient 获取ssh连接实例
4     GetClient(option ...SSHOption) (*ssh.Client, error)
5 }

```

而其中的 SSHOption 作为更新 SSHConfig 的函数：


 复制代码

```

1 // SSHOption 代表初始化的时候的选项
2 type SSHOption func(container framework.Container, config *SSHConfig) error

```

我们封装配置结构为 SSHConfig：


 复制代码

```

1 // SSHConfig 为hade定义的SSH配置结构
2 type SSHConfig struct {
3     NetWork string
4     Host     string
5     Port     string
6     *ssh.ClientConfig
7 }

```

对应的配置文件如下 config/testing/ssh.yaml，你可以看看每个配置的说明：

 复制代码

```

1 timeout: 1s
2 network: tcp
3 web-01:
4     host: 118.190.3.55 # ip地址
5     port: 22 # 端口
6     username: yejianfeng # 用户名
7     password: "123456" # 密码
8 web-02:
9     network: tcp
10    host: localhost # ip地址
11    port: 3306 # 端口
12    username: jianfengye # 用户名
13    rsa_key: "/Users/user/.ssh/id_rsa"
14    known_hosts: "/Users/user/.ssh/known_hosts"

```

这里注意下，SSH 的连接方式有两种，一种是直接使用用户名密码来连接远程服务器，还有一种是使用 rsa key 文件来连接远端服务器，所以这里的配置需要同时支持两种配置。

对于使用 rsa key 文件的方式，需要设置 rsk_key 的私钥地址和负责安全验证的 known_hosts。

定义好了 SSH 的服务协议，服务提供者和服务实现并没有什么特别，就不展示具体代码了，在 GitHub 上的 [provider/ssh/provider.go](https://github.com/provider-ssh/provider.go) 和 [provider/ssh/service.go](https://github.com/provider-ssh/service.go) 中。我们简单说一下思路。

对于服务提供者，我们实现基本的五个函数 Register/Boot/IsDefer/Param/Name。另外这个 ssh 服务并不是框架启动时候必要加载的，所以设置 IsDefer 为 true，而 Param 我们就照例把服务容器 container 作为参数，传递给 Register 设定的实例化方法。

而 SSH 服务的具体实现，同样类似 Redis，先配置更新，再查询是否已经实例化，若已经实例化，返回实例化对象；若没有实例化，实例化 client，并且存在 map 中。

完成了 SSH 的服务协议、服务提供者、服务实例，我们就重点讨论下如何使用 SSH 的服务协议来实现自动化部署。

自动化部署

首先还是思考清楚自动化部署的命令设计。我们的 hade 框架是同时支持前后端的开发框架，所以自动化部署是需要同时支持前后端部署的，也就是说它的命令也需要支持前后端的部署，这里我们设计一个显示帮助信息的一级命令 ./hade deploy 和四个二级命令：

```
./hade deploy frontend , 部署前端  
./hade deploy backend , 部署后端  
./hade deploy all , 同时部署前后端  
./hade deploy rollback , 部署回滚
```

同时也设计一下部署配置文件。

首先，我们是需要知道部署在哪个或者哪几个服务器上的，所以需要有一个数组配置项 `connections` 来定义部署服务器。而部署服务器的具体用户名密码配置，在前面 SSH 的配置里是存在的，所以这里直接把 SSH 的配置路径放在我们的 `connections` 中就可以了。

其次，还要知道我们要部署的远端服务器的目标文件夹是什么？所以这里需要有一个 `remote_folder` 配置项来配置远端文件夹。

然后就是前端部署的配置 `frontend` 了。我们知道，在本地编译之后，会直接编译成了 `dist` 目录下的 HTML/JS/CSS 文件，这些文件直接上传到远端文件夹就是可以使用的了。

但是，在上传前端编译文件之前和在远端服务器执行一些命令之后，是有可能要做一些操作的。比如上传前先清空远端文件夹、上传后更新 `nginx` 等。所以这里，**我们设计两个数组结构 `pre_action` 和 `post_action` 来分别存放部署的前置命令和部署的后置命令。**

最后就是后端部署的配置 `backend`。同前端部署一样，我们也有部署的前置命令和后置命令。但是后端编译还有一个不同点。

因为后端是 `Golang` 编译的，而它的编译其实是分平台的，加上 `Go` 支持“交叉编译”。就是说，比如我的工作机器是 `Mac` 操作系统，`Web` 服务器是 `Linux` 操作系统，那么我需要编译 `Linux` 操作系统的后端程序，但是我可以直接在 `Mac` 操作系统上使用 `GOOS` 和 `GOARCH` 来编译 `Linux` 操作系统的程序：

```
1 GOOS=linux GOARCH=amd64 go build ./
```

[复制代码](#)

这样编译出来的文件就是可以在 `Linux` 运行的后端进程了。所以在后端部署的配置项里面，我们增加 `GOOS` 和 `GOARCH` 分别表示后端的交叉编译参数。

完整的配置文件在 `config/development/deploy.yaml` 中：

