



下载APP



## 加餐 | 如何拉取私有的Go Module ?

2022-01-21 Tony Bai

《Tony Bai · Go语言第一课》

课程介绍 >



讲述 : Tony Bai

时长 18:52 大小 17.29M



你好，我是 Tony Bai。

我们这门课程上线以来收到了童鞋们的众多留言与热烈反馈，在这些留言和反馈中，有关 Go Module 的问题占比比较大，其中又以下面这两个问题比较突出：

在某 module 尚未发布到类似 GitHub 这样的网站前，如何 import 这个本地的 module ?

如何拉取私有 module ?

借这次加餐机会，今天我就针对这两个问题和你聊聊我知道的一些解决方案。



首先我们先来看第一个问题：如何导入本地的 module。

## 导入本地 module


在前面的 06 和 07 讲，我们已经系统讲解了 Go Module 构建模式。Go Module 从 Go 1.11 版本开始引入到 Go 中，现在它已经成为了 Go 语言的依赖管理与构建的标准，因此，我也一直建议你彻底抛弃 Gopath 构建模式，全面拥抱 Go Module 构建模式。并且，这门课中的所有例子和实战小项目，我使用的都是 Go Module 构建模式。

当我们的项目依赖已发布在 GitHub 等代码托管站点的公共 Go Module 时，Go 命令工具可以很好地完成依赖版本选择以及 Go Module 拉取的工作。

**不过，如果我们的项目依赖的是本地正在开发、尚未发布到公共站点上的 Go Module，那么我们应该如何做呢？**我们来看一个例子。

假设你有一个项目，这个项目中的 module a 依赖 module b，而 module b 是你另外一个项目中的 module，它本来是要发布到 `github.com/user/b` 上的。


但此时此刻，module b 还没有发布到公共托管站点上，它源码还在你的开发机器上。也就是说，go 命令无法在 `github.com/user/b` 上找到并拉取 module a 的依赖 module b，这时，如果你针对 module a 所在项目使用 `go mod tidy` 命令，就会收到类似下面这样的报错信息：

 复制代码

```
1 $go mod tidy
2 go: finding module for package github.com/user/b
3 github.com/user/a imports
4     github.com/user/b: cannot find module providing package github.com/user/b:
5     server response:
6     not found: github.com/user/b@latest: terminal prompts disabled
7     Confirm the import path was entered correctly.
8     If this is a private repository, see https://golang.org/doc/faq#git_https
```

这个时候，我们就可以借助 **go.mod 的 replace 指示符**，来解决这个问题。解决的步骤是这样的：


首先，我们需要在 module a 的 go.mod 中的 require 块中，手工加上这一条（这也可以通过 `go mod edit` 命令实现）：

 复制代码

```
1 require github.com/user/b v1.0.0
```

注意了，这里的 v1.0.0 版本号是一个“假版本号”，目的是满足 go.mod 中 require 块的语法要求。

然后，我们再在 module a 的 go.mod 中使用 replace，将上面对 module b v1.0.0 的依赖，替换为本地路径上的 module b:

 复制代码

```
1 replace github.com/user/b v1.0.0 => module b的本地源码路径
```

这样修改之后，go 命令就会让 module a 依赖你本地正在开发、尚未发布到代码托管网站的 module b 的源码了。

而且，如果 module b 已经提交到类 GitHub 的站点上，但 module b 的作者正在本地开发新版本，那么上面这种方法，也同样适合 module b 的作者在本地的测试验证 module b 的最新版源码。

虽然“伪造” go.mod 文件内容，可以解决上述这两个场景中的问题，但显然这种方法也是有“瑕疵”的。

首先，这个方法中，require 指示符将 github.com/user/b v1.0.0 替换为一个本地路径下的 module b 的源码版本，但这个**本地路径**是因开发者环境而异的。

前面课程中我们讲过，go.mod 文件通常是要上传到代码服务器上的，这就意味着，另外一个开发人员下载了这份代码后，极大可能是无法成功编译的，他要想完成 module a 的编译，就得将 replace 后面的本地路径改为适配自己环境下的路径。

于是，每当开发人员 pull 代码后，第一件事就是要修改 module a 的 go.mod 中的 replace 块，每次上传代码前，可能也要将 replace 路径复原，这是一个很糟心的事情。但即便如此，目前 Go 版本（最新为 Go 1.17.x）也没有一个完美的应对方案。

针对这个问题，Go 核心团队在 Go 社区的帮助下，在预计 2022 年 2 月发布的 Go 1.18 版本中加入了 Go 工作区（Go workspace，也译作 Go 工作空间）辅助构建机制。

