



下载APP



14 | 项目管理：如何编写高质量的Makefile？

2021-06-26 孔令飞

《Go 语言项目开发实战》

课程介绍 >



讲述：孔令飞

时长 14:54 大小 13.66M



你好，我是孔令飞。今天我们来聊聊如何编写高质量的 Makefile。

我们在 [第 10 讲](#) 学习过，要写出一个优雅的 Go 项目，不仅仅是要开发一个优秀的 Go 应用，而且还要能够高效地管理项目。有效手段之一，就是通过 Makefile 来管理我们的项目，这就要求我们要为项目编写 Makefile 文件。

在和其他开发同学交流时，我发现大家都认可 Makefile 强大的项目管理能力，也会自己编写 Makefile。但是其中的一些人项目管理做得并不好，我和他们进一步交流后发现，[这些](#) 同学在用 Makefile 简单的语法重复编写一些低质量 Makefile 文件，根本没有把 Makefile 的功能充分发挥出来。

下面给你举个例子，你就会理解低质量的 Makefile 文件是什么样的了。

 复制代码

```
1 build: clean vet
2   @mkdir -p ./Role
3   @export GOOS=linux && go build -v .
4
5 vet:
6   go vet ./...
7
8 fmt:
9   go fmt ./...
10
11 clean:
12   rm -rf dashboard
```

上面这个 Makefile 存在不少问题。例如：功能简单，只能完成最基本的编译、格式化等操作，像构建镜像、自动生成代码等一些高阶的功能都没有；扩展性差，没法编译出可在 Mac 下运行的二进制文件；没有 Help 功能，使用难度高；单 Makefile 文件，结构单一，不适合添加一些复杂的管理功能。

所以，我们不光要编写 Makefile，还要编写高质量的 Makefile。那么如何编写一个高质量的 Makefile 呢？我觉得，可以通过以下 4 个方法来实现：

1. 打好基础，也就是熟练掌握 Makefile 的语法。
2. 做好准备工作，也就是提前规划 Makefile 要实现的功能。
3. 进行规划，设计一个合理的 Makefile 结构。
4. 掌握方法，用好 Makefile 的编写技巧。

那么接下来，我们就详细看看这些方法。

熟练掌握 Makefile 语法

工欲善其事，必先利其器。编写高质量 Makefile 的第一步，便是熟练掌握 Makefile 的核心语法。

因为 Makefile 的语法比较多，我把一些建议你重点掌握的语法放在了近期会更新的特别放送中，包括 Makefile 规则语法、伪目标、变量赋值、条件语句和 Makefile 常用函数等等。

如果你想更深入、全面地学习 Makefile 的语法，我推荐你学习陈皓老师编写的 [《跟我一起写 Makefile》](#) (PDF 重制版)。

规划 Makefile 要实现的功能

接着，我们需要规划 Makefile 要实现的功能。提前规划好功能，有利于你设计 Makefile 的整体结构和实现方法。

不同项目拥有不同的 Makefile 功能，这些功能中一小部分是通过目标文件来实现的，但更多的功能是通过伪目标来实现的。对于 Go 项目来说，虽然不同项目集成的功能不一样，但绝大部分项目都需要实现一些通用的功能。接下来，我们就来看看，在一个大型 Go 项目中 Makefile 通常可以实现的功能。

