



从零开始制作 deb 文件

为 go-faker 命令行工具制作 deb 分发包 ...

🕒 9 分钟阅读

🏠 > Packaging > Deb > 从零开始制作 deb 文件



hedzr

#modern-cxx #golang #rust #kotlin
#dotnetcore #devops

📍 Chongqing, CN

✉ Email

🌐 Website

🐦 Twitter

🏠 GitHub

OPEN SOURCE PROJECTS

📖 CMDR Docs

📁 Tmpl Repo

META

© License

📄 Terms & Privacy Policy

1. 前言

在这篇文章里，我们会从零开始创建一个 deb 文件。

我们已经有一个基于 [cmdr](#) 开发的命令行工具软件 [faker](#)，我们也在 [go-faker: 一个 mock 数据生成器](#) 中粗略地介绍过这个 CLI 工具有何用途。它很简单，也没有什么高深的技巧，仅仅是因为有时候做什么测试需要有一个随手可得的数据源，这就是理由了。

在 macOS 中我已经提供了 faker 的 Homebrew 安装方式，那可以工作。对于 Windows 用户来说，他们可以下载 release 页面中预先构建的二进制 exe 解包后就能使用。或者也可以通过 `docker pull hedzr/faker` 的方式取用。

为了能在 brew 分发时做自动更新，我还专门做了一个 [go-bumper](#)。原因是 `brew bump-formula-pr` 只能做单个 url 及其 sha256sum 的修订，却不能支持多组 url+sha256sum，而其源码逻辑又很难对此问题做出解决方法，所以只好自己在 release 时自己修改。

对于 Linux 用户尤其是 Debian 系用户来说，今天，我们就准备构建出 .deb 安装包，以便 Debian/Ubuntu 用户能够 wget deb 包之后进行安装。

■ 在本页上

前言

背景知识

Binary package 和 Source Package

由零开始

准备打包内容

bin/ 和 etc/

completions/

manpages

准备控制文件

DEBIAN/control

changelog, compat, copyright

维护脚本

postinst

postrm

preinst 和 prerm

preinst

prerm

维护脚本与安装行为间的关系

Install

Removal

Purging

More

文件布局

生成 deb

2. 背景知识

deb 文件是一个 Unix [ar Archive](#) 格式的文件，在 Linux 环境中，使用 [mc](#) 能够直接查看 deb 所打包的具体内容。以 ar 的视角来说，deb 包含这样：

```
1 | $ ar t faker_0.2-5_amd64.deb
2 |   debian-binary
3 |   control.tar.xz
4 |   data.tar.xz
5 | # Or: ar tv faker_0.2-5_amd64.deb
```

debian-binary 是一个文本文件，包含一个版本号戳记，这代表着 deb 文件格式的版本号，它目前通常都是 2.0。

control.tar.xz 是 DEBIAN 文件夹中的内容，这在稍后会有相应的阐述。简单地说，这里面包含了 deb 包的描述信息，可选的安装、卸载脚本，等等控制文件。

data.tar.xz 是软件包的实际内容，它是一个微型的 Linux 文件系统的 mimic，这个文件系统镜像在安装时会被解包后原样填充到目标系统中。

很久很久以前，Linux 默认使用 gzip 作为压缩选择，所以那时候你会得到 control.tar.gz 这样的文件，但现在都是使用 xz 压缩格式了，它没有额外耗费太多算力的情况下压缩比率有大幅度的提高。gzip 虽然好，但毕竟是行程码风格的压缩算法，即使是对文本文件其压缩比也有限。

进一步地，你还可以使用 tar 命令来展开上面的 tar.xz 文件，例如：

```
1 | tar -xf control.tar.xz -C DEBIAN/
2 | tar -xf data.tar.xz
```

2.1. Binary package 和 Source Package

关于 deb 打包，一直以来有两种：Binary package 和 Source Package。

Binary Package 是针对一个二进制可知性文件（或者可执行脚本等等）进行 deb 构建的简称。这种方式中，直接使用 dpkg-deb –build 的方式将源目录打包为 deb，要求源目录中必须有 DEBIAN/control 文件。