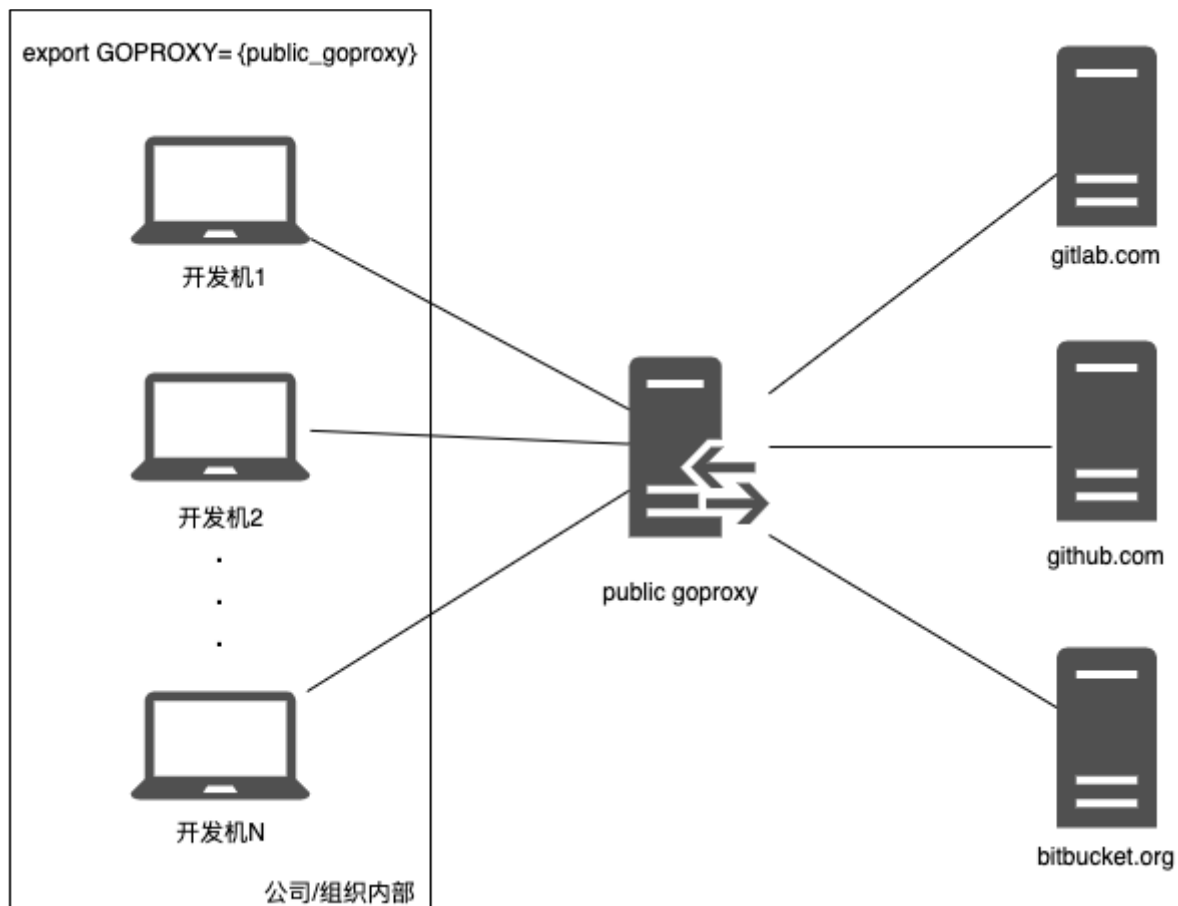
基于这个机制，我们可以将多个本地路径放入同一个 workspace 中，这样，在这个 workspace 下各个 module 的构建将优先使用 workspace 下的 module 的源码。工作区配置数据会放在一个名为 go.work 的文件中，这个文件是开发者环境相关的，因此并不需要提交到源码服务器上，这就解决了上面“伪造 go.mod”方案带来的那些问题。

不过，Go 1.18 版本尚未发布，我这里就不再深入讲解了 Go workspace 机制了，如果你有兴趣，可以去下载 Go 1.18 Beta1 版本抢先体验。

