首页

Glide: Go的包版本管理工具

Glide 从概念上来讲是一个 Go 的包管理工具，类似于 Rust 的 Cargo， Node.js 的 NPM，Python 的 Pip，Ruby 的 Boundler 等等…

Glide 提供了以下功能：

- 在 `glide.yaml` 文件中记录依赖信息。包括了名称、版本或版本范围、私有仓库或者类型不能识别时的版本控制信息等等
- 通过 `glide.lock` 文件来追踪每个包的具体修改，这使得能够重用依赖树。
- 适用于语义版本和语义版本范围。
- 支持 Git、Bzr、HG 和 SVN。和 `go get` 支持的版本控制系统一致。
- 利用 `vendor/` 目录使得不同项目可以拥有相同依赖的不同版本。
- 允许使用包别名，这对 `forks` 非常有用。
- 支持从 Godep、GPM、Gom 和 GB 导入配置

安装 Glide

有以下几种方式安装 Glide：

1. 使用脚本自动安装。`curl https://glide.sh/get | sh`
2. 下载发布版本来安装。Glide 发布语义版本。
3. 使用系统包管理工具来安装Glide。例如，如果你在 Mac 上使用 Homebrew，则可以使用 `brew install glide` 来安装。
4. 使用 `go get` 来安装最新的开发快照版本（这不是发布版本）。例如，`go get -u github.com/Masterminds/glide`。

入门指南

入门指南

在你安装了Glide之后，这是一个快速入门指南。

检测项目依赖

Glide可以检测项目的依赖关系，并为你创建一个初始的 `glide.yaml` 文件。此检测可以从 Godep、GPM、Gom和GB中导入配置。进入项目的顶级目录并执行以下命名即可：

```
$ glide init
```

完成之后将会生成一个 `glide.yaml` 文件。你可以打开此文件，甚至编辑它以添加诸如版本之类的信息。

运行 `glide init` 还会询问您是否要使用向导来发现有关依赖关系版本的信息，并使用版本或范围。每个决定都是交互的。

更新依赖

要获取依赖项并将其设置为 `glide.yaml` 文件中指定的任何版本，请使用以下命令：

```
$ glide up
```

`up` 是 `update` 的简写。这将获取 `glide.yaml` 文件中指定的任何依赖项，依次遍历依赖关系树，以确保依赖包的依赖都被获取，并将其设置为适当的版本。遍历依赖关系树时将确定版本，导入 Godep、GPM、Gom 和 GB 中配置的依赖。

获取的依赖包全部放在项目根目录下的 `vendor/` 文件夹中。如果你使用Go 1.6+或Go 1.5，启用了Go 1.5 的 `vendor` 功能，那么 `go` 工具链将在搜索 `GOPATH` 或 `GOROOT` 之前使用这里的依赖包。

Glide 之后会创建一个 `glide.lock` 文件。这个文件包含了特定 `commit id` 的整个依赖关系树。这个文件，我们稍后会看到，可以用来重新创建所使用的确定的依赖关系树和版本。

如果要从依赖关系中删除嵌套的 `vendor/` 目录，请使用 `--strip-vendor` 或 `-v` 参数。

依赖扁平化

Glide 获取的所有依赖都放置在项目顶级目录下的 `vendor/` 中。Go 支持每个包都有一个 `vendor/` 目录，但 Glide 仅使用顶级文件夹有两个原因：

1. 每个导入都会被编译成二进制文件。如果同一个依赖包出现在三个`vendor/`目录中，那么它会被编译三次。这样会迅速导致二进制文件体积变大。
2. 一个`vendor/`目录中的依赖包创建的实例与其他位置的依赖包不兼容，即便它们是相同的版本。Go 将它们视作不同的包，因为它们位于不同的位置。这对 数据库驱动、日志记录器 和 其他很多事情都是个问题。如果你 尝试将一个个位置的包创建的实例传递给另一个位置的包将发生错误

如果一个依赖包有它自己的 `vendor/` 目录，Glide 默认情况下不会去删除它。 `go` 工具链的做法是如果这些嵌套的依赖包存在则会使用它。要删除它们，请在 `up` 或 `install` 命令上使用 `--strip-vendor` 或者 `-v` 参数。