下面是 IAM 项目的 Makefile 所集成的功能，希望会对你日后设计 Makefile 有一些帮助。

[复制代码](#)

```
1 $ make help
2
3 Usage: make <TARGETS> <OPTIONS> ...
4
5 Targets:
6   # 代码生成类命令
7   gen                Generate all necessary files, such as error code files.
8
9   # 格式化类命令
10  format             Gofmt (reformat) package sources (exclude vendor dir if e
11
12  # 静态代码检查
13  lint               Check syntax and styling of go sources.
14
15  # 测试类命令
16  test               Run unit test.
17  cover              Run unit test and get test coverage.
18
19  # 构建类命令
20  build              Build source code for host platform.
21  build.multiarch    Build source code for multiple platforms. See option PLAT
22
23  # Docker镜像打包类命令
24  image              Build docker images for host arch.
25  image.multiarch    Build docker images for multiple platforms. See option PL
26  push               Build docker images for host arch and push images to regi
27  push.multiarch     Build docker images for multiple platforms and push image
```

```

28
29 # 部署类命令
30 deploy          Deploy updated components to development env.
31
32 # 清理类命令
33 clean           Remove all files that are created by building.
34
35 # 其他命令, 不同项目会有区别
36 release        Release iam
37 verify-copyright Verify the boilerplate headers for all files.
38 ca            Generate CA files for all iam components.
39 install        Install iam system with all its components.
40 swagger        Generate swagger document.
41 tools          install dependent tools.
42
43 # 帮助命令
44 help          Show this help info.
45
46 # 选项
47 Options:
48     DEBUG        Whether to generate debug symbols. Default is 0.
49     BINS         The binaries to build. Default is all of cmd.
50                 This option is available when using: make build/build.multiarch
51                 Example: make build BINS="iam-apiserver iam-authz-server"
52     ...

```

更详细的命令, 你可以在 IAM 项目仓库根目录下执行make help查看。

通常而言, Go 项目的 Makefile 应该实现以下功能: 格式化代码、静态代码检查、单元测试、代码构建、文件清理、帮助等等。如果通过 docker 部署, 还需要有 docker 镜像打包功能。因为 Go 是跨平台的语言, 所以构建和 docker 打包命令, 还要能够支持不同的 CPU 架构和平台。为了能够更好地控制 Makefile 命令的行为, 还需要支持 Options。

为了方便查看 Makefile 集成了哪些功能, 我们需要支持 help 命令。help 命令最好通过解析 Makefile 文件来输出集成的功能, 例如:

 复制代码

```

1 ## help: Show this help info.
2 .PHONY: help
3 help: Makefile
4     @echo -e "\nUsage: make <TARGETS> <OPTIONS> ... \n\nTargets:"
5     @sed -n 's/^##//p' $< | column -t -s ':' | sed -e 's/^/ /'
6     @echo "$$USAGE_OPTIONS"

```

上面的 help 命令，通过解析 Makefile 文件中的##注释，获取支持的命令。通过这种方式，我们以后新加命令时，就不用再对 help 命令进行修改了。

你可以参考上面的 Makefile 管理功能，结合自己项目的需求，整理出一个 Makefile 要实现的功能列表，并初步确定实现思路和方法。做完这些，你的编写前准备工作就基本完成了。