接下来，我们再来看看拉取私有 module 的可行解决方案。

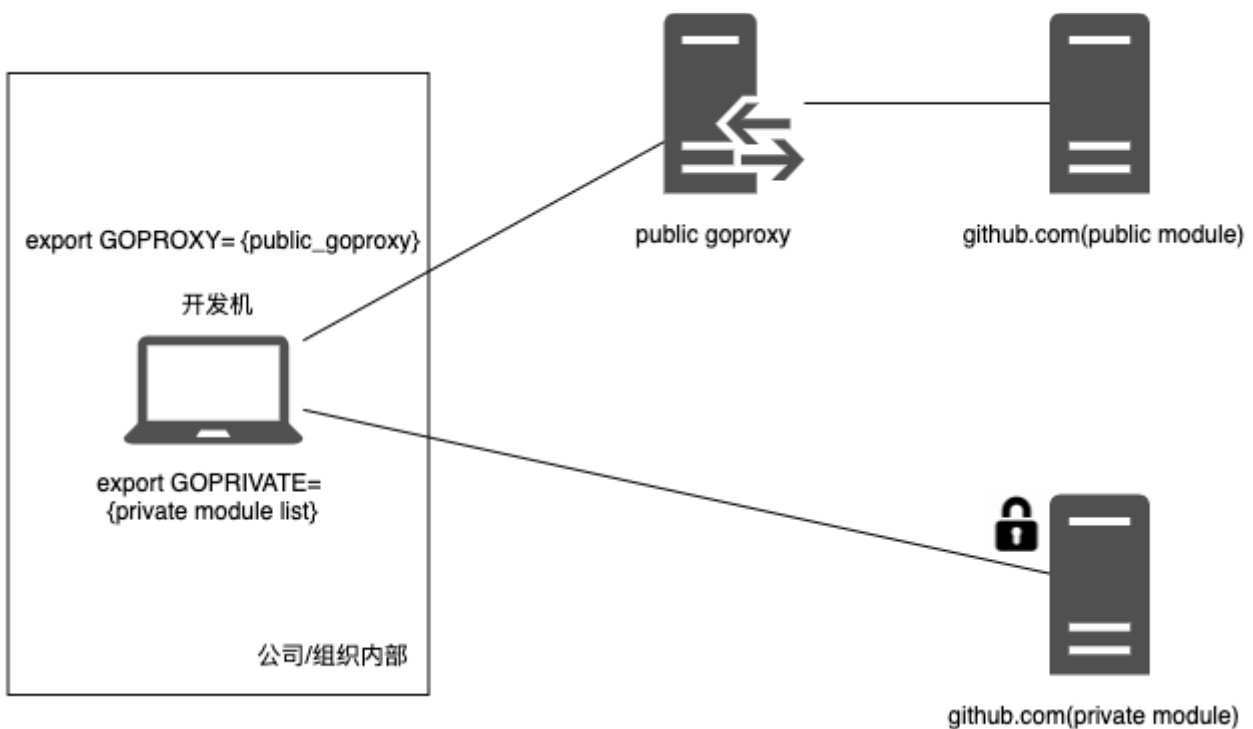
## 拉取私有 module 的需求与参考方案

Go 1.11 版本引入 Go Module 构建模式后，用 Go 命令拉取项目依赖的公共 Go Module，已不再是“痛点”，我们只需要在每个开发机上为环境变量 GOPROXY，配置一个高效好用的公共 GOPROXY 服务，就可以轻松拉取所有公共 Go Module 了：



但随着公司内 Go 使用者和 Go 项目的增多，“重造轮子”的问题就出现了。抽取公共代码放入一个独立的、可被复用的内部私有仓库成为了必然，这样我们就有了**拉取私有 Go Module 的需求**。

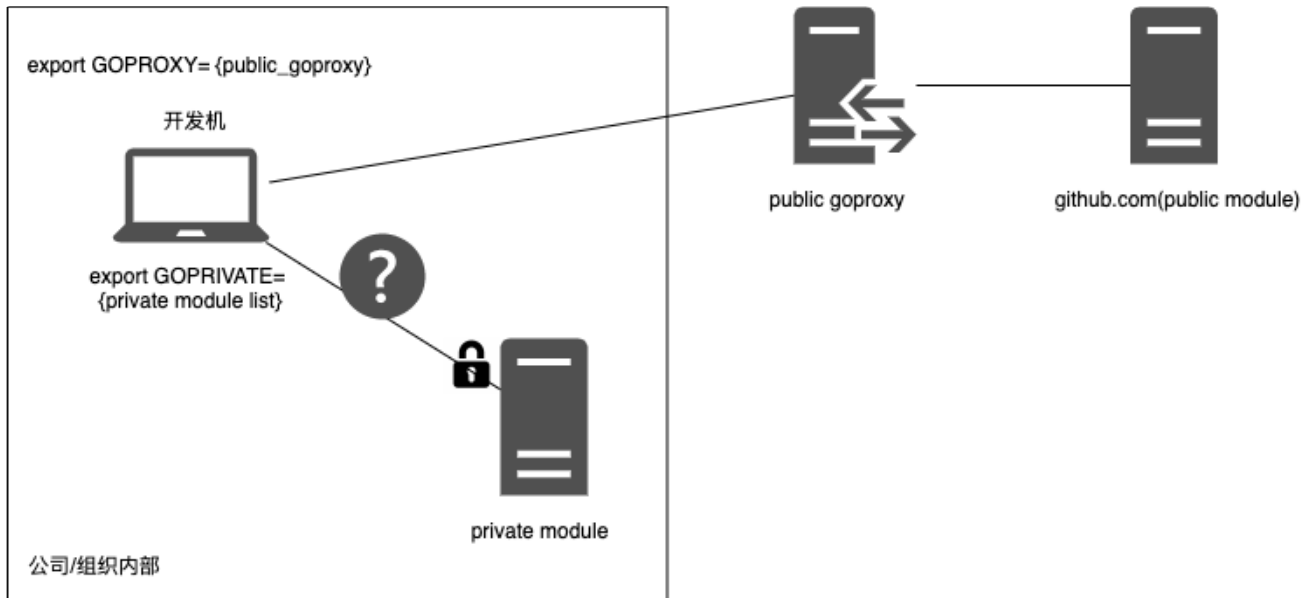
一些公司或组织的所有代码，都放在公共 vcs 托管服务商那里（比如 github.com），私有 Go Module 则直接放在对应的公共 vcs 服务的 private repository（私有仓库）中。如果你的公司也是这样，那么拉取托管在公共 vcs 私有仓库中的私有 Go Module，也很容易，见下图：



也就是说，只要我们在每个开发机上，配置公共 GOPROXY 服务拉取公共 Go Module，同时再把私有仓库配置到 GOPRIVATE 环境变量，就可以了。这样，所有私有 module 的拉取，都会直连代码托管服务器，不会走 GOPROXY 代理服务，也不会去 GOSUMDB 服务器做 Go 包的 hash 值校验。