安装依赖

如果你想安装一个项目所需要的依赖包，请使用 `install` 命令：

```
$ glide install
```

```
```

这个命令做了以下两者之一：

- 如果一个glide.lock文件存在，它将检索（`vendor/`目录中缺失的）依赖包，并将其设置为`glide.lock`文件中的确切版本。获取依赖包设置版本是并发的，因此操作相当快。
- 如果没有`glide.lock`，那么`update`将会被执行。

如果你不管理项目的依赖包的版本，但是需要安装依赖包，则应使用 `install` 命令。

添加更多依赖

Glide可以帮助您使用 `get` 命令为 `glide.yaml` 文件添加更多的依赖项。

```
1. $ glide get github.com/Masterminds/semver
```

`get` 命令类似于 `go get`，将依赖包下载到 `vendor/` 目录中，并将它们添加到 `glide.yaml` 文件中。此命令可以获取一个或者多个依赖包。

`get` 命令也可以使用版本。

```
1. $ glide get github.com/Masterminds/semver#~1.2.0
```

`#` 用作依赖包和要使用的版本之间的分隔符。版本可以是一个语义版本、版本范围、 `branch` 、 `tag` 、或者 `commit id` 。

如果没有指定版本或范围，并且依赖包使用语义版本，那么Glide将询问你是否要使用它们。

配置文件
glide.yaml 文件

`glide.yaml` 文件包含了项目和依赖包的信息。这里列出了 `glide.yaml` 文件中的元素。

```
package: github.com/Masterminds/glide
homepage: https://masterminds.github.io/glide
license: MIT
owners:
- name: Matt Butcher
  email: technosophos@gmail.com
  homepage: http://technosophos.com
- name: Matt Farina
  email: matt@mattfarina.com
  homepage: https://www.mattfarina.com
ignore:
- appengine
excludeDirs:
- node_modules
import:
- package: gopkg.in/yaml.v2
- package: github.com/Masterminds/vcs
  version: ^1.2.0
  repo:    git@github.com:Masterminds/vcs
  vcs:     git
- package: github.com/codegangsta/cli
  version: f89effe81c1ece9c5b0fda359ebd9cf65f169a51
- package: github.com/Masterminds/semver
  version: ^1.0.0
testImport:
- package: github.com/arschles/assert
```

这些元素是：

- `package`：顶级包是`GOPATH`中的位置。这用于确保不会导入顶级包的内容。
- `homepage`：可以找到有关包或应用程序的详细信息的地方。例如http://k8s.io
- `license`：许可证是 SPDX 许可证字符串或许可证的文件路径。这允许自动化和用户轻松识别许可证。
- `owners`：一个或者多个项目所有者列表。可以是个人或者组织，这对反馈安全问题给所有者很有用。
- `ignore`：需要Glide忽略导入的依赖包列表。它们是包名称而不是包目录。
- `excludeDirs`：本地代码库中的目录列表，用以从扫描依赖关系中排除。
- `import`: A list of packages to import. Each package can include:
- `import`：导入的依赖包列表。每个依赖包可以包含：
  - `package`：需要导入的包名称，这是唯一的必选项。包名称遵循`go`工具一样的模式。这意味着：
    - 映射到`VCS`远程位置的包名称，以 .git ，.bzr ，.hg 或者 .svn。例如，`example.com/foo/pkg.git/subpkg`。
    - GitHub, BitBucket, Launchpad, IBM Bluemix Services, and Go on Google Source are special cases that don't need the VCS extension.
    - GitHub，BitBucket，Launchpad，IBM Bluemix Services以及Google Source是不需要VCS扩展的特殊情况。
  - `version`：使用的语义版本、语义版本范围，`branch`，`tag`，或者 `commint id`。更多信息请看版本文档。
  - `repo`：如果包名称不是一个仓库地址，或者是一个私有仓库，可以列在这里。包将从仓库检出，并存放到包名称指定的地方。这允许使用`forks`。
  - `vcs`：版本控制系统，例如 git，hg，bzr 或者 svn。这仅在不能从包名称推测出类型的时候需要。例如,一个以 .git 结尾或者 GitHub 的包可以被检测为 Git。对于 Bitbucket 的仓库，我盟可以调用 API 来检测类型。
  - `subpackages`：在仓库内部使用的包的记录。这并不包括仓库中的所有包，而是直包括正在使用的包。
  - `os`：用于过滤的操作系统列表。设置后它将比较当前运行的操作系统是否和其中一个匹配，只有在匹配的时候才会获取依赖包。不设置则会跳过过滤。名称和 构建参数、GOOS 环境变量中使用的名称相同。
  - `arch`：用户过滤的系统架构列表。设置后它将比较当前运行的系统架构是否和其中一个匹配，只有在匹配的时候才会获取依赖包。不设置则会跳过过滤。名称和 构建参数、GOARCH 环境变量中使用的名称相同。
- `testImport`：在前面`import`中未列出的测试中使用的包列表。每个包具有与`import`下列出的相同的细节。