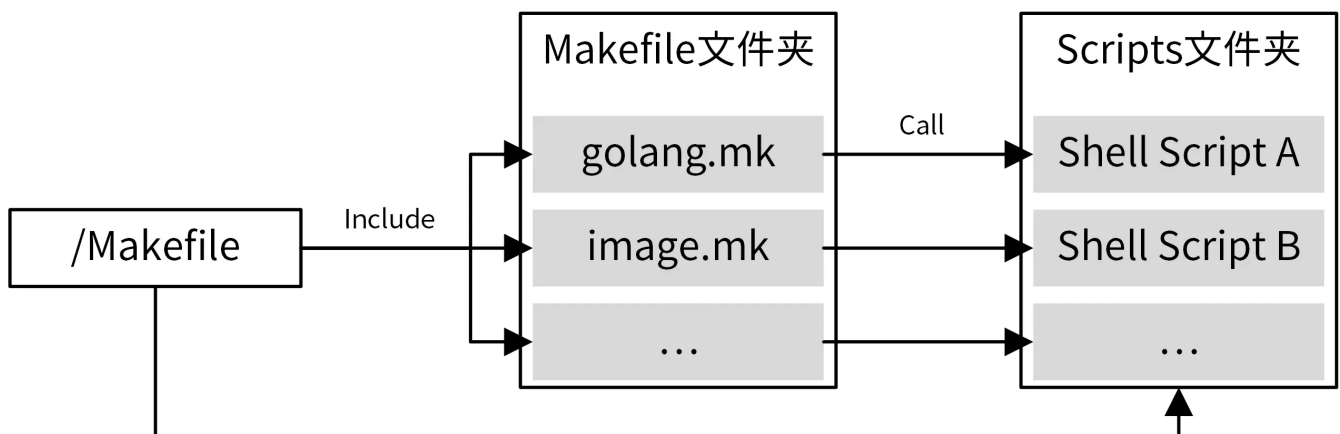
设计合理的 Makefile 结构

设计完 Makefile 需要实现的功能，接下来我们就进入 Makefile 编写阶段。编写阶段的第一步，就是设计一个合理的 Makefile 结构。

对于大型项目来说，需要管理的内容很多，所有管理功能都集成在一个 Makefile 中，可能会导致 Makefile 很大，难以阅读和维护，所以**建议采用分层的设计方法，根目录下的 Makefile 聚合所有的 Makefile 命令，具体实现则按功能分类，放在另外的 Makefile 中。**

我们经常会在 Makefile 命令中集成 shell 脚本，但如果 shell 脚本过于复杂，也会导致 Makefile 内容过多，难以阅读和维护。并且在 Makefile 中集成复杂的 shell 脚本，编写体验也很差。对于这种情况，**可以将复杂的 shell 命令封装在 shell 脚本中，供 Makefile 直接调用，而一些简单的命令则可以直接集成在 Makefile 中。**

所以，最终我推荐的 Makefile 结构如下：



也可以直接调用shell script

在上面的 Makefile 组织方式中，根目录下的 Makefile 聚合了项目所有的管理功能，这些管理功能通过 Makefile 伪目标的方式实现。同时，还将这些伪目标进行分类，把相同类别的伪目标放在同一个 Makefile 中，这样可以使得 Makefile 更容易维护。对于复杂的命令，则编写成独立的 shell 脚本，并在 Makefile 命令中调用这些 shell 脚本。

举个例子，下面是 IAM 项目的 Makefile 组织结构：

[复制代码](#)

```
1 |— Makefile
2 |— scripts
3 |   |— gendoc.sh
4 |   |— make-rules
5 |       |— gen.mk
6 |       |— golang.mk
7 |       |— image.mk
8 |       |— ...
9 |— ...
```

我们将相同类别的操作统一放在 scripts/make-rules 目录下的 Makefile 文件中。Makefile 的文件名参考分类命名，例如 golang.mk。最后，在 /Makefile 中 include 这些 Makefile。

为了跟 Makefile 的层级相匹配，golang.mk 中的所有目标都按 go.xxx 这种方式命名。通过这种命名方式，我们可以很容易分辨出某个目标完成什么功能，放在什么文件里，这在复杂的 Makefile 中尤其有用。以下是 IAM 项目根目录下，Makefile 的内容摘录，你可以看一看，作为参考：

[复制代码](#)

```
1 include scripts/make-rules/golang.mk
2 include scripts/make-rules/image.mk
3 include scripts/make-rules/gen.mk
4 include scripts/make-rules/...
5
6 ## build: Build source code for host platform.
7 .PHONY: build
8 build:
9     @$(MAKE) go.build
10
11 ## build.multiarch: Build source code for multiple platforms. See option PLATF
12 .PHONY: build.multiarch
13 build.multiarch:
```

```
14    @$(MAKE) go.build.multiarch
15
16  ## image: Build docker images for host arch.
17  .PHONY: image
18  image:
19    @$(MAKE) image.build
20
21  ## push: Build docker images for host arch and push images to registry.
22  .PHONY: push
23  push:
24    @$(MAKE) image.push
25
26  ## ca: Generate CA files for all iam components.
27  .PHONY: ca
28  ca:
29    @$(MAKE) gen.ca
```

另外，一个合理的 Makefile 结构应该具有前瞻性。也就是说，要在不改变现有结构的情况下，接纳后面的新功能。这就需要你整理好 Makefile 当前要实现的功能、即将要实现的功能和未来可能会实现的功能，然后基于这些功能，利用 Makefile 编程技巧，编写可扩展的 Makefile。

这里需要你注意：上面的 Makefile 通过 .PHONY 标识定义了大量的伪目标，定义伪目标一定要加 .PHONY 标识，否则当有同名的文件时，伪目标可能不会被执行。