```
1 connections: # 要自动化部署的连接
2   - ssh.web-01
3
4 remote_folder: "/home/yejianfeng/coredemo/" # 远端的部署文件夹
```

[复制代码](#)


```
5 frontend: # 前端部署配置
6     pre_action: # 部署前置命令
7         - "pwd"
8     post_action: # 部署后置命令
9         - "pwd"
10
11 backend: # 后端部署配置
12     goos: linux # 部署目标操作系统
13     goarch: amd64 # 部署目标cpu架构
14     pre_action: # 部署前置命令
15         - "pwd"
16     post_action: # 部署后置命令
17         - "chmod 777 /home/yejianfeng/coredemo/hade"
18         - "/home/yejianfeng/coredemo/hade app restart"
19
```

好，配置文件设计好了，下面我们开始实现对应的命令。

其实估计你对如何实现，已经大致心中有数了。一级命令 `./hade deploy` 还是并没有什么内容，只是将帮助信息打印出来，之前也做过很多次，就不描述了。二级命令按之前的套路，一般是先编译，再部署，最后上传到目标服务器。

部署前端

看二级命令 `./hade deploy frontend`。对于部署前端，我们分为三个步骤：

创建要部署的文件夹；

编译前端文件到部署文件夹中；

上传部署文件夹，并且执行对应的前置和后置的 shell。

在 `framework/command/deploy.go` 中：

```
1 // deployFrontendCommand 部署前端
2 var deployFrontendCommand = &cobra.Command{
3     Use:     "frontend",
4     Short:   "部署前端",
5     RunE: func(c *cobra.Command, args []string) error {
6         container := c.GetContainer()
7
8         // 创建部署文件夹
9         deployFolder, err := createDeployFolder(container)
```

 复制代码

```
10     if err != nil {
11         return err
12     }
13
14     // 编译前端到部署文件夹
15     if err := deployBuildFrontend(c, deployFolder); err != nil {
16         return err
17     }
18
19     // 上传部署文件夹并执行对应的shell
20     return deployUploadAction(deployFolder, container, "frontend")
21 },
22 }
```

这里可能你会有个疑惑，为什么要创建一个部署文件夹？我们直接将前端编译的 dist 目录上传到目标服务器不就行了么？来为你解答下。


部署服务是一个很细心的过程，因为它会影响现在的线上服务，而每次部署都是有可能失败的，也就很有可能需要进行回滚操作，就是我们前面定义的部署回滚操作命令 `./hade deploy rollback`。而回滚的时候，需要能找到某个特定版本的编译内容，这里就需要部署文件夹。

这个部署文件夹我们定义为目录 `deploy/xxxxxx`，其中的 `xxxx` 直接设置为细化到秒的时间。对应的创建部署文件夹的函数如下：

[复制代码](#)

```
1 // 创建部署的folder
2 func createDeployFolder(c framework.Container) (string, error) {
3     appService := c.MustMake(contract.AppKey).(contract.App)
4     deployFolder := appService.DeployFolder()
5
6     // 部署文件夹的名称
7     deployVersion := time.Now().Format("20060102150405")
8     versionFolder := filepath.Join(deployFolder, deployVersion)
9     if !util.Exists(versionFolder) {
10         return versionFolder, os.Mkdir(versionFolder, os.ModePerm)
11     }
12     return versionFolder, nil
13 }
```


这里的 `appService.DeployFolder()` 是我们在 `appService` 下创建的一个新的目录 `deploy`，在 `framework/contract/app.go` 中：

 复制代码

```
1 // App 定义接口
2 type App interface {
3     ...
4     // DeployFolder 存放部署的时候创建的文件夹
5     DeployFolder() string
6     ...
7 }
```

有了这个部署文件夹，每次的发布都有“档案”存储了，这就为回滚命令提供了可能性。我们每次编译的文件，也都会先经过这个部署文件夹，再中转上传到目标服务器。

第一步创建部署文件夹实现了，我们再回头看下部署前端的第二个步骤，编译前端文件到部署文件夹。可以直接使用 `buildFrontendCommand` 的 `RunE` 方法，它会将前端编译到 `dist` 目录下，然后我们再将 `dist` 目录文件拷贝到部署文件夹中：

 复制代码

```
1 func deployBuildFrontend(c *cobra.Command, deployFolder string) error {
2     container := c.GetContainer()
3     appService := container.MustMake(contract.AppKey).(contract.App)
4
5     // 编译前端
6     if err := buildFrontendCommand.RunE(c, []string{}); err != nil {
7         return err
8     }
9
10    // 复制前端文件到deploy文件夹
11    frontendFolder := filepath.Join(deployFolder, "dist")
12    if err := os.Mkdir(frontendFolder, os.ModePerm); err != nil {
13        return err
14    }
15
16    buildFolder := filepath.Join(appService.BaseFolder(), "dist")
17    if err := util.CopyFolder(buildFolder, frontendFolder); err != nil {
18        return err
19    }
20    return nil
21 }
```

第三步，上传部署文件夹，并且执行对应的前置和后置的 shell。


这个步骤的实现是今天这节课的重点了。首先遍历配置文件中的 `deploy.connections`，明确我们要在哪几个远端节点中进行部署；然后对每个远端服务创建一个 `ssh.Client`，由于前面已经写好了 SSH 服务，所以直接使用 `GetClient` 方法就能为每个节点创建一个 `sshClient` 了：

 复制代码