Example

```
# The name of this package.
package: github.com/Masterminds/glide

# External dependencies.
import:
  # Minimal definition
  # This will use "go get [-u]" to fetch and update the package, and it will
  # attempt to keep the release at the tip of master. It does this by looking
  # for telltale signs that this is a git, bzr, or hg repo, and then acting
  # accordingly.
  - package: github.com/kylelemons/go-gypsy

  # Full definition
  # This will check out the given Git repo, set the version to master,
  # use "git" (not "go get") to manage it, and alias the package to the
  # import path github.com/Masterminds/cookoo
  - package: github.com/Masterminds/cookoo
    vcs: git
    version: master
    repo: git@github.com:Masterminds/cookoo.git

  # Here's an example with a commit hash for a version. Since repo is not
  # specified, this will use git to to try to clone
  # 'http://github.com/aokoli/goutils' and then set the revision to the given
  # hash.
  - package: github.com/aokoli/goutils
    vcs: git
    version: 9c37978a95bd5c709a15883b6242714ea6709e64

  # MASKING: This takes my fork of goamz (technosophos/goamz) and clones it
  # as if it were the crowdmob/goamz package. This is incredibly useful for
  # masking packages and/or working with forks or clones.
  #
  # Note that absolutely no namespace munging happens on the code. If you want
  # that, you'll have to do it on your own. The intent of this masking was to
  # make it so you don't have to vendor imports.
  - package: github.com/crowdmob/goamz
    vcs: git
    repo: git@github.com:technosophos/goamz.git

  - package: bzr.example.com/foo/bar/trunk
    vcs: bzr
    repo: bzr://bzr.example.com/foo/bar/trunk
    # The version can be a branch, tag, commit id, or a semantic version
    # constraint parsable by https://github.com/Masterminds/semver
    version: 1.0.0

  - package: hg.example.com/foo/bar
    vcs: hg
    repo: http://hg.example.com/foo/bar
    version: ae081cd1d6cc

  # For SVN, the only valid version is a commit number. Tags and branches go in
  # the repo URL.
  - package: svn.example.com/foo/bar/trunk
    vcs: svn
    repo: http://svn.example.com/foo/bar/trunk


  # If a package is dependent on OS, you can tell Glide to only
  # fetch for certain OS or architectures.
  #
  # os can be any valid GOOS.
  # arch can be any valid GOARCH.
  - package: github.com/unixy/package
```

```
    os:
      - linux
      - darwin
    arch:
      - amd64
```

版本和范围

Glide 支持 语义版本， `SemVer` 范围， `branch` ， `tag` 和 `commit id` 作为版本。

基本范围

一个简单的范围是 `> 1.2.3` 。这告诉Glide使用 `1.2.3` 之后的最新版本。Glide 支持一下操作符：

- `=` ：相等（相当于没有使用操作符）
- `!=` ： 不等
- `>` ： 大于
- `<` ： 小于
- `>=` ： 大于等于
- `<=` ：小于等于

它们可以组合。 `,` 是 `and` 关系， `||` 是 `or` 关系。 `or` 关系会分组进行检查。例如， `">= 1.2, < 3.0.0 || >= 4.2.3"` 。