掌握 Makefile 编写技巧

最后，在编写过程中，你还需要掌握一些 Makefile 的编写技巧，这些技巧可以使你编写的 Makefile 扩展性更强，功能更强大。

接下来，我会把自己长期开发过程中积累的一些 Makefile 编写经验分享给你。这些技巧，你需要在实际编写中多加练习，并形成编写习惯。

技巧 1：善用通配符和自动变量

Makefile 允许对目标进行类似正则运算的匹配，主要用到的通配符是%。通过使用通配符，可以使不同的目标使用相同的规则，从而使 Makefile 扩展性更强，也更简洁。

我们的 IAM 实战项目中，就大量使用了通配符%，例如：go.build.%、ca.gen.%、deploy.run.%、tools.verify.%、tools.install.%等。

这里，我们来看一个具体的例子，`tools.verify.%`（位于 [scripts/make-rules/tools.mk](#) 文件中）定义如下：

[复制代码](#)

```
1 tools.verify.%:  
2     @if ! which $* &>/dev/null; then $(MAKE) tools.install.$*; fi
```

`make tools.verify.swagger, make tools.verify.mockgen` 等均可以使用上面定义的规则，`%` 分别代表了 `swagger` 和 `mockgen`。

如果不使用 `%`，则我们需要分别为 `tools.verify.swagger` 和 `tools.verify.mockgen` 定义规则，很麻烦，后面修改也困难。

另外，这里也能看出 `tools.verify.%` 这种命名方式的好处：`tools` 说明依赖的定义位于 `scripts/make-rules/tools.mk` Makefile 中；`verify` 说明 `tools.verify.%` 伪目标属于 `verify` 分类，主要用来验证工具是否安装。通过这种命名方式，你可以很容易地知道目标位于哪个 Makefile 文件中，以及想要完成的功能。

另外，上面的定义中还用到了自动变量 `$*`，用来指代被匹配的值 `swagger`、`mockgen`。


技巧 2：善用函数

Makefile 自带的函数能够帮助我们实现很多强大的功能。所以，在我们编写 Makefile 的过程中，如果有功能需求，可以优先使用这些函数。我把常用的函数以及它们实现的功能整理在了 [Makefile 常用函数列表](#) 中，你可以参考下。

IAM 的 Makefile 文件中大量使用了上述函数，如果你想查看这些函数的具体使用方法和场景，可以参考 IAM 项目的 Makefile 文件 [make-rules](#)。


技巧 3：依赖需要用到的工具

如果 Makefile 某个目标的命令中用到了某个工具，可以将该工具放在目标的依赖中。这样，当执行该目标时，就可以指定检查系统是否安装该工具，如果没有安装则自动安装，从而实现更高度度的自动化。例如，`/Makefile` 文件中，`format` 伪目标，定义如下：

 复制代码


```
1 .PHONY: format
2 format: tools.verify.golines tools.verify.goimports
3   @echo "=====> Formating codes"
4   @$(FIND) -type f -name '*.go' | $(XARGS) gofmt -s -w
5   @$(FIND) -type f -name '*.go' | $(XARGS) goimports -w -local $(ROOT_PACKAGE)
6   @$(FIND) -type f -name '*.go' | $(XARGS) golines -w --max-len=120 --reformat
```

你可以看到，format 依赖tools.verify.golines tools.verify.goimports。我们再来看下tools.verify.golines的定义：

 复制代码

```
1 tools.verify.%:
2   @if ! which $* &>/dev/null; then $(MAKE) tools.install.$*; fi
```

再来看下tools.install.\$*规则：

 复制代码

```
1 .PHONY: install.golines
2 install.golines:
3   @$(GO) get -u github.com/segmentio/golines
```

通过tools.verify.%规则定义，我们可以知道，tools.verify.%会先检查工具是否安装，如果没有安装，就会执行tools.install.\$*来安装。如此一来，当我们执行tools.verify.%目标时，如果系统没有安装 golines 命令，就会自动调用go get安装，提高了 Makefile 的自动化程度。