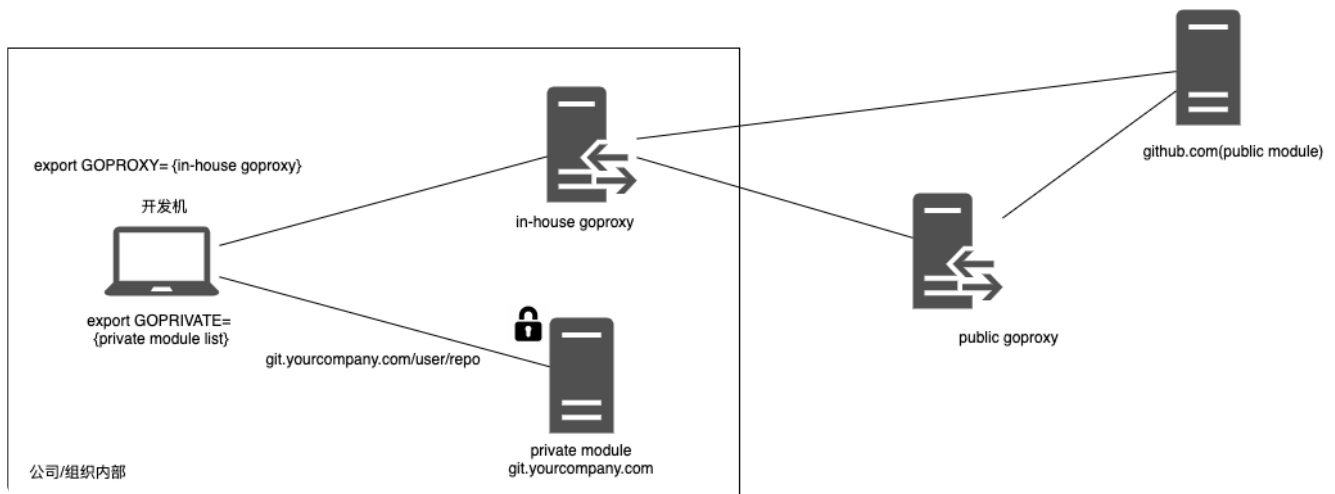
当然，这个方案有一个前提，那就是每个开发人员都需要具有访问公共 vcs 服务上的私有 Go Module 仓库的权限，凭证的形式不限，可以是 basic auth 的 user 和 password，也可以是 personal access token（类似 GitHub 那种），只要按照公共 vcs 的身份认证要求提供就可以了。

不过，更多的公司 / 组织，可能会将私有 Go Module 放在公司 / 组织内部的 vcs（代码版本控制）服务器上，就像下面图中所示：



那么这种情况，我们该如何让 Go 命令，自动拉取内部服务器上的私有 Go Module 呢？这里给出两个参考方案。

**第一个方案是通过直连组织公司内部的私有 Go Module 服务器拉取。**



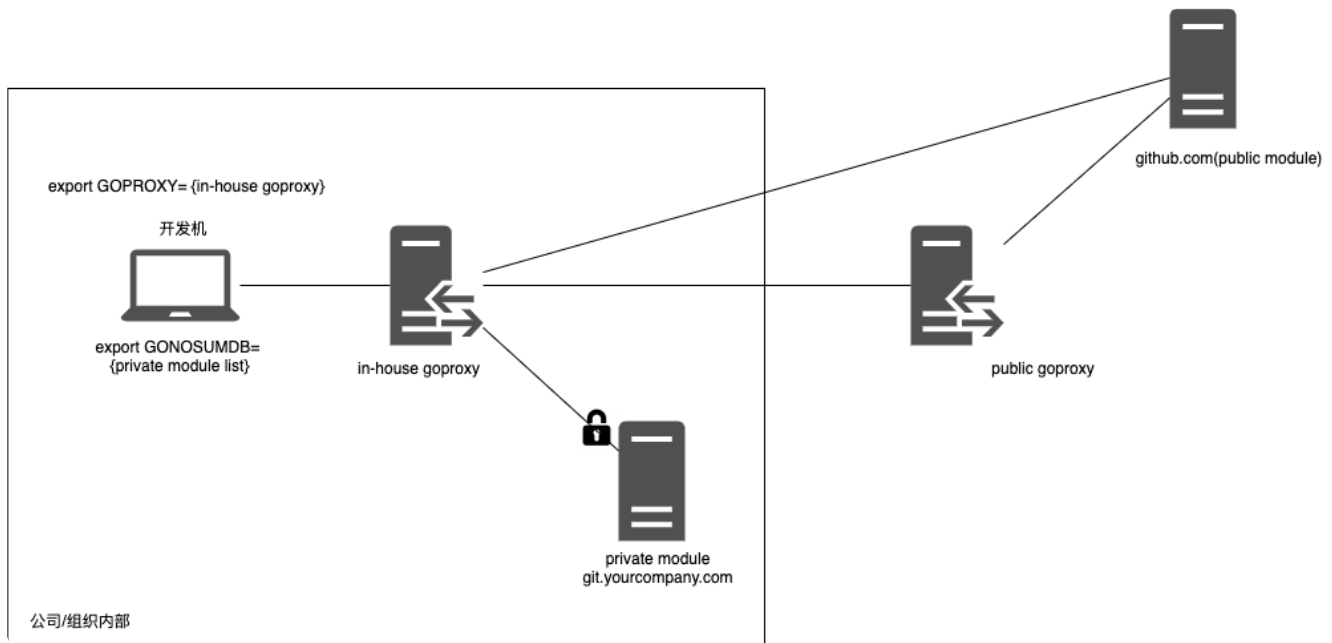
在这个方案中，我们看到，公司内部会搭建一个内部 goproxy 服务（也就是上图中的 in-house goproxy）。这样做有两个目的，一是为那些无法直接访问外网的开发机器，以及 ci 机器提供拉取外部 Go Module 的途径，二来，由于 in-house goproxy 的 cache 的存在，这样做还可以加速公共 Go Module 的拉取效率。

