## **Go的代码管理从最早的GOPATH模式逐步演进到目前最流行的Go Module模式，目前最新Go 1.18支持工作区模式，可以本地同时开发多个互相依赖的Go Module。**

## **前言**

Go 1.18除了引入泛型(generics)、模糊测试(Fuzzing)之外，另外一个重大功能是引入了工作区模式(workspace mode)。

Go官方团队的*Beth Brown*于2022.04.05在官网上专门写了一篇博文，详细介绍了workspace模式的使用场景和最佳实践。

本人针对官方原文做了一个翻译，以飨读者。**同时在本文最后，附上了对workspace模式的入门介绍。**

## **原文翻译**

Go官方团队*Beth Brown*

2022.04.05

Go 1.18新增了工作区模式(workspace mode)，让你可以同时跨多个Go Module进行开发。

你可以从**download[1]**地址下载Go 1.18，**release notes[2]**有更多关于Go 1.18的变化细节。

## **工作区(workspaces)**

Go 1.18引入的**工作区[3]**模式，可以让你不用修改每个Go Module的go.mod，就能同时跨多个Go Module进行开发。工作区里的每个Go Module在解析依赖的时候都被当做根Module。

在Go 1.18以前，如果遇到以下场景：Module A新增了一个feature，Module B需要使用Module A的这个新feature，你有2种方案：

* 发布Module A的修改到代码仓库，Module B更新依赖的Module A的版本即可
* 修改Module B的go.mod，使用replace指令把对Module A的依赖指向你本地未发布的Module A所在目录。等Module A发布后，在发布Module B的时候，再删除Module B的go.mod文件里的replace指令。

有了Go工作区模式之后，针对上述场景，我们有了更为简单的方案：你可以在工作区目录维护一个go.work文件来管理你的所有依赖。go.work里的use和replace指令会覆盖工作区目录下的每个Go Module的go.mod文件，因此没有必要去修改Go Module的go.mod文件了。

### **go work init**

你可以使用go work init来创建一个workspace，go work init 的语法如下所示：

go work init **[**moddirs**]**

moddirs是Go Module所在的本地目录。如果有多个Go Module，就用空格分开。如果go work init后面没有参数，会创建一个空的workspace。

执行go work init后会生成一个go.work文件，go.work里列出了该workspace需要用到的Go Module所在的目录，workspace目录不需要包含你当前正在开发的Go Module代码。

### **go work use**

如果要给workspace新增Go Module，可以使用如下命令：

go work use **[**-r**]** moddir

或者手动编辑go work文件。

如果带有-r参数，会递归查找-r后面的路径参数下的所有子目录，把所有包含go.mod文件的子目录都添加到go work文件中。

如果某个Go Module的目录已经被加到go.work里了，后面该目录没有go.mod文件了或者该目录被删除了，那对该目录再次执行go work use命令，该目录的use指令会从go.work文件里自动移除。(**注意**：自动移除要从Go 1.18正式版本才会生效，Go 1.18beta1版本有bug，自动删除不会生效)

### **go.work**

go.work的语法和go.mod类似，包含如下3个指令：

* go: go的版本，例如 go 1.18
* use: 添加一个本地磁盘上的Go Module到workspace的主Module集合里。use后面的参数是go.mod文件所在目录相对于workspace目录的相对路径，例如use ./main。use指令不会添加指定目录的子目录下的Go Module到workspace的主Module集合里。
* replace: 和go.mod里的 replace指令类似。go.work里的 replace指令可以替换某个Go Module的特定版本或者所有版本的内容。

## **使用场景和最佳实践**

Workspace使用起来很灵活，接下来会介绍最常见的几种使用场景及其最佳实践。

### **使用场景1**

**给上游模块新增feature，然后在你的Module里使用这个新feature**

1. 为你的workspace(工作区)创建一个目录。
2. Clone一份你要修改的上游模块的代码到本地。
3. 本地修改上游模块的代码，增加新的feature。
4. 在workspace目录运行命令go work init [path-to-upstream-mod-dir]。
5. 为了使用上游模块的新feature，修改你自己的Go Module代码。
6. 在workspace目录运行命令 go work use [path-to-your-module] 。  
   go work use 命令会添加你的Go Module的路径到 go.work 文件里：  
   go 1.18  
     
   use (  
   ./path-to-upstream-mod-dir  
   ./path-to-your-module  
   )
7. 运行和测试你的Go Module。
8. 发布上游模块的新feature。
9. 发布你自己的Go Module代码。

### **使用场景2**

**同一个代码仓库里有多个互相依赖的Go Module**

当我们在同一个代码仓库里开发多个互相依赖的Go Module时，我们可以使用go.work，而不是在go.mod里使用replace指令。

1. 为你的workspace(工作区)创建一个目录。
2. Clone仓库里的代码到你本地。代码存放的位置不一定要放在工作区目录下，因为你可以在go.work里使用use指令来指定Module的相对路径。
3. 在工作区目录运行 go work init [path-to-module-one] [path-to-module-two] 命令。  
   示例: 你正在开发 example.com/x/tools/groundhog 这个Module，该Module依赖 example.com/x/tools 下的其它Module。  
   你Clone仓库里的代码到你本地，然后在工作区目录运行命令 go work init tools tools/groundhog 。  
   go.work 文件里的内容如下所示：

go 1.18  
  
use (  
./tools  
./tools/groundhog  
)  
  
tools路径下其它Module的本地代码修改都会被 tools/groundhog 直接使用到。

### **使用场景3：切换依赖配置**

如果要测试你开发的代码在不同的本地依赖配置下的场景，你有2种选择：

* 创建多个workspace，每个workspace使用各自的go.work文件，每个go.work里指定一个版本的路径。
* 创建一个workspace，在go.work里注释掉你不想要的use指令。