```
1 for _, node := range deployNodes {
2     sshClient, err := sshService.GetClient(ssh.WithConfigPath(node))
3     if err != nil {
4         return err
5     }
6     ...
7 }
```

接下来就要执行命令了，那怎么执行前置或者后置命令呢？

我们需要为每个命令创建一个 `session`，然后使用 `session.CombinedOut` 来输出这个命令的结果，把每个命令的结果都输出在控制台中。相关代码如下：

 复制代码

```
1 for _, action := range preActions {
2     // 创建session
3     session, err := sshClient.NewSession()
4     if err != nil {
5         return err
6     }
7     // 执行命令，并且等待返回
8     bts, err := session.CombinedOutput(action)
9     if err != nil {
10         session.Close()
11         return err
12     }
13     session.Close()
14
15     // 执行前置命令成功
16     logger.Info(context.Background(), "execute pre action", map[string]interface{}{
17         "cmd":      action,
18         "connection": node,
19         "out":      strings.ReplaceAll(string(bts), "\n", ""),
20     })
21 }
```

执行了前置命令之后，下面就是要把部署文件夹中的文件上传到目标服务器了。如何通过 SSH 服务将文件上传到目标服务器呢？

这里需要使用到一个成熟的第三方库 [sftp](#) 了，目前已经有 1.1k star，采用 BSD-2 的开源协议，允许修改商用，但是要保留申明。这个库就是封装 SSH 的，将 SFTP 文件传输协议封装了一下。SFTP 是什么？它是基于 SSH 协议来进行文件传输的一个协议，功能与 FTP 相似，区别就是它的连接通道使用 SSH。

SFTP 的底层连接实际上就是 SSH，只是把传输的文件内容进行了一下加密等工作，增加了传输的安全性。所以 **SFTP 本质就是“使用 SSH 连接来完成文件传输功能”**。这点可以从它的实例化看出，`sftp.Client` 的唯一参数就是 `ssh.Client`。

[复制代码](#)

```
1 client, err := sftp.NewClient(sshClient)
2 if err != nil {
3     return err
4 }
```

SFTP 这个库，在初始化 `sftp.Client` 之后，会将这个 client 封装地和官方的本地操作文件 OS 库一样，你在使用 `sftp.Client` 的时候完全没有障碍。

比如，OS 库创建一个文件是 `os.Create`，在 SFTP 中就是使用 `client.Create`；OS 库获取一个文件信息的函数是 `os.Stat`，在 SFTP 中就是 `client.Stat`。但是注意下，这里完全是 SFTP 刻意将这个库函数设计的和 OS 库一样的，它们之间并没有什么嵌套关系。

我们使用 `ssh.Client` 初始化一个 `sftp.Client` 之后，写一个 `uploadFolderToSFTP` 的函数来实现将本地文件夹同步到远端文件夹：

[复制代码](#)

```
1 // 上传部署文件夹
2 func uploadFolderToSFTP(container framework.Container, localFolder, remoteFolder string) error {
3     logger := container.MustMake(contract.LogKey).(contract.Log)
4     // 遍历本地文件
5     return filepath.Walk(localFolder, func(path string, info os.FileInfo, err error) error {
6         // 获取除了folder前缀的后续文件名称
7         relPath := strings.Replace(path, localFolder, "", 1)
8         if relPath == "" {
9             return nil
10        }
11        // 上传文件
12        return client.Create(remoteFolder + relPath, info.Mode())
13    })
14 }
```

```
10     }
11     // 如果是遍历到了一个目录
12     if info.IsDir() {
13         logger.Info(context.Background(), "mkdir: "+filepath.Join(remoteFo
14         // 创建这个目录
15         return client.MkdirAll(filepath.Join(remoteFolder, relPath))
16     }
17
18     // 打开本地的文件
19     rf, err := os.Open(filepath.Join(localFolder, relPath))
20     if err != nil {
21         return errors.New("read file " + filepath.Join(localFolder, relPat
22     }
23     // 检查文件大小
24     rfStat, err := rf.Stat()
25     if err != nil {
26         return err
27     }
28     // 打开/创建远端文件
29     f, err := client.Create(filepath.Join(remoteFolder, relPath))
30     if err != nil {
31         return errors.New("create file " + filepath.Join(remoteFolder, rel
32     }
33
34     // 大于2M的文件显示进度
35     if rfStat.Size() > 2*1024*1024 {
36         logger.Info(context.Background(), "upload local file: "+filepath.J
37         " to remote file: "+filepath.Join(remoteFolder, relPath)+" sta
38         // 开启一个goroutine来不断计算进度
39         go func(localFile, remoteFile string) {
40             // 每10s计算一次
41             ticker := time.NewTicker(2 * time.Second)
42             for range ticker.C {
43                 // 获取远端文件信息
44                 remoteFileInfo, err := client.Stat(remoteFile)
45                 if err != nil {
46                     logger.Error(context.Background(), "stat error", map[s
47                         "err":          err,
48                         "remote_file": remoteFile,
49                 })
50                 continue
51             }
52             // 如果远端文件大小等于本地文件大小, 说明已经结束了
53             size := remoteFileInfo.Size()
54             if size >= rfStat.Size() {
55                 break
56             }
57             // 计算进度并且打印进度
58             percent := int(size * 100 / rfStat.Size())
59             logger.Info(context.Background(), "upload local file: "+fi
60             " to remote file: "+filepath.Join(remoteFolder, relPat
61         }
```