连字符范围

有多个快捷方式来处理范围，第一个是连字符范围。这些看起来像：

- `1.2 - 1.4.5` 相当与 `>= 1.2, <= 1.4.5`
- `2.3.4 - 4.5` 相当于 `>= 2.3.4, <= 4.5`

通配符比较

`x` ， `X` 和 `*` 字符可以被用作通配符。这适用于所有比较运算符。当在 `=` 操作符上使用时，它将进行补丁级别的比较（参见下面的波浪号）。例如：

- `1.2.x` 等价于 `>= 1.2.0, < 1.3.0`
- `>= 1.2.x` 等价于 `>= 1.2.0`
- `<= 2.x` 等价于 `< 3`
- `*` 等价于 `>= 0.0.0`

波浪号范围比较(补丁)

波形符号（ ）运算符用于补丁级别范围比较，当指定次要版本（ `minor` ）时，版本在次要版本号之间变动。当次要版本缺失时，版本在主要版本（ `major` ）号之间变动。例如：

- `~1.2.3` 相当于 `>= 1.2.3, < 1.3.0`
- `~1` 相当于 `>= 1, < 2`
- `~2.3` 相当于 `>= 2.3, < 2.4`
- `~1.2.x` 相当于 `>= 1.2.0, < 1.3.0`
- `~1.x` 相当于 `>= 1, < 2`

插入符号范围比较(主版本)

插入符号（ `^` ）比较运算符用于主要版本的更改。这是比较有用的API版本作为一个主要的变化是API 破坏。例如

- `^1.2.3` 相当于 `>= 1.2.3, < 2.0.0`
- `^1.2.x` 相当于 `>= 1.2.0, < 2.0.0`
- `^2.3` 相当于 `>= 2.3, < 3`
- `^2.x` 相当于 `>= 2.0.0, < 3`

锁文件(Lock file)

glide.lock 文件

`glide.yaml` 文件包含本地代码库的依赖关系，版本（包括范围）和其他配置，相关的glide.lock文件包含完整的依赖关系树和正在使用的修订（commit id）。

了解完整的依赖关系树对于Glide是有用的。例如，当完整树已知时，`glide install` 命令可以同时安装并设置多个依赖关系的正确版本。这是可重复安装依赖包的快速操作。

锁文件还提供了完整树的记录，超出了你的代码库所需要以及所使用的版本。在排除问题时，这对于出了问题后审计或检测依赖关系树中发生了什么变化的事情非常有用。

此文件的详细信息不包括在此，因为此文件不应手动编辑。如果您知道如何阅读 `glide.yaml` 文件，您将能够大体了解 `glide.lock` 文件。

命令

命令

以下是 Glide 的命令，大部分是帮助管理工作区的。

glide create（别名：glide init）

初始化新的工作区。除此之外，它会在尝试猜测要放入的包和版本时创建一个 `glide.yaml` 文件。例如，如果你的项目使用 Godep，它将其使用指定的版本。Glide 足够聪明地扫描代码库并检测正在使用的导入，无论它们是否由另一个程序包管理器指定。

```
$ glide create
[INFO]  Generating a YAML configuration file and guessing the dependencies
[INFO]  Attempting to import from other package managers (use --skip-import to skip)
[INFO]  Scanning code to look for dependencies
[INFO]  --> Found reference to github.com/Masterminds/semver
[INFO]  --> Found reference to github.com/Masterminds/vcs
[INFO]  --> Found reference to github.com/codegangsta/cli
[INFO]  --> Found reference to gopkg.in/yaml.v2
[INFO]  Writing configuration file (glide.yaml)
[INFO]  Would you like Glide to help you find ways to improve your glide.yaml configuration?
[INFO]  If you want to revisit this step you can use the config-wizard command at any time.
[INFO]  Yes (Y) or No (N)?
n
[INFO]  You can now edit the glide.yaml file. Consider:
[INFO]  --> Using versions and ranges. See https://glide.sh/docs/versions/
[INFO]  --> Adding additional metadata. See https://glide.sh/docs/glide.yaml/
[INFO]  --> Running the config-wizard command to improve the versions in your configuration
```