技巧 4：把常用功能放在 /Makefile 中，不常用的放在分类 Makefile 中

一个项目，尤其是大型项目，有很多需要管理的地方，其中大部分都可以通过 Makefile 实现自动化操作。不过，为了保持 /Makefile 文件的整洁性，我们不能把所有的命令都添加在 /Makefile 文件中。

一个比较好的建议是，将常用功能放在 /Makefile 中，不常用的放在分类 Makefile 中，并在 /Makefile 中 include 这些分类 Makefile。

例如，IAM 项目的 /Makefile 集成了 format、lint、test、build 等常用命令，而将 gen.errcode.code、gen.errcode.doc 这类不常用的功能放在 scripts/make-rules/gen.mk 文件中。当然，我们也可以直接执行 `make gen.errcode.code` 来执行 `make gen.errcode.code` 伪目标。通过这种方式，既可以保证 /Makefile 的简洁、易维护，又可以通过 make 命令来运行伪目标，更加灵活。

技巧 5：编写可扩展的 Makefile


什么叫可扩展的 Makefile 呢？在我看来，可扩展的 Makefile 包含两层含义：

1. 可以在不改变 Makefile 结构的情况下添加新功能。
2. 扩展项目时，新功能可以自动纳入到 Makefile 现有逻辑中。

其中的第一点，我们可以通过设计合理的 Makefile 结构来实现。要实现第二点，就需要我们在编写 Makefile 时采用一定的技巧，例如多用通配符、自动变量、函数等。这里我们来看一个例子，可以让你更好地理解。

在我们 IAM 实战项目的 [golang.mk](#) 中，执行 `make go.build` 时能够构建 cmd/ 目录下的所有组件，也就是说，当有新组件添加时，`make go.build` 仍然能够构建新增的组件，这就实现了上面说的第二点。

具体实现方法如下：

 复制代码

```
1 COMMANDS ?= $(filter-out %.md, $(wildcard ${ROOT_DIR}/cmd/*))
2 BINS ?= $(foreach cmd,${COMMANDS},${notdir ${cmd}})
3
4 .PHONY: go.build
5 go.build: go.build.verify $(addprefix go.build., $(addprefix $(PLATFORM)., $(B
6 .PHONY: go.build.%
7
8 go.build.%:
9     $(eval COMMAND := $(word 2,$(subst ., ,${*})))
10    $(eval PLATFORM := $(word 1,$(subst ., ,${*})))
11    $(eval OS := $(word 1,$(subst _, ,$(PLATFORM))))
12    $(eval ARCH := $(word 2,$(subst _, ,$(PLATFORM))))
13    @echo "=====> Building binary $(COMMAND) $(VERSION) for $(OS) $(ARCH)"
14    @mkdir -p $(OUTPUT_DIR)/platforms/$(OS)/$(ARCH)
15    @CGO_ENABLED=0 GOOS=$(OS) GOARCH=$(ARCH) $(GO) build $(GO_BUILD_FLAGS) -o $(
```

当执行`make go.build`时, 会执行 `go.build` 的依赖 `$(addprefix go.build., $(addprefix $(PLATFORM)., $(BINS)))`, `addprefix` 函数最终返回字符串 `go.build.linux_amd64.iamctl go.build.linux_amd64.iam-authz-server go.build.linux_amd64.iam-apiserver ...`, 这时候就会执行 `go.build.%` 伪目标。

在 `go.build.%` 伪目标中, 通过 `eval`、`word`、`subst` 函数组合, 算出了 `COMMAND` 的值 `iamctl/iam-apiserver/iam-authz-server/...`, 最终通过 `$(ROOT_PACKAGE)/cmd/$(COMMAND)` 定位到需要构建的组件的 `main` 函数所在目录。

上述实现中有两个技巧, 你可以注意下。首先, 通过

[复制代码](#)

```
1 COMMANDS ?= $(filter-out %.md, $(wildcard ${ROOT_DIR}/cmd/*))
2 BINS ?= $(foreach cmd,${COMMANDS},$(notdir ${cmd}))
```

获取到了 `cmd/` 目录下的所有组件名。

接着, 通过使用通配符和自动变量, 自动匹配到 `go.build.linux_amd64.iam-authz-server` 这类伪目标并构建。

可以看到, 想要编写一个可扩展的 `Makefile`, 熟练掌握 `Makefile` 的用法是基础, 更多的是需要我们动脑思考如何去编写 `Makefile`。