另外，对于私有 Go Module，开发机只需要将它配置到 GOPRIVATE 环境变量中就可以了，这样，Go 命令在拉取私有 Go Module 时，就不会再走 GOPROXY，而会采用直接访问 vcs（如上图中的 git.yourcompany.com）的方式拉取私有 Go Module。



这个方案十分适合内部有完备 IT 基础设施的公司。这类型的公司内部 vcs 服务器都可以通过域名访问（比如 `git.yourcompany.com/user/repo`），因此，公司内部员工可以像访问公共 vcs 服务那样，访问内部 vcs 服务器上的私有 Go Module。

**第二种方案，是将外部 Go Module 与私有 Go Module 都交给内部统一的 GOPROXY 服务去处理：**



在这种方案中，开发者只需要把 GOPROXY 配置为 in-house goproxy，就可以统一拉取外部 Go Module 与私有 Go Module。

但由于 go 命令默认会对所有通过 goproxy 拉取的 Go Module，进行 sum 校验（默认到 `sum.golang.org`），而我们的私有 Go Module 在公共 sum 验证 server 中又没有数据记录。因此，开发者需要将私有 Go Module 填到 GONOSUMDB 环境变量中，这样，go 命令就不会对其进行 sum 校验了。

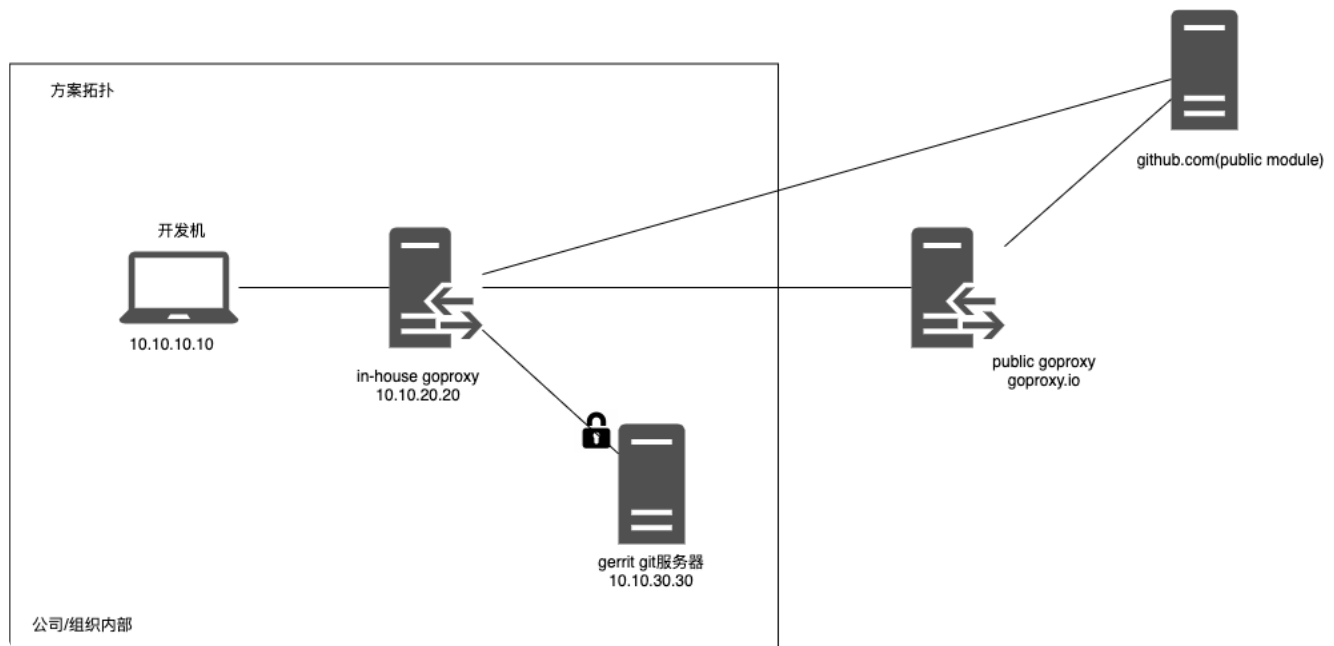
不过这种方案有一处要注意：in-house goproxy 需要拥有对所有 private module 所在 repo 的访问权限，才能保证每个私有 Go Module 都拉取成功。

你可以对比一下上面这两个参考方案，看看你更倾向于哪一个，我推荐第二个方案。在第二个方案中，我们可以**将所有复杂性都交给 in-house goproxy 这个节点**，开发人员可以无差别地拉取公共 module 与私有 module，心智负担降到最低。

那么我们该怎么实现这个方案呢？接下来我就来分析一个可行的实现思路与具体步骤。

## 统一 Goproxy 方案的实现思路与步骤

我们先为后续的方案实现准备一个示例环境，它的拓扑如下图：



### 选择一个 GOPROXY 实现

🔗 [Go module proxy 协议规范](#)发布后，Go 社区出现了很多成熟的 Goproxy 开源实现，比如有最初的 🔗 [athens](#)，还有国内的两个优秀的开源实现：🔗 [goproxy.cn](#)和 🔗 [goproxy.io](#)等。其中，goproxy.io 在官方站点给出了 🔗 [企业内部部署的方法](#)，所以今天我们就基于goproxy.io 来实现我们的方案。