这里提到的配置向导（`config-wizard`）可以在这里运行，也可以在以后手动运行。此向导可帮助找出可用于依赖关系的版本和范围

glide config-wizard

这将运行一个向导，扫描依赖关系并检索其上的信息，以提供可交互选择的建议。例如，它可以发现依赖关系是否使用语义版本，并帮助你选择要使用的版本范围。

glide get [package name]

你可以将一个或多个软件包下载到你的 `vendor/` 目录，并将其添加到你的 `glide.yaml` 文件中。

```
$ glide get github.com/Masterminds/cookoo
```

当使用 `glide get` 时，它会检测列出的包来解决它的依赖关系，包括使用 Godep，GPM，Gom 和GB 的配置文件。

`glide get` 命令可以给包名传递一个版本或范围。例如，

```
$ glide get github.com/Masterminds/cookoo#^1.2.3
```

包名和版本通过一个 `#` 来分隔。如果没有指定版本或范围，并且依赖关系使用语义版本，那么 Glide 将询问是否要使用它们。

glide update（别名： glide up）

下载或更新 `glide.yaml` 文件中列出的所有库，并将它们放在 `vendor/` 目录中。它还将递归地遍历依赖包以获取所需的任何配置文件。

```
```

```
$ glide up
```

这将会覆盖由Glide，Godep，gb，gom和GPM管理的其他项目。当找到这些包将根据需要安装。

将使用固定到特定版本的依赖关系创建或更新 `glide.lock` 文件。例如，如果在 `glide.yaml` 文件中将版本指定为范围（例如，`^1.2.3`），则 `glide.lock` 文件中将其设置为特定的 `commit id` 。这允许可重复的安装（参见 `glide install` ）。

从获取的包中移除任何嵌套的 `vendor/` 目录请看 `-v` 参数。

glide install

当你想从 `glide.lock` 文件安装特定版本的包时使用 `glide install` 。

```
$ glide install
```

这将会读取 `glide.lock` 文件，如果它没有绑定到 `glide.yaml` 文件则会发出警告，并安装特定 `commit id` 的版本。

当 `glide.lock` 文件没有绑定到 `glide.yaml` 文件，例如有更改，它将提示警告。运行 `glide up` 将在更新依赖关系树时重新创建 `glide.lock` 文件。

如果没有 `glide.lock` 文件， `glide install` 将执行 `update` 并生成一个锁文件。

从获取的包中移除任何嵌套的 `vendor/` 目录请看 `-v` 参数。

glide novendor（别名： glide nv）

当您运行 `go test ./..` 命令时，它将遍历包括 `vendor/` 目录在内的所有子目录。当测试应用程序时，你可能只需要测试应用程序文件，而无需运行依赖包及其依赖包的所有测试。这时 `novendor` 命令派上了用场，它列出了除 `vendor/` 之外的所有目录。

```
$ go test $(glide novendor)
```

这将会在你项目除 `vendor` 外所有目录中运行运行 `go test` 。

glide name

当您使用 Glide 进行脚本编写时，您需要知道您正在处理的包的名称。 `glide name` 返回 `glide.yaml` 文件中列出的的名称。

glide list

Glide的 `list` 命令按字母顺序显示项目导入的所有包。

```
$ glide list
INSTALLED packages:
   vendor/github.com/Masterminds/cookoo
   vendor/github.com/Masterminds/cookoo/fmt
   vendor/github.com/Masterminds/cookoo/io
   vendor/github.com/Masterminds/cookoo/web
   vendor/github.com/Masterminds/semver
   vendor/github.com/Masterminds/vcs
   vendor/github.com/codegangsta/cli
   vendor/gopkg.in/yaml.v2
```

glide help

打印 glide 帮助信息。

```
$ glide help
```

glide —version

打印 glide 版本并退出。

```
$ glide --version
glide version 0.12.0
```

glide mirror

镜像提供了将一个仓库替换为其镜像仓库的能力。当您希望为连续集成（CI）系统设置缓存或者想要在本地依赖包上工作时，这很有用。

镜像存放在你 `GLIDE_HOME` 中的一个 `mirrors.yaml` 文件中。

The three commands to manage mirrors are `list`, `set`, and `remove`.
管理镜像有三个命令： `list` 、 `set` 和 `remove` 。