Source Package 相对较为庞大和复杂，一般是借助于 dh-make 软件包的

Linux 账户
面对 deb
调试和除错
外部依赖
后记
参考

工具, 从源代码(往往是 C++ 源代码, 但并不限于此)开始经过源码构建, 控制信息组织和自动生成, deb 构建等一系列流程。此时你需要在源目录下建立 debian/control 文件的基本骨架, 然后你需要提供和放置源代码的 tar.gz tarball 包(或者 tar.xz tarball 包), 然后通过 debuild 方式完成构建。此构建过程中会借助 debhelper 等插件来支持不同源码的编译构建。

Source Package 方式要求小写的 debian 控制文件夹, 完整的文档在 [Debian 新维护者手册](#)。它和本文的内容与目标不相同。

3. 由零开始

我们选择的是作为 binary package 进行打包, 这需要的依赖非常少, 你只需要安装 dpkg 软件包:

```
1 | sudo apt install -y dpkg
```

有的时候, 你会发现 debhelper, debmaker 等提供的辅助工具可能会是有用的, 所以你可以安装更多的东西:

```
1 | sudo apt install -y dpkg build-essential devscripts quilt dh-make git
```

但对于本文来说, dpkg 就够了。

由于依赖如此的少, 所以 binary package 的 deb 构建甚至可以在 macOS 中交叉完成。你需要通过 homebrew 安装 dpkg:

```
1 | brew install dpkg
```

这个过程本文就不展开了。

本文所涉及到的流程, 均通过 ubuntu server VM 来完成。你可以使用 Debian 系的任何发行版完成本文讲述的内容。你也可以在其它发行版中完成, 方法大同小异, 只是安装软件包的不同。

3.1. 准备打包内容

现在我们要准备要打包的内容。这里的内容都在 Ubuntu 上予以完成。

首先我们建立一串文件夹:

```
1 | mkdir faker.work
2 | cd faker.work
```

```
3 |  
4 | mkdir -pv faker_0.2-5_amd64/DEBIAN  
5 | mkdir -pv faker_0.2-5_amd64/usr/lib/faker/{bin,completions,etc,man}
```

以下的操作我们都会基于 `faker.work` 工作目录来进行。

3.1.1. bin/ 和 etc/

首先一件事是让 `faker` 可执行文件就位：

```
1 | wget https://github.com/hedzr/go-faker/releases/download/v0.2.5/faker-linux-amd64.tg  
2 | tar -xf faker-linux-amd64.tgz -C faker_0.2-5_amd64/usr/lib/faker/
```

它确立了 `faker_0.2-5_amd64/usr/lib/faker/bin/faker` 这个可执行文件，以及 `faker_0.2-5_amd64/usr/lib/faker/etc` 这个配置文件夹。

我们还要利用 `faker` 来生成一些附加文件。

3.1.2. completions/

我们为 Shell 自动完成预先生成相应的完成脚本文件，这对于基于 [cmdr](#) 开发的 `golang` CLI app 来说很容易：

```
1 | $ mkdir -pv faker_0.2-5_amd64/usr/lib/faker/completions/{zsh,bash}  
2 | $ ./faker_0.2-5_amd64/usr/lib/faker/bin/faker gen sh --zsh \  
3 | > faker_0.2-5_amd64/usr/lib/faker/completions/zsh/_faker  
4 | $ ./faker_0.2-5_amd64/usr/lib/faker/bin/faker gen sh --bash \  
5 | > faker_0.2-5_amd64/usr/lib/faker/completions/bash/faker
```

3.1.3. manpages

也要生成 Linux 手册页：

```
1 | $ ./faker_0.2-5_amd64/usr/lib/faker/bin/faker gen man \  
2 | -d faker_0.2-5_amd64/usr/lib/faker/man/man1/  
3 | $ for f in faker_0.2-5_amd64/usr/lib/faker/man/man1/*.1; do gzip -f $f; done
```