我们在上图中的 in-house goproxy 节点上执行这几个步骤安装 goproxy：


```
1 $mkdir ~/.bin/goproxy
2 $cd ~/.bin/goproxy
3 $git clone https://github.com/goproxyio/goproxy.git
4 $cd goproxy
5 $make
```

📄 复制代码

编译后，我们会在当前的 bin 目录 ( ~/.bin/goproxy/goproxy/bin ) 下看到名为goproxy 的可执行文件。


然后，我们建立 goproxy cache 目录：



 复制代码

```
1 $mkdir /root/.bin/goproxy/goproxy/bin/cache
```

再启动 goproxy :

 复制代码

```
1 $./goproxy -listen=0.0.0.0:8081 -cacheDir=/root/.bin/goproxy/goproxy/bin/cache
2 goproxy.io: ProxyHost https://goproxy.io
```

启动后，goproxy 会在 8081 端口上监听（即便不指定，goproxy 的默认端口也是 8081），指定的上游 goproxy 服务为 goproxy.io。


不过要注意下：goproxy 的这个启动参数并不是最终版本的，这里我仅仅想验证一下 goproxy 是否能按预期工作。我们现在就来实际验证一下。

首先，我们在开发机上配置 GOPROXY 环境变量指向 10.10.20.20:8081：

 复制代码


```
1 // .bashrc
2 export GOPROXY=http://10.10.20.20:8081
```

生效环境变量后，执行下面命令：

 复制代码

```
1 $go get github.com/pkg/errors
```

结果和我们预期的一致，开发机顺利下载了 github.com/pkg/errors 包。我们可以在 goproxy 侧，看到了相应的日志：

 复制代码

```
1 goproxy.io: ----- --- /github.com/pkg/@v/list [proxy]
2 goproxy.io: ----- --- /github.com/pkg/errors/@v/list [proxy]
3 goproxy.io: ----- --- /github.com/@v/list [proxy]
4 goproxy.io: 0.146s 404 /github.com/@v/list
```

```
5 goproxy.io: 0.156s 404 /github.com/pkg/@v/list
6 goproxy.io: 0.157s 200 /github.com/pkg/errors/@v/list
```

在 goproxy 的 cache 目录下，我们也看到了下载并缓存的 github.com/pkg/errors 包：

[复制代码](#)

```
1 $cd /root/.bin/goproxy/goproxy/bin/cache
2 $tree
3 .
4 └─ pkg
5     └─ mod
6         └─ cache
7             └─ download
8                 └─ github.com
9                     └─ pkg
10                        └─ errors
11                            └─ @v
12                                └─ list
13
14 8 directories, 1 file
```

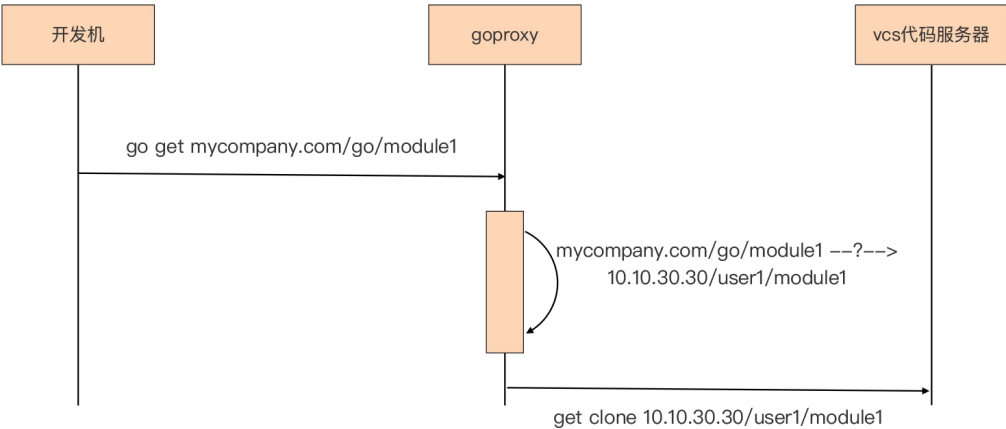
这就标志着我们的 goproxy 服务搭建成功，并可以正常运作了。

## 自定义包导入路径并将其映射到内部的 vcs 仓库

一般公司可能没有为 vcs 服务器分配域名，我们也不能在 Go 私有包的导入路径中放入 ip 地址，因此我们需要给我们的私有 Go Module 自定义一个路径，比如：