```
62         }(filepath.Join(localFolder, relPath), filepath.Join(remoteFolder,
63         })
64
65         // 将本地文件并发读取到远端文件
66         if _, err := f.ReadFromWithConcurrency(rf, 10); err != nil {
67             return errors.New("Write file " + filepath.Join(remoteFolder, relP
68         })
69         // 记录成功信息
70         logger.Info(context.Background(), "upload local file: "+filepath.Join(
71             " to remote file: "+filepath.Join(remoteFolder, relPath)+" finish"
72         return nil
73     })
74 }
```

这段代码长一点。首先我们使用功能 `filePath.Walk` 来遍历本地文件夹中的所有文件，如果遍历到的是子文件夹，就创建子文件夹，否则的话，我们就将本地文件上传到远端。而上传远端的操作大致就是三步：打开本地文件、打开远端文件、将本地文件传输到远端文件。

在上述函数中大致是这几句代码：

[复制代码](#)

```
1 // 打开本地的文件
2 rf, err := os.Open(filepath.Join(localFolder, relPath))
3
4 // 打开/创建远端文件
5 f, err := client.Create(filepath.Join(remoteFolder, relPath))
6
7 // 将本地文件并发读取到远端文件
8 if _, err := f.ReadFromWithConcurrency(rf, 10); err != nil
```

SFTP 提供了并发读取到远端文件 `ReadFromWithConcurrency` 的方法，我们可以使用这个并发读的方法提高上传效率。

但是即使是并发读，对于比较大的文件，还是需要等候比较长的时间。而这个等待时长，对于在控制台敲下部署命令的使用者来说是非常不友好的。我们希望能**每隔一段时间显示一下当前的部署进度**，这个怎么做呢？

这里我们设计大于 2M 的文件，执行这个操作。2M 是我自己实验出来体验比较差的一个阈值。然后每 2s 就打印一下当前进度，所以使用了一个 ticker，来计算时间。每次这个

ticker 结束的时候, 计算一下远端文件的大小, 再计算一下本地文件的大小。两者相除就是这个文件的上传进度, 再使用日志打印就能打印出具体的进度了。

最后的效果如下:

```
[Info] 2021-11-10T09:57:27+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110095722/hade to remote file: /home/yejianfeng/coredemo/hade start" map[]
[Info] 2021-11-10T09:57:29+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110095722/hade to remote file: /home/yejianfeng/coredemo/hade 19% 10321920/53532678" map[]
[Info] 2021-11-10T09:57:31+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110095722/hade to remote file: /home/yejianfeng/coredemo/hade 50% 27131984/53532678" map[]
[Info] 2021-11-10T09:57:33+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110095722/hade to remote file: /home/yejianfeng/coredemo/hade 79% 42565632/53532678" map[]
[Info] 2021-11-10T09:57:34+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110095722/hade to remote file: /home/yejianfeng/coredemo/hade finish" map[]
```


到这里部署前端的代码就开发完成了。

部署后端

理解了如何部署前端, 部署后端的对应方法基本如出一辙。唯一不同的地方就是编译。

编译 Golang 的后端需要指定对应的编译平台和编译 CPU 架构, 就是前面说的 GOOS 和 GOARCH。所以我们就不能直接使用 build 命令来编译后端了。改成定位 go 程序, 来执行 go build, 并且需要修改输出文件路径, 输出到部署文件夹中。

当然这个部署文件夹还是按照我们之前的设计为 deploy/xxxxxx, 其中的 xxxx 直接设置为细化到秒的时间, 继续在 framework/command/deploy.go 中写入:

 复制代码

```
1 // 编译后端
2 path, err := exec.LookPath("go")
3 if err != nil {
4     log.Fatalln("hade go: 请在Path路径中先安装go")
5 }
6 // 组装命令
7 deployBinFile := filepath.Join(deployFolder, binFile)
8 cmd := exec.Command(path, "build", "-o", deployBinFile, ".")
9 cmd.Env = os.Environ()
10 // 设置GOOS和GOARCH
11 if configService.GetString("deploy.backend.goos") != "" {
12     cmd.Env = append(cmd.Env, "GOOS="+configService.GetString("deploy.backend.g
13 }
14 if configService.GetString("deploy.backend.goarch") != "" {
15     cmd.Env = append(cmd.Env, "GOARCH="+configService.GetString("deploy.backend
16 }
17 // 执行命令
18 ctx := context.Background()
19 out, err := cmd.CombinedOutput()
20 if err != nil {
```