使用 `set` ：

```
glide mirror set [original] [replacement]
```

或者：

```
glide mirror set [original] [replacement] --vcs [type]
```

例如，

```
glide mirror set https://github.com/example/foo https://git.example.com/example/foo.git
```

或

```
glide mirror set https://github.com/example/foo file:///path/to/local/repo --vcs git
```

使用 `remove` ：

```
glide mirror remove [original]
```

例如，

```
glide mirror remove https://github.com/example/foo
```

导入解析

导入解析

Glide 扫描应用程序代码库以发现在 `vendor/` 目录中管理的项目。这以几种不同的方式发生。知道如何运作可以帮助您了解Glide正在做什么

初始化时

当你为代码库运行 `glide create` 或者 `glide init` 来穿件一个 `glide.yaml` 时，Glide 将会扫描代码库来识别导入。它通过文件来识别包。在每个包中，它读取Go文件中的导入。

从这里，它将尝试找出外部包。外部包按根版本控制系统仓库与其下列列出的子包来分组。遵循与 go 工具一样的规则来找出根版本控制仓库与其下的包相比。

1. GitHub, Bitbucket, Launchpad, IBM Jazz 和 go.googlesource.com 会用特殊规则解析。 我们可以知道或可以调用API来了解这些包。
2. 如果与仓库相关联的包以.git，.hg，.bzr或.svn结尾，则可以用于确定版本控制系统的根和类型。
3. 如果规则无法解析，则会尝试通过 `go get` 来查找信息。

再次，和使用 `go get` 确定外部包位置时的方式是一样的。

如果项目具有存储在Godep，GPM，Gom或GB文件中的依赖包配置，则该信息将用于填充 `glide.yaml` 文件中的版本

更新时

当运行 `glide update` ， `glide up` ， `glide get` 或者 `glide install` （没有 `glide.lock` 时），Glide 会尝试检测完整的依赖树。那是所有的依赖关系，包含了依赖的依赖。

默认选项

默认方式是遍历导入引用树。解析器首先扫描本地应用程序以获取导入列表。然后，它查看特定的包导入，扫描导入的包进行导入，并不断查找，直到获取完整的树。

这意味者仅源码中引用的导入才会获取。

当获取版本控制仓库时，它将获取完整的仓库。但是，它不扫描仓库中的所有包来获取依赖关系。相反，只有被扫描时在树中引用的包才会继续导入。

按照 Glide, Godep, GPM, Gom 和 GB 配置方式来确定版本和要获取的仓库。先进入引导树的版本优先。

所有可能的依赖

在 `glide update` 使用 `--all-dependencies` 将会改变扫描的行为。它会去文件系统中获取所有可能的引用的包，而不是遍历导入树。这将下载树中的所有包。即使那些在程序代码中或代码导入的包并未引用到的。

与其他情况一样，Glide，Godep，GPM，Gom 和 GB 文件用于设置所获取仓库的版本。

Importing

Glide has limited support for importing from other formats.

Note: If you'd like to help build importers, we'd love some pull

requests. Just take a look at `cmd/godeps.git`.

Godeps and Godeps-Git

To import from Godeps or Godeps-Git format, run `glide godeps`. This will read the `glide.yaml`, then look for `Godeps` or `Godeps-Git` files to also read. It will then attempt to merge the packages in those files into the current YAML, printing the resulting YAML to standard out.