技巧 6 : 将所有输出存放在一个目录下, 方便清理和查找

在执行 `Makefile` 的过程中, 会输出各种各样的文件, 例如 `Go` 编译后的二进制文件、测试覆盖率数据等, 我建议你把这些文件统一放在一个目录下, 方便后期的清理和查找。通常我们可以把它们放在 `_output` 这类目录下, 这样清理时就很方便, 只需要清理 `_output` 文件夹就可以, 例如:

[复制代码](#)

```
1 .PHONY: go.clean
2 go.clean:
```

```
3 @echo "=====> Cleaning all build output"
4 @rm -vrf $(OUTPUT_DIR)
```

这里要注意，要用`-rm`，而不是`rm`，防止在没有`_output`目录时，执行`make go.clean`报错。

技巧 7：使用带层级的命名方式

通过使用带层级的命名方式，例如`tools.verify.swagger`，我们可以实现**目标分组管理**。这样做的好处有很多。首先，当 Makefile 有大量目标时，通过分组，我们可以更好地管理这些目标。其次，分组也能方便理解，可以通过组名一眼识别出该目标的功能类别。最后，这样做还可以大大减小目标重名的概率。

例如，IAM 项目的 Makefile 就大量采用了下面这种命名方式。

[复制代码](#)

```
1 .PHONY: gen.run
2 gen.run: gen.clean gen.errcode gen.docgo
3
4 .PHONY: gen.errcode
5 gen.errcode: gen.errcode.code gen.errcode.doc
6
7 .PHONY: gen.errcode.code
8 gen.errcode.code: tools.verify.codegen
9     ...
10 .PHONY: gen.errcode.doc
11 gen.errcode.doc: tools.verify.codegen
12     ...
```

技巧 8：做好目标拆分

还有一个比较实用的技巧：我们要合理地拆分目标。比如，我们可以将安装工具拆分成两个目标：验证工具是否已安装和安装工具。通过这种方式，可以给我们的 Makefile 带来更大的灵活性。例如：我们可以根据需求选择性地执行其中一个操作，也可以两个操作一起执行。

这里来看一个例子：

```
1 gen.errcode.code: tools.verify.codegen
2
3 tools.verify.%:
4     @if ! which $* &>/dev/null; then $(MAKE) tools.install.$*; fi
5
6 .PHONY: install.codegen
7 install.codegen:
8     @$(GO) install ${ROOT_DIR}/tools/codegen/codegen.go
```

[复制代码](#)

上面的 Makefile 中，gen.errcode.code 依赖了 tools.verify.codegen，tools.verify.codegen 会先检查 codegen 命令是否存在，如果不存在，再调用 install.codegen 来安装 codegen 工具。

如果我们的 Makefile 设计是：

```
1 gen.errcode.code: install.codegen
```

[复制代码](#)

那每次执行 gen.errcode.code 都要重新安装 codegen 命令，这种操作是不必要的，还会导致 make gen.errcode.code 执行很慢。

技巧 9：设置 OPTIONS

编写 Makefile 时，我们还需要把一些可变的功能通过 OPTIONS 来控制。为了帮助你理解，这里还是拿 IAM 项目的 Makefile 来举例。

假设我们需要通过一个选项 V，来控制是否需要在执行 Makefile 时打印详细的信息。这可以通过下面的步骤来实现。

首先，在 /Makefile 中定义 USAGE_OPTIONS。定义 USAGE_OPTIONS 可以使开发者在执行 make help 后感知到此 OPTION，并根据需要进行设置。

```
1 define USAGE_OPTIONS
2
3 Options:
4     ...
5     BINS          The binaries to build. Default is all of cmd.
```

[复制代码](#)

```

6         ...
7     ...
8     V          Set to 1 enable verbose build. Default is 0.
9 endef
10 export USAGE_OPTIONS

```

接着，在 [scripts/make-rules/common.mk](#) 文件中，我们通过判断有没有设置 V 选项，来选择不同的行为：

```

1 ifndef V
2 MAKEFLAGS += --no-print-directory
3 endif