```
21     logger.Error(ctx, "go build err", map[string]interface{}{
22         "err": err,
23         "out": string(out),
24     })
25     return err
26 }
27 logger.Info(ctx, "编译成功", nil)
```

同时除了生成二进制文件，还要记得把.env 文件（如果有的话）、config 目标文件传递到本地的部署目录：

[复制代码](#)

```
1 // 复制.env
2 if util.Exists(filepath.Join(appService.BaseFolder(), ".env")) {
3     if err := util.CopyFile(filepath.Join(appService.BaseFolder(), ".env"), fil
4         return err
5 }
6 }
7
8 // 复制config文件
9 deployConfigFolder := filepath.Join(deployFolder, "config", env)
10 if !util.Exists(deployConfigFolder) {
11     if err := os.MkdirAll(deployConfigFolder, os.ModePerm); err != nil {
12         return err
13     }
14 }
15 if err := util.CopyFolder(filepath.Join(appService.ConfigFolder(), env), deplo
16     return err
17 }
```

自动化部署后端的命令，除了以上的编译文件到部署目录之外，其他部分都和自动化部署前端的命令一致：

[复制代码](#)

```
1 // deployBackendCommand 部署后端
2 var deployBackendCommand = &cobra.Command{
3     Use: "backend",
4     Short: "部署后端",
5     RunE: func(c *cobra.Command, args []string) error {
6         container := c.GetContainer()
7
8         // 创建部署文件夹
9         deployFolder, err := createDeployFolder(container)
10        if err != nil {
```

```
11         return err
12     }
13
14     // 编译后端到部署文件夹
15     if err := deployBuildBackend(c, deployFolder); err != nil {
16         return err
17     }
18
19     // 上传部署文件夹并执行对应的shell
20     return deployUploadAction(deployFolder, container, "backend")
21 },
22 }
```

部署全部

而对于同时部署前后端命令，其实就是在编译阶段，把前端和后端同时进行编译，并且最终上传部署文件夹。同样放在 framework/command/deploy.go：

[复制代码](#)

```
1 var deployAllCommand = &cobra.Command{
2     Use:     "all",
3     Short:   "全部部署",
4     RunE: func(c *cobra.Command, args []string) error {
5         container := c.GetContainer()
6
7         deployFolder, err := createDeployFolder(container)
8         if err != nil {
9             return err
10        }
11
12        // 编译前端
13        if err := deployBuildFrontend(c, deployFolder); err != nil {
14            return err
15        }
16
17        // 编译后端
18        if err := deployBuildBackend(c, deployFolder); err != nil {
19            return err
20        }
21
22        // 上传前端+后端，并执行对应的shell
23        return deployUploadAction(deployFolder, container, "all")
24    },
25 }
```

部署回滚

最后就是部署回滚操作，主要明确一下需要传递的参数：

一个是回滚版本号。这个版本号就是我们的部署目录的名称，前面说过部署目录为 `deploy/xxxxxx`，`xxxx` 设置为细化到秒的时间。比如 `20211110233354`，表示是我们 2021 年 11 月 10 日 23 点 33 分 54 秒创建的版本。

另外一个就是标记希望回滚前端，还是后端，还是全部回滚。这里主要涉及执行前端的回滚命令，还是执行后端的回滚命令。

这两个参数我们直接以参数形式，跟在 `deploy rollback` 命令之后，如下：

[复制代码](#)

```
1 ./hade deploy rollback 20211110233354 backend
```

明确了参数，它的具体实现就很简单了，因为它没有任何的编译过程，我们只需要把回滚版本所在目录的编译结果，上传到目标服务器就可以了，同样，我们把这个命令放在 `framework/command/deploy.go` 中：

[复制代码](#)

```
1 // deployRollbackCommand 部署回滚
2 var deployRollbackCommand = &cobra.Command{
3     Use: "rollback",
4     Short: "部署回滚",
5     RunE: func(c *cobra.Command, args []string) error {
6         container := c.GetContainer()
7
8         if len(args) != 2 {
9             return errors.New("参数错误,请按照参数进行回滚 ./hade deploy rollback [ve
10        ]
11
12        version := args[0]
13        end := args[1]
14
15        // 获取版本信息
16        appService := container.MustMake(contract.AppKey).(contract.App)
17        deployFolder := filepath.Join(appService.DeployFolder(), version)
18
19        // 上传部署文件夹并执行对应的shell
20        return deployUploadAction(deployFolder, container, end)
21    },
22 }
```

到这里四个自动化部署命令就都开发完成。我们来验证一下。

验证

要验证部署命令，我们当然需要有一个目标部署服务器，这是我设置的 web-01 服务器配置，在 config/development/ssh.yaml 中：

[复制代码](#)