The preferred procedure for merging:

```
$ glide godeps # look at the output and see if it's okay
$ glide -q godeps > glide.yaml # Write the merged file
```

Vendor 目录

Vendor 目录

随着 Go 1.5 的发布， `vendor/` 目录除了 `GOPATH` 和 `GOROOT` 之外的依赖包的解析位置。Go 1.6 之前这是一个需要设置环境变量 `GO15VENDOREXPERIMENT=1` 才能启用的功能，Go 1.6 中是一个默认启用的功能。

_注意，即使你使用了 `vendor/` ，你的代码库还是需要放置在 `GOPATH` 之中，这对 `Go` 工具链来说是无法避免的。

一个依赖包的解析位置是：

- 当前包中的`vendor/`目录。
- 目录树中查找父级中的`vendor/`目录。
- 查找`GOPATH`中的包。
- 如果存在于`GOROOT`(标准库所在的位置)中，则使用`GOROOT`中的包。

建议

自从它们首次发布依赖，使用 `vendor/` 目录过程中，我们得出了一些结论和建议。Glide 试图在以下几方面帮助你：

1. 库（不带`main`包的代码库）不应将外部包存储在其 VCS 中的`vendor/`中，除非它们具有特定的原因并且了解为什么要执行此操作。
2. 在应用程序（带有`main`包的代码库）中，代码库顶层只应该有一个`vendor/`目录。

这些建议有一些重要的原因。

- 包的每个实例，即使是同一个版本的包，在目录结构中将会在编译进最终的二进制文件中。如果每个包分别存储自己的依赖包，这将很快导致二进制文件变大。
- 不同位置的相同的包创建的实例**不兼容**，那么它们是相同的版本。你可以自己查看。这意味着日志记录器、数据库连接和其他共享的实例无法正常工作。

因此 Glide 将依赖关系平铺到单一的顶层 `vendor/` 目录中。如果一些包刚好在它自己的 `vendor/` 中有一些依赖包， `go` 工具会正确的解析该版本。

为何使用一个`vendor` 目录?

如果我们已经有了 `GOPATH` 来存储包，那么为什么需要一个 `vendor/` 目录?这完全是一个正常的问题。

如果 `GOPATH` 中的多个应用程序使用同一个包的不同版本怎么办？这是一个在 `Go` 应用程序中遇到的正常问题，并且已经存在了很长时间。

`vendor/` 目录允许不同的代码库具有自己的版本，而不会受到另一个需要不同版本的代码库的干扰。 它为每个项目提供了一个隔离级别。

插件

Glide 插件

Glide 支持一个类似于Git的简单的插件系统。

已有的插件

Glide 已经包含一些插件：

- glide-vc - 构建应用时允许你跳过 `vendor/` 目录中你不需要的文件。
- glide-brew - 帮助你转换通过Go deps管理的项目到 `Homebrew`，便于为你的 Go 程序构建 brew 版本。
- glide-hash - 生成与Glides内部哈希兼容的 `glide.yaml` 文件的哈希值。
- glide-cleanup - 从 `glide.yaml` 文件中移除没有使用的包。
- glide-pin - 从 `glide.lock` 文件中取出所有的依赖包，并把它们和 `glide.yaml` 中正确的对应起来。

注意，往这个列表添加插件请提交一个 `pull request` 。

插件如何工作

当 Glide 遇到一个未知的子命令时，会根据以下规则尝试将其委托给另一个可执行文件。

例子：

```
$ glide install # 已知的命令，所以直接执行它
$ glide foo     # 未知的命令，所以会寻找一个合适的插件
```

In the example above, when glide receives the command `foo`, which it does not know, it will do the following:
上面的例子中，当 glide 接收到不认识的命令 `foo`，它将执行以下操作：

1. 将 `foo` 变为 `glide-foo`
2. 在系统的 `$PATH` 中寻找 `glide-foo`，如果找到则执行它。
3. 否则在当前项目的根目录下寻找 `glide-foo`。(也就是说，在与 `glide.yaml` 相同的目录中)。如果找到则执行它。
4. 如果没有找到适当的命令，则错误退出。

编写 Glide 插件

Glide 插件可以用任何你想要的语言编写，只要它可以作为Glide的子进程从命令行执行。Glide包含的示例是一个简单的Bash脚本。我们可以很容易地用 Go，Python，Perl，甚至是Java（带包装器）来编写。

Glide插件必须位于两个位置之一：

1. `PATH` 路径中
2. 和 `glide.yaml` 相同的目录

建议系统级别的插件放在 `/usr/local/bin` 或者 `$GOPATH/bin` 中，而项目特定的插件放在和 `glide.yaml` 所在的目录里。

参数和标志

说 Glide 是这样执行的：

```
$ glide foo -name=Matt myfile.txt
```

Glide会将此解释为使用 `--name=Matt myfile.txt` 参数来执行 `glide-foo` 的请求。它不会试图解析这些参数或以任何方式修改它们