```

[复制代码](#)

当然，我们还可以通过下面的方法来使用 V：

```

1 ifeq ($(origin V), undefined)
2 MAKEFLAGS += --no-print-directory
3 endif

```

[复制代码](#)

上面，我介绍了 V OPTION，我们在 Makefile 中通过判断有没有定义 V，来执行不同的操作。其实还有一种 OPTION，这种 OPTION 的值我们在 Makefile 中是直接使用的，例如 BINS。针对这种 OPTION，我们可以通过以下方式使用：

```

1 BINS ?= $(foreach cmd,${COMMANDS},${notdir ${cmd}})
2 ...
3 go.build: go.build.verify $(addprefix go.build., $(addprefix $(PLATFORM)., $(B


```

[复制代码](#)

也就是说，通过 **?=** 来判断 BINS 变量有没有被赋值，如果没有，则赋予等号后的值。接下来，就可以在 Makefile 规则中使用它。

技巧 10：定义环境变量

我们可以在 Makefile 中定义一些环境变量，例如：

 复制代码


```
1 GO := go
2 GO_SUPPORTED_VERSIONS ?= 1.13|1.14|1.15|1.16|1.17
3 GO_LDFLAGS += -X $(VERSION_PACKAGE).GitVersion=$(VERSION) \
4   -X $(VERSION_PACKAGE).GitCommit=$(GIT_COMMIT) \
5   -X $(VERSION_PACKAGE).GitTreeState=$(GIT_TREE_STATE) \
6   -X $(VERSION_PACKAGE).BuildDate=$(shell date -u +%Y-%m-%dT%H:%M:%SZ)
7 ifneq ($(DLV),)
8   GO_BUILD_FLAGS += -gcflags "all=-N -l"
9   LDFLAGS = ""
10 endif
11 GO_BUILD_FLAGS += -tags=jsoniter -ldflags "$(GO_LDFLAGS)"
12 ...
13 FIND := find . ! -path './third_party/*' ! -path './vendor/*'
14 XARGS := xargs --no-run-if-empty
```

这些环境变量和编程中使用宏定义的作用是一样的：只要修改一处，就可以使很多地方同时生效，避免了重复的工作。

通常，我们可以将 GO、GO_BUILD_FLAGS、FIND 这类变量定义为环境变量。

技巧 11：自己调用自己

在编写 Makefile 的过程中，你可能会遇到这样一种情况：A-Target 目标命令中，需要完成操作 B-Action，而操作 B-Action 我们已经通过伪目标 B-Target 实现过。为了达到最大的代码复用度，这时候最好的方式是在 A-Target 的命令中执行 B-Target。方法如下：

 复制代码

```
1 tools.verify.%:
2   @if ! which $* &>/dev/null; then $(MAKE) tools.install.$*; fi
```

这里，我们通过 \$(MAKE) 调用了伪目标 tools.install.\$*。要注意的是，默认情况下，Makefile 在切换目录时会输出以下信息：

 复制代码

```
1 $ make tools.install.codegen
2 =====> Installing codegen
3 make[1]: Entering directory `/home/colin/workspace/golang/src/github.com/marmot
4 make[1]: Leaving directory `/home/colin/workspace/golang/src/github.com/marmot
```


如果觉得 **Entering directory** 这类信息很烦人，可以通过设置 `MAKEFLAGS += --no-print-directory` 来禁止 Makefile 打印这些信息。

总结

如果你想要高效管理项目，使用 Makefile 来管理是目前的最佳实践。我们可以通过下面的几个方法，来编写一个高质量的 Makefile。

首先，你需要熟练掌握 Makefile 的语法。我建议你重点掌握以下语法：Makefile 规则语法、伪目标、变量赋值、特殊变量、自动化变量。

接着，我们需要提前规划 Makefile 要实现的功能。一个大型 Go 项目通常要实现以下功能：代码生成类命令、格式化类命令、静态代码检查、测试类命令、构建类命令、Docker 镜像打包类命令、部署类命令、清理类命令，等等。