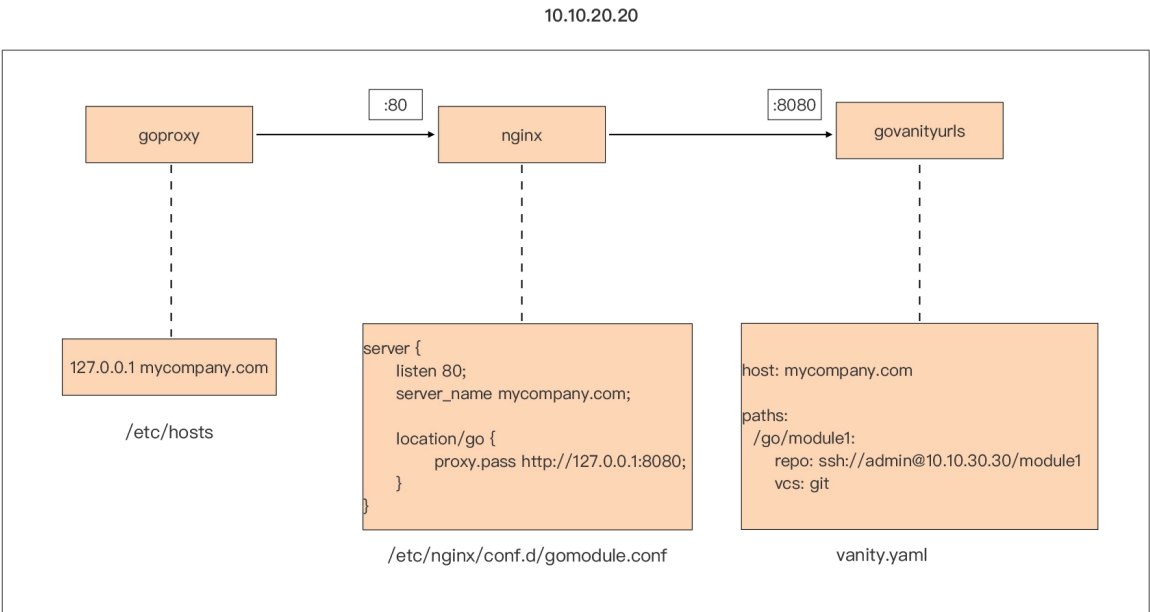
mycompany.com/go/module1。我们统一将私有 Go Module 放在 mycompany.com/go 下面的代码仓库中。

那么，接下来的问题就是，当 goproxy 去拉取 mycompany.com/go/module1 时，应该得到 mycompany.com/go/module1 对应的内部 vcs 上 module1 仓库的地址，这样，goproxy 才能从内部 vcs 代码服务器上下载 module1 对应的代码，具体的过程如下：



那么我们如何实现为私有 module 自定义包导入路径，并将它映射到内部的 vcs 仓库呢？

其实方案不止一种，这里我使用了 Google 云开源的一个名为 [govanityurls](#) 的工具，来为私有 module 自定义包导入路径。然后，结合 govanityurls 和 nginx，我们就可以将私有 Go Module 的导入路径映射为其在 vcs 上的代码仓库的真实地址。具体原理你可以看一下这张图：



首先，goproxy 要想不把收到的拉取私有 Go

Module ( mycompany.com/go/module1 ) 的请求转发给公共代理，需要在其启动参数上做一些手脚，比如下面这个就是修改后的 goproxy 启动命令：

[复制代码](#)

```
1 $./goproxy -listen=0.0.0.0:8081 -cacheDir=/root/.bin/goproxy/goproxy/bin/cache
```

这样，凡是与 -exclude 后面的值匹配的 Go Module 拉取请求，goproxy 都不会转给 goproxy.io，而是直接请求 Go Module 的“源站”。

而上面这张图中要做的，就是将这个“源站”的地址，转换为企业内部 vcs 服务中的一个仓库地址。然后我们假设 mycompany.com 这个域名并不存在（很多小公司没有内部域名解析能力），从图中我们可以看到，我们会在 goproxy 所在节点的 /etc/hosts 中加上这样一条记录：

[复制代码](#)

```
1 127.0.0.1 mycompany.com
```

这样做了后，goproxy 发出的到 mycompany.com 的请求实际上是发向了本机。而上面这图中显示，监听本机 80 端口的正是 nginx，nginx 关于 mycompany.com 这一主机的配置如下：

[复制代码](#)

```
1 // /etc/nginx/conf.d/gomodule.conf
2
3 server {
4     listen 80;
5     server_name mycompany.com;
6
7     location /go {
8         proxy_pass http://127.0.0.1:8080;
9         proxy_redirect off;
10        proxy_set_header Host $host;
11        proxy_set_header X-Real-IP $remote_addr;
12        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
13
14        proxy_http_version 1.1;
15        proxy_set_header Upgrade $http_upgrade;
16        proxy_set_header Connection "upgrade";
```

```
17         }  
18     }
```

我们看到，对于路径为 `mycompany.com/go/xxx` 的请求，`nginx` 将请求转发给了 `127.0.0.1:8080`，而这个服务地址恰恰就是 `govanityurls` 工具监听的地址。

`govanityurls` 这个工具，是前 Go 核心开发团队成员 [@Jaana B.Dogan](#) 开源的一个工具，这个工具可以帮助 Gopher 快速实现自定义 Go 包的 `go get` 导入路径。

`govanityurls` 本身，就好比一个“导航”服务器。当 `go` 命令向自定义包地址发起请求时，实际上是将请求发送给了 `govanityurls` 服务，之后，`govanityurls` 会将请求中的包所在仓库的真实地址（从 `vanity.yaml` 配置文件中读取）返回给 `go` 命令，后续 `go` 命令再从真实的仓库地址获取包数据。

注：`govanityurls` 的安装方法很简单，直接 `go install/go get github.com/GoogleCloudPlatform/govanityurls` 就可以了。  
在我们的示例中，`vanity.yaml` 的配置如下：

```
1 host: mycompany.com  
2  
3 paths:  
4   /go/module1:  
5     repo: ssh://admin@10.10.30.30/module1  
6     vcs: git
```

[复制代码](#)