Glide会将此解释为使用 `-name = Matt myfile.txt` 参数来执行 `glide-foo` 的请求。它不会试图解析这些参数或以任何方式修改它们

假设，如果 Glide 有自己的 `-x` 标志，你可以这样调用：

```
$ glide -x foo -name=Matt myfile.txt
```

这种情况下，glide 将会解释并私吞 `-x` ，并将其余部分传递给 `glide-foo` ，如上例所示。

插件例子

文件： glide-foo

```
#!/bin/bash

echo "Hello"
```

```
#!/bin/bash

# You can execute me through Glide by doing the following:
# - Copy me to a directory on $PATH. _vendor/bin/ will work just fine.
# - Execute `glide plugin-example`
# - ???
# - Profit

echo "I received arguments '$@'"

# This should match the directory from which you executed glide.
echo "My current working directory is $(pwd)"

# This is the GOPATH for the current glide session.
echo "My GOPATH is $GOPATH"

# This is the base directory of your project.
echo "The project directory is $GLIDE_PROJECT"

# This is the location of the glide.yaml file.
echo "The Glide YAML is in $GLIDE_YAML"

# This is the PATH that the plugin inherited.
echo "My PATH is $PATH"
```

常见问题

常见问题

Q: 为什么 Glide 有子包的概念而 Go 没有?

Go 中每个目录都是一个包。这在你有一个包含了所有包的仓库时工作正常。当你有不同的包在不同的 VCS 时事情就有一点复杂了。包含包的集合的项目应使用与版本相同的信息进行处理。通过以这种方式对包进行分组，我们可以管理相关信息。

Q: bzr（或者 hg）没有按照我的预期工作。为什么？

这些是正在进行的工程，可能需要一些额外的调整。请看看vcs包。如果您看到更好的处理方式，请告诉我们。

Q: 我应该让`vendor/`进版本控制么？

这看你。这是个人或组织的决定。Glide 将帮助你根据需要安装外部依赖包，或帮你管理已进入你的版本控制系统的依赖包。

By default, commands such as `glide update` and `glide install` install on-demand. To manage a vendor folder that's checked into version control use the flags:
默认情况下，`glide update` 和 `glide install` 等命令将按需安装。要管理进入到版本控制的 `vendor/` 文件夹，请使用标志：

- `--update-vendored`（别名：`-u`）更新 `vendor` 依赖.
- `--strip-vcs`（别名：`-s`）从 `vendor` 中删除 VCS 元数据（例如 `.git` 目录）。
- `--strip-vendor`（别名：`-v`）删除嵌套的`vendor/`目录。

Q: 怎么从 GPM, Godep, Gom, or GB 导入配置？

导入有两步。

1. 如果你导入的软件包具有 GPM ，Godep，Gom 或 GB 配置，Glide将自动递归地安装依赖包。
2. 如果要从 GPM ，Godep，Gom 或 GB 导入配置到 Glide，请参阅`glide import`命令。例如，您可以运行Glide的`glide import godep`来检测项目`Godep`配置并为您生成`glide.yaml`文件。

它们中的每一个都会将您现有的 `glide.yaml` 文件与它为这些管理器找到的依赖关系进行合并，然后将该文件输出出来。**它不会覆盖你的 `glide.yaml` 文件。**

你可以这样将它写入文件：

```
$ glide import godep -f glide.yaml
```

Q: Glide 能够按照操作系统或架构来获取一个包么？

可以的。使用 `os` 和 `arch` 字段，可以指定应该为哪个操作系统和体系架构提取包。例如，以下软件包将只能获取64位的 Darwin / OSX 系统：

```
- package: some/package
  os:
    - darwin
  arch:
    - amd64
```

其他体系架构或操作系统的将不会被获取。

Q: Glide 是如何得名的?

除了好读，”glide” 是”Go Elide”的缩写。这个想法是将通常需要大量时间的任务压缩到短短的几秒钟内。