然后，我们还需要通过 Makefile 功能分类、文件分层、复杂命令脚本化等方式，来设计一个合理的 Makefile 结构。

最后，我们还需要掌握一些 Makefile 编写技巧，例如：善用通配符、自动变量和函数；编写可扩展的 Makefile；使用带层级的命名方式，等等。通过这些技巧，可以进一步保证我们编写出一个高质量的 Makefile。

课后练习

1. 走读 IAM 项目的 Makefile 实现，看下 IAM 项目是如何通过 `make tools.install` 一键安装所有功能，通过 `make tools.install.xxx` 来指定安装 xxx 工具的。
2. 你编写 Makefile 的时候，还用到过哪些编写技巧呢？欢迎和我分享你的经验，或者你踩过的坑。

期待在留言区看到你的思考和答案，也欢迎和我一起探讨关于 Makefile 的问题，我们下一讲见！

分享给需要的人，Ta 订阅后你可得 **24 元现金奖励**

© 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 | API 风格（下）：RPC API介绍

下一篇 15 | 研发流程实战：IAM项目是如何进行研发流程管理的？

更多课程推荐

容器实战高手课

在实战中深入理解容器技术的本质

李程远

eBay 总监级工程师
云平台架构师



涨价倒计时

今日订阅 **¥69**，7月20日涨价至 **¥129**

精选留言 (4)

写留言



helloworld

2021-06-26

格式化代码、静态代码检查，这种ide或vim都会配置保存时格式化和代码检查，还有必要写在makefile中吗

作者回复: ide/vim不会进行静态代码检查。

ide/vim的格式化是可配的。而且ide/vim的格式化更多的是用了gofmt -w这种格式化。

这里将格式化代码放在Makefile有个选择：

1. 更丰富, 可配置的格式化代码功能, 比如: 支持golines、goimport、gofmt后面还可以根据需要增加更多
2. 确保每一个开发者用的都是同一种格式化方法 (ide/vim的格式化不一定每一个开发者都会陪, 更不一定配置的格式化选项是一致的)

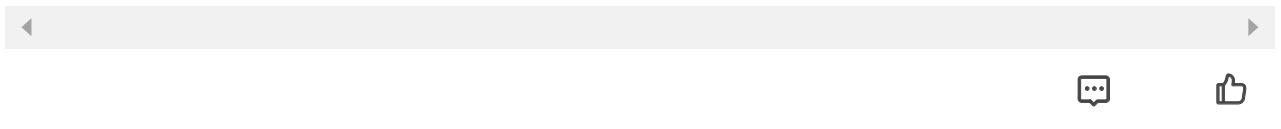
将格式化放在Makefile中, 可以确保某个动作一定会被执行, 并且执行的效果是一致的。



2021-06-28

```
[going@dev iam]$ make install
make[1]: *** No rule to make target 'install.install'. Stop.
make: *** [Makefile:154: install] Error 2
[going@dev iam]$
```

作者回复: 这个命令目前还只是个占位符, 不建议用。底层代码已实现, 但还没测试, 后续会补上

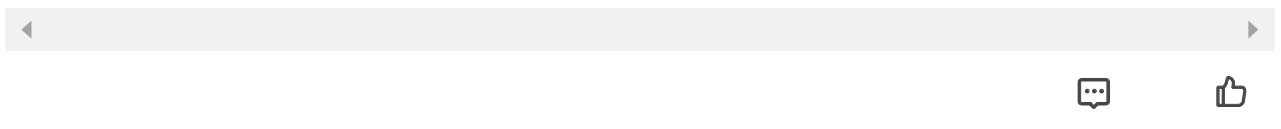


2021-06-26

看iam源码, 各功能的service和store都是每次调用的时候, 返回一个新的实例, 这样对性能有影响吗, 或者是有别的考虑吗?

展开 ▾

作者回复: 返回的是已经初始化好的实例



2021-06-26

写好一个功能齐全的项目的makefile, 然后只对makefile 的各个功能做编排, 是不是就可以做到基本的持续发布了?

作者回复: 持续发布需要CI/CD系统的支持。

写好Makefile只能说方便CI系统直接调用。离CI/C差的还远