也就是说，当 `govanityurls` 收到 `nginx` 转发的请求后，会将请求与 `vanity.yaml` 中配置的 `module` 路径相匹配，如果匹配 ok，就会将该 `module` 的真实 `repo` 地址，通过 `go` 命令期望的应答格式返回。在这里我们看到，`module1` 对应的真实 `vcs` 上的仓库地址为：  
`ssh://admin@10.10.30.30/module1`。

所以，`goproxy` 会收到这个地址，并再次向这个真实地址发起请求，并最终将 `module1` 缓存到本地 `cache` 并返回给客户端。

## 开发机（客户端）的设置

前面示例中，我们已经将开发机的 GOPROXY 环境变量，设置为 goproxy 的服务地址。但我们说过，凡是通过 GOPROXY 拉取的 Go Module，go 命令都会默认把它的 sum 值放到公共 GOSUM 服务器上去校验。

但我们实质上拉取的是私有 Go Module，GOSUM 服务器上并没有我们的 Go Module 的 sum 数据。这样就会导致 go build 命令报错，无法继续构建过程。

因此，开发机客户端还需要将 mycompany.com/go，作为一个值设置到 GONOSUMDB 环境变量中：

```
1 export GONOSUMDB=mycompany.com/go
```

[复制代码](#)

这个环境变量配置一旦生效，就相当于告诉 go 命令，凡是与 mycompany.com/go 匹配的 Go Module，都不需要在做 sum 校验了。

到这里，我们就实现了拉取私有 Go Module 的方案。

## 方案的“不足”

当然这个方案并不是完美的，它也有自己的不足的地方：

### 第一点：开发者还是需要额外配置 GONOSUMDB 变量。

由于 Go 命令默认会对从 GOPROXY 拉取的 Go Module 进行 sum 校验，因此我们需要将私有 Go Module 配置到 GONOSUMDB 环境变量中，这就给开发者带来了一个小小的“负担”。

对于这个问题，我的解决建议是：公司内部可以将私有 go 项目都放在一个特定域名下，这样就不需要为每个 go 私有项目单独增加 GONOSUMDB 配置了，只需要配置一次就可以了。

### 第二点：新增私有 Go Module，vanity.yaml 需要手工同步更新。

这是这个方案最不灵活的地方了，由于目前 `govanityurls` 功能有限，针对每个私有 Go Module，我们可能都需要单独配置它对应的 vcs 仓库地址，以及获取方式（git、svn or hg）。

关于这一点，我的建议是：在一个 vcs 仓库中管理多个私有 Go Module。相比于最初 go 官方建议的一个 repo 只管理一个 module，新版本的 go 在 [一个 repo 下管理多个 Go Module](#) 方面，已经有了长足的进步，我们已经可以通过 repo 的 tag 来区别同一个 repo 下的不同 Go Module。

不过对于一个公司或组织来说，这点额外工作与得到的收益相比，应该也不算什么！

### 第三点：无法划分权限。

在讲解上面的方案的时候我们也提到过，`goproxy` 所在节点需要具备访问所有私有 Go Module 所在 vcs repo 的权限，但又无法对 go 开发者端做出有差别授权，这样，只要是 `goproxy` 能拉取到的私有 Go Module，go 开发者都能拉取到。

不过对于多数公司而言，内部所有源码原则上都是企业内部公开的，这个问题似乎也不大。如果觉得这是个问题，那么只能使用前面提到的第一个方案，也就是直连私有 Go Module 的源码服务器的方案了。

## 小结

好了，今天的加餐讲到这里就结束了。今天我们针对前期专栏反馈较多的有关 Go Module 的两个问题，进行了逐一分析，给出了各自的可行的方案。

针对导入本地 Go Module 的问题，在 Go 1.18 版本未发布之前，最好的方法就是使用 `replace` “大法”，通过“伪造”的 `go.mod` 让 go 命令优先使用项目依赖的 Go Module 的本地版本。不过这个方案，也会给开发人员协作方面带去一些额外负担，要想完美解决这一问题，还需要等待加入了 Go 工作区机制的 Go 1.18 版本。

无论大厂小厂，当对 Go 的使用逐渐深入，接纳 Go 的人以及 Go 项目逐渐增多后，拉取私有 Go Module 这样的问题肯定会摆到桌面上来。这一讲我们介绍了直连私有 Go Module 源码服务器和使用统一 GOPROXY 代理两种方案，我推荐你使用第二种方案，可以降低开发人员拉取私有 module 的心智负担。



## 思考题


针对我们这一讲提到的两个问题，你是否有自己的不同的解决方案呢？如果有，欢迎在留言区分享一下你采用的方案。

欢迎把这一节课分享给更多感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独订阅本课程，你将得 20 元

 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 35 | 即学即练：如何实现一个轻量级线程池？

### 更多学习推荐

## 2021 最新大厂 Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

限量免费领取 



## 精选留言 (2)

[写留言](#)**罗杰**

2022-01-21

一直使用的都是 replace，还好我们的代码不是太多，人员也少，还没有自己造出比较好用的轮子，所以 replace 的都是本仓库内的代码，仓库内包含公共库和服务。期待 1.18 能尽可能降低开发者的心智负担，replace 我刚开始用的时候，理解起来还是非常费劲的。虽然比 gopath 好了不少，但是 go 在引包的这一块相比简单哲学还是不够友好。

展开 ▾

**哈哈哈哈哈**

2022-01-21

好耶%\*%