对于创建多个workspace的方案：

1. 为每个workspace创建独立的目录。比如你开发的代码依赖了example.com/util这个Go Module，但是想测试example.com/util2个版本的区别，你可以创建2个workspace目录。
2. 在各自的workspace目录运行 go work init 来初始化workspace。
3. 在各自的workspace目录运行 go work use [path-to-dependency]来添加依赖的Go Module特定版本的目录。
4. 在各自的workspace目录运行 go run [path-to-your-module] 来测试go.work里指定的依赖版本。

对于使用同一个workspace的方案，可以直接编辑go.work文件，修改use指令后面的目录地址即可。

### **还在使用GOPATH模式存放代码?**

也许使用工作区会改变你的想法。 GOPATH 用户可以使用位于其GOPATH 目录底部的go.work 文件来解决他们的依赖关系。 工作区的目标不是完全重建 GOPATH 工作流程，而是创建一个可以共享 GOPATH 的便利和Go Module优点的设置。

为GOPATH创建工作区：

* 在GOPATH目录的根目录下运行 go work init。
* 要在工作区中使用本地模块或特定版本作为依赖项，请运行go work use [path-to-module]。
* 要替换Go Module go.mod 文件中的现有依赖项，请使用 go work replace [path-to-module]。
* 要添加 GOPATH 或任何目录中的所有Module，请运行 go work use -r 命令，该命令以递归方式将带有 go.mod 文件的目录添加到你的工作区。 如果一个目录没有 go.mod 文件，或者该目录不再存在，那该目录的 use 指令将从你的 go.work 文件中自动移除。

注意：如果你的工程里没有go.mod文件，但是你想把它加入到workspace里，你需要进入你的工程目录，执行go mod init来添加go.mod，然后运行 go work use [path-to-module] 来把你的工程添加到workspace中。

## **Workspace命令**

除了 go work init 和 go work use，Go 1.18还为Workspace引入了如下命令：

* go work sync: 把go.work文件里的依赖同步到workspace包含的Module的go.mod文件中。
* go work edit: 提供了用于修改go.work的命令行接口，主要是给工具或脚本使用。

编译命令以及go mod的一些子命令会检查GOWORK环境变量，用于判断当前go命令是否处于工作区模式下。

如果GOWORK环境变量的值是以.work结尾的文件路径，则启用工作区模式。

要确定目前正在使用哪个go.work文件，可以运行go env GOWORK命令。如果go命令不在工作区模式，那go env GOWORK的输出结果为空。

工作区模式开启后，go.work 文件会被解析，用来确定工作区模式下的3个参数：

* Go版本
* workspace下的Module的所在目录
* 被替换的Module的信息

工作区模式下还可以尝试如下命令：

go work init

go work sync

go work use

go list

go build

go test

go run

go vet

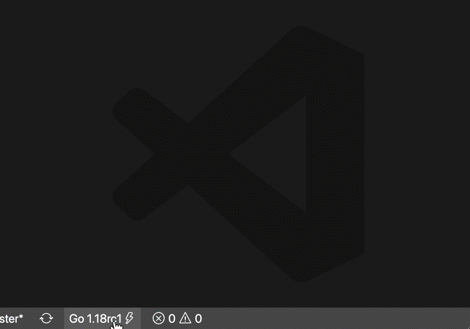
## **代码编辑器体验优化**

对于Go的语言服务器**gopls[4]** 和**VSCode Go 插件[5]** 的升级，我们感到非常兴奋。这可以让我们在兼容LSP(Langugage Server Protocol，语言服务器协议)的代码编辑器上使用Go workspace的体验非常棒。

gopls的 **0.8.1[6]** 版本为go.work文件引入了代码诊断、代码补全、代码格式化和提示悬浮。你可以在任何兼容LSP的代码编辑器上享受到gopls的新功能。

### **代码编辑器相关的使用细节**

* 最新的 **vscode-go 插件[7]** 支持通过编辑器左下角的Go状态栏的快速访问菜单访问go.work文件。



Access the go.work file via the Go status bar’s Quick Pick menu

* **GoLand[8]** 支持Workspace工作区模式，也有计划为go.work文件新增语法高亮和代码补全功能。

更多关于不同编辑器使用gopls的信息，可以参考 gopls**文档[9]**.

## **下一步做什么？**

* 下载和安装**Go 1.18[10]**.
* 尝试跟着我们的**Go workspace教程[11]**来学习使用**workspaces[12]**。
* 如果你有关于Go Workspace的任何问题或者建议，请提交**issue[13]**。
* 阅读 **workspace说明文档[14]**。
* 探索更多Go命令，包括 go work init, go work sync 等等。