```
1 timeout: 3s
2 network: tcp
3 web-01:
4   host: 111.222.333.444 # ip地址
5   port: 22 # 端口
6   username: yejianfeng # 用户名
7   password: "123456" # 密码
```

而在 config/development/deploy.yaml 中我的配置如下：

[复制代码](#)

```
1 connections: # 要自动化部署的连接
2   - ssh.web-01
3
4 remote_folder: "/home/yejianfeng/coredemo/" # 远端的部署文件夹
5
6 frontend: # 前端部署配置
7   pre_action: # 部署前置命令
8     - "pwd"
9   post_action: # 部署后置命令
10    - "pwd"
11
12 backend: # 后端部署配置
13   goos: linux # 部署目标操作系统
14   goarch: amd64 # 部署目标cpu架构
15   pre_action: # 部署前置命令
16     - "rm /home/yejianfeng/coredemo/hade"
17   post_action: # 部署后置命令
18     - "chmod 777 /home/yejianfeng/coredemo/hade"
19     - "/home/yejianfeng/coredemo/hade app restart"
```

重点看后端部署配置。在部署后端之前，我们先运行一个 `rm` 命令来将旧的 `hade` 二进制进程删除，然后部署后端文件，其中包括这个二进制进程。最后执行了两个命令，一个是 `chmod` 命令，保证上传上去的二进制进程命令可以执行；第二个就是 `./hade app restart` 命令，能将远端的命令启动。

这里就演示下部署后端服务 `./hade deploy backend`，输出结果如下：