3.2. 准备控制文件

对于 `binary package` 来说，我们还需要 `DEBIAN/control` 文件。这是最小需求，不过实际上往往会需要稍微多一点。

按照 deb 文件格式规范, DEBIAN 文件夹中包含一系列控制文件。这当中最重要的一个是 control 文件。

3.2.1. DEBIAN/control

我们首先准备一份 control 文件:

```
1 $ cat > ./faker_0.2-5_amd64/DEBIAN/control <<EOF
2 Package: faker
3 Version: 0.2.5
4 Section: utils
5 Priority: extra
6 Architecture: amd64
7 Multi-Arch: foreign
8 Build-Depends: debhelper (>=9)
9 Standards-Version: 3.9.6
10 Maintainer: Hedzr Yeh <hedzrz@gmail.com>
11 Homepage: https://github.com/hedzr/go-faker
12 Description: faked records generator.
13     faker is a CLI app for generating the faked records
14     such as names, addresses, and phone numbers.
15 EOF
```

对于这个文件来说, 必须具备的字段包括:

- Package – 无需解释
- Version – 无需解释
- Architecture – 目标系统的 CPU 架构, 一般来说是 amd64, 具体情况以后会有所讨论。有时候, 它们可以是 any 或者 all。
- Maintainer – deb 包的维护者, 及其邮件地址
- Description – 软件包的描述信息。多行也可以, 前面缩进至少一个空格。

可以给出更多字段, 其中需要解释的字段还有:

- Section: 可以给它一个值, 如 utils, admin, devel, doc, libs, net, 或者 unknown 等等。这是软件包的分类子类别, 代表着该软件包在 Debian 仓库中将被归属到什么样的逻辑子分类中。
- Priority: optional 优先级适用于与优先级为 required、important 或 standard 的软件包不冲突的新软件包。也可以做其它取值。若是不明了, 请使用 optional。
- Multi-Arch, Build-Depends, Standards-Version: 如果你想使用他们, 可以照搬。今后我们会对它们有所讨论。

- Depends : 软件包中的执行文件运行时所需要的各种依赖关系。本文中我们使用的是 golang app, 因此通常没有外部依赖的需要。

可以参考:[Debian 新维护者手册](#)。

请谨慎参考新维护者手册, 因为该文档所描述的内容是针对 source packger 的 deb 构建而言的。

所以可用的字段基本上应限制在我们已经列举出来的范围, 而且在 control 中的排列方式也请按照我们给出的范例来做, 请勿使用新维护者手册中 control 的样本格式。

3.2.2. changelog, compat, copyright

这些文件最好也被提供。

但它们是可选的。尤其是当你没有计划让 deb 包进入 Debian apt 仓库中的话。

3.3. 维护脚本

在 DEBIAN 文件夹中, 你还可以提供 preinst, postinst, prerm, postrm 文件, 它们应该是一个 shell 脚本文件, 具有 0755 执行权限, 并且遵循少许的约定。

首先来说, 请勿对目标系统的 Shell 环境做任何假设。例如, 你不要要求目标系统一定有 bash, 甚至一定要有 zsh——尽管几乎所有的 Debian 发行版默认时都至少会有 bash 存在, 但请别奢求。

因此这些维护脚本只应该使用 dash 语法, /bin/sh 提供一个最小集合的 shell 环境, 在 Debian 系的发行版中, /bin/sh 实际上是 dash。

dash 的脚本语法和 bash 的区别不算太大, 但功能要更受限一些。可以查阅 [shell - Difference between sh and Bash - Stack Overflow](#)。