```
~/Documents/UGit/coredemo P geekbang/28 ./.hade deploy backend
[Info] 2021-11-10T23:35:37+08:00 "编译成功" map[]
[Info] 2021-11-10T23:35:37+08:00 "build local ok" map[]
[Info] 2021-11-10T23:35:37+08:00 "execute pre action start" map[cmd:rm /home/yejianfeng/coredemo/hade connection:ssh.web-01]
[Info] 2021-11-10T23:35:37+08:00 "execute pre action" map[cmd:rm /home/yejianfeng/coredemo/hade connection:ssh.web-01 out:]
[Info] 2021-11-10T23:35:37+08:00 "mkdir: /home/yejianfeng/coredemo/config" map[]
[Info] 2021-11-10T23:35:37+08:00 "mkdir: /home/yejianfeng/coredemo/config/development" map[]
[Info] 2021-11-10T23:35:37+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/config/development/app.yaml to remote file: /home/yejianfeng/coredemo/config/development/app.yaml finish" map[]
[Info] 2021-11-10T23:35:37+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/config/development/database.yaml to remote file: /home/yejianfeng/coredemo/config/development/database.yaml finish" map[]
[Info] 2021-11-10T23:35:37+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/config/development/deploy.yaml to remote file: /home/yejianfeng/coredemo/config/development/deploy.yaml finish" map[]
[Info] 2021-11-10T23:35:37+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/config/development/log.yaml to remote file: /home/yejianfeng/coredemo/config/development/log.yaml finish" map[]
[Info] 2021-11-10T23:35:37+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/config/development/ssh.yaml to remote file: /home/yejianfeng/coredemo/config/development/ssh.yaml finish" map[]
[Info] 2021-11-10T23:35:40+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/hade to remote file: /home/yejianfeng/coredemo/hade start" map[]
[Info] 2021-11-10T23:35:40+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/hade to remote file: /home/yejianfeng/coredemo/hade 2% 1343488/53539058" map[]
[Info] 2021-11-10T23:35:44+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/hade to remote file: /home/yejianfeng/coredemo/hade 4% 2586672/53539058" map[]
[Info] 2021-11-10T23:35:46+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/hade to remote file: /home/yejianfeng/coredemo/hade 6% 3538944/53539058" map[]
[Info] 2021-11-10T23:35:46+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/hade to remote file: /home/yejianfeng/coredemo/hade 8% 4628288/53539058" map[]
[Info] 2021-11-10T23:36:50+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/hade to remote file: /home/yejianfeng/coredemo/hade 91% 49028928/53539058" map[]
[Info] 2021-11-10T23:37:00+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/hade to remote file: /home/yejianfeng/coredemo/hade 94% 50593792/53539058" map[]
[Info] 2021-11-10T23:37:02+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/hade to remote file: /home/yejianfeng/coredemo/hade 96% 51904512/53539058" map[]
[Info] 2021-11-10T23:37:04+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/hade to remote file: /home/yejianfeng/coredemo/hade 98% 52985856/53539058" map[]
[Info] 2021-11-10T23:37:05+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233533/hade to remote file: /home/yejianfeng/coredemo/hade finish" map[]
[Info] 2021-11-10T23:37:05+08:00 "upload folder success" map[]
[Info] 2021-11-10T23:37:05+08:00 "execute post action start" map[cmd:chmod 777 /home/yejianfeng/coredemo/hade connection:ssh.web-01]
[Info] 2021-11-10T23:37:05+08:00 "execute post action finish" map[cmd:chmod 777 /home/yejianfeng/coredemo/hade connection:ssh.web-01 out:]
[Info] 2021-11-10T23:37:05+08:00 "execute post action start" map[cmd:/home/yejianfeng/coredemo/hade app restart connection:ssh.web-01]
[Info] 2021-11-10T23:37:06+08:00 "execute post action finish" map[cmd:/home/yejianfeng/coredemo/hade app restart connection:ssh.web-01 out:结束进程成功:4382app启动成功, pid: 4528日志文件: /home/yejianfeng/storage/log/app.log]
```

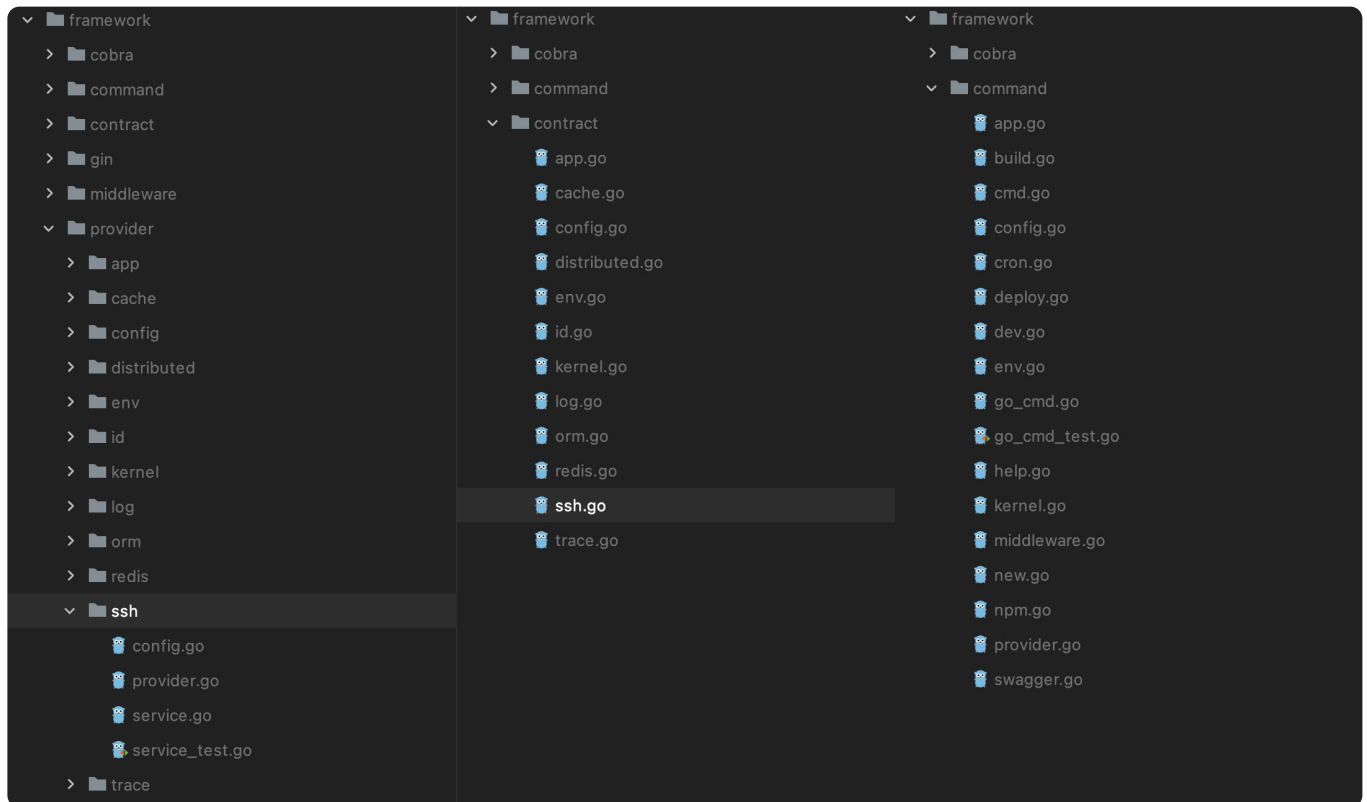
我们看到，它成功地编译后端服务，到目标文件夹 `deploy/20211110233533`，并且上传了编译的 `hade` 命令，在远端启动了进程。

接着验证下回滚命令。在之前已经发布过版本 `20211110233354` 了。所以这里直接运行命令 `./hade deploy rollback 20211110233354 backend` 将版本回滚到 `20211110233354`。

```
~/Documents/UGit/coredemo P geekbang/28 ./.hade deploy rollback 20211110233354 backend
[Info] 2021-11-10T23:39:29+08:00 "execute pre action start" map[cmd:rm /home/yejianfeng/coredemo/hade connection:ssh.web-01]
[Info] 2021-11-10T23:39:29+08:00 "execute pre action" map[cmd:rm /home/yejianfeng/coredemo/hade connection:ssh.web-01 out:]
[Info] 2021-11-10T23:39:29+08:00 "mkdir: /home/yejianfeng/coredemo/config" map[]
[Info] 2021-11-10T23:39:29+08:00 "mkdir: /home/yejianfeng/coredemo/config/development" map[]
[Info] 2021-11-10T23:39:29+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/config/development/app.yaml to remote file: /home/yejianfeng/coredemo/config/development/app.yaml finish" map[]
[Info] 2021-11-10T23:39:29+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/config/development/database.yaml to remote file: /home/yejianfeng/coredemo/config/development/database.yaml finish" map[]
[Info] 2021-11-10T23:39:30+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/config/development/deploy.yaml to remote file: /home/yejianfeng/coredemo/config/development/deploy.yaml finish" map[]
[Info] 2021-11-10T23:39:30+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/config/development/log.yaml to remote file: /home/yejianfeng/coredemo/config/development/log.yaml finish" map[]
[Info] 2021-11-10T23:39:30+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/config/development/ssh.yaml to remote file: /home/yejianfeng/coredemo/config/development/ssh.yaml finish" map[]
[Info] 2021-11-10T23:39:30+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/hade to remote file: /home/yejianfeng/coredemo/hade start" map[]
[Info] 2021-11-10T23:39:32+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/hade to remote file: /home/yejianfeng/coredemo/hade 2% 1507328/53450392" map[]
[Info] 2021-11-10T23:39:34+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/hade to remote file: /home/yejianfeng/coredemo/hade 5% 2719744/53450392" map[]
[Info] 2021-11-10T23:39:36+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/hade to remote file: /home/yejianfeng/coredemo/hade 7% 3899392/53450392" map[]
[Info] 2021-11-10T23:39:38+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/hade to remote file: /home/yejianfeng/coredemo/hade 10% 5050824/53450392" map[]
[Info] 2021-11-10T23:41:05+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/hade to remote file: /home/yejianfeng/coredemo/hade 92% 49184768/53450392" map[]
[Info] 2021-11-10T23:41:06+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/hade to remote file: /home/yejianfeng/coredemo/hade 93% 49848128/53450392" map[]
[Info] 2021-11-10T23:41:09+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/hade to remote file: /home/yejianfeng/coredemo/hade 95% 50987008/53450392" map[]
[Info] 2021-11-10T23:41:10+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/hade to remote file: /home/yejianfeng/coredemo/hade 97% 51937280/53450392" map[]
[Info] 2021-11-10T23:41:12+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/hade to remote file: /home/yejianfeng/coredemo/hade 99% 52985856/53450392" map[]
[Info] 2021-11-10T23:41:13+08:00 "upload local file: /Users/yejianfeng/Documents/UGit/coredemo/deploy/20211110233354/hade to remote file: /home/yejianfeng/coredemo/hade finish" map[]
[Info] 2021-11-10T23:41:13+08:00 "upload folder success" map[]
[Info] 2021-11-10T23:41:13+08:00 "execute post action start" map[cmd:chmod 777 /home/yejianfeng/coredemo/hade connection:ssh.web-01]
[Info] 2021-11-10T23:41:13+08:00 "execute post action finish" map[cmd:chmod 777 /home/yejianfeng/coredemo/hade connection:ssh.web-01 out:]
[Info] 2021-11-10T23:41:13+08:00 "execute post action start" map[cmd:/home/yejianfeng/coredemo/hade app restart connection:ssh.web-01]
[Info] 2021-11-10T23:41:14+08:00 "execute post action finish" map[cmd:/home/yejianfeng/coredemo/hade app restart connection:ssh.web-01 out:结束进程成功:4528app启动成功, pid: 4713日志文件: /home/yejianfeng/storage/log/app.log]
```

验证成功！

本节课我们对 framework 下的 provider、contract、command 目录都有修改。目录截图如下，供你对比查看，所有代码都已经上传到 [geekbang/28](https://github.com/geekbang/28) 分支了。



小结

今天我们实现了将代码自动化部署到 Web 服务器的机制。为了实现这个自动化部署，先实现了一个 SSH 服务，然后定制了一套自动化部署命令，包括部署前端、部署后端、部署全部和部署回滚。

虽然说这个由框架负责的自动化部署机制在大项目中可能用不上，毕竟现在大项目都采用 Docker 化和 k8s 部署了。不过对于小型项目，这种部署机制还是有其便利性的。所以我们的 hade 框架还是决定提供这个机制。

在实现这个机制的过程中，要做到熟练掌握 Golang 对于 SSH、SFTP 等库的操作。基本上这两个库的操作你熟悉了，就能在一个程序中同时自动化操作多个服务器了。在实际工作中，如果遇到类似的需求，可以按照这节课所展示的技术来自动化你的需求。

思考题

其实今天的内容涉及自动化运维的范畴了，我们就布置一个课外研究吧。自动化运维范畴中有一个很出名的自动化运维配置框架 ansible，你可以去浏览下 [Ansible 中文权威指南](#) 网站，学习一下 ansible 有哪些功能，分享一下你的学习心得。

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 1  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 27 | 缓存服务：如何基于Redis实现封装？

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

 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。