在 faker 中提供了 postinst 和 postrm, 所以我们首先阐释它们。

3.3.1. postinst

postinst 是在核心安装完成后执行的脚本。核心安装通常是指微型文件系统被合并到目标系统中的过程。

faker 的 postinst 需要完成一系列的符号链接, 但并不需要其它更多操作。

我们实际的 postinst 如下：

```
1  #!/bin/sh -e
2
3  set -e
4
5  # ref: postinst in sudo package
6
7  case "$1" in
8  configure)
9      for f in $(ls -b /usr/lib/faker/man/man1/*.1.gz); do
10         fn=$(basename $f)
11         #echo " . f=$f, fn=$fn"
12         if [ ! -L /usr/share/man/man1/$fn ]; then
13             ln -s $f /usr/share/man/man1/$fn
14         fi
15     done
16
17     if [ ! -L /usr/bin/faker ]; then
18         ln -s /usr/lib/faker/bin/faker /usr/bin/
19     fi
20
21     if [ -d /etc/bash_completion.d ]; then
22         if [ ! -L /etc/bash_completion.d/faker ]; then
23             ln -s /usr/lib/faker/completions/bash/faker /etc/bash_completion.d/
24         fi
25     fi
26
27     if [ -x /bin/zsh ]; then
28         if [ ! -d /usr/local/share/zsh/site-functions ]; then
29             mkdir -pv /usr/local/share/zsh/site-functions
30         fi
31         if [ -d /usr/share/zsh/site-functions ]; then
32             if [ ! -L /usr/share/zsh/site-functions/_faker ]; then
33                 ln -s /usr/lib/faker/completions/zsh/_faker /usr/share/zsh/site-functions/
34             fi
35         elif [ -d /usr/local/share/zsh/site-functions ]; then
36             if [ ! -L /usr/local/share/zsh/site-functions/_faker ]; then
37                 ln -s /usr/lib/faker/completions/zsh/_faker /usr/local/share/zsh/site-functions/
38             fi
39         fi
40     fi
41
42     [ -d /etc/faker ] || ln -s /usr/lib/faker/etc/faker /etc/
43     ;;
44
45 abort-removal)
46     [ -L /usr/bin/faker ] && rm -rf /usr/bin/faker
47
48     for f in /usr/share/man/man1/faker-*.1.gz; do
49         [ -L $f ] && rm -rf $f
50     done
51     [ -L /usr/share/man/man1/faker.1.gz ] && rm -rf /usr/share/man/man1/faker.1.gz
```

```

52 [ -L /etc/bash_completion.d/faker ] && rm -rf /etc/bash_completion.d/faker
53
54 [ -L /usr/share/zsh/site-functions/_faker ] && rm -rf /usr/share/zsh/site-functions/_fal
55 [ -L /usr/local/share/zsh/site-functions/_faker ] && rm -rf /usr/local/share/zsh/site-fu
56 ;;
57
58 esac

```

Shbang 声明强调了一定要使用 `/bin/sh` (即 `dash`), 不做任何过分的预设。

`set -e` 代表着遇到脚本执行错误时立即终止整个脚本序列的执行。

`postinst` 在被安装核心调度执行时, 带有传入的参数 `$1`, 其值可能为 `configure`, `abort-remove`。初次安装时会得到 `configure`, 若是卸载失败时 `postinst` 也会被调用, 并带有参数 `abort-remove`。

我们的 `postinst` 看起来繁复, 但实际上全数为符号链接, 非常简单。

3.3.2. postrm

同样道理, 为了在卸载时释放上面建立的符号链接, 我们需要定制的 `postrm`:

```

1  #!/bin/sh -e
2
3  case "$1" in
4  purge)
5      rm -rf /var/lib/faker
6      ;;
7
8  remove)
9      [ -L /usr/bin/faker ] && rm -rf /usr/bin/faker
10
11     for f in /usr/share/man/man1/faker-*.1.gz; do
12         [ -L $f ] && rm -rf $f
13     done
14     [ -L /usr/share/man/man1/faker.1.gz ] && rm -rf /usr/share/man/man1/faker.1.gz
15
16     [ -L /etc/bash_completion.d/faker ] && rm -rf /etc/bash_completion.d/faker
17     [ -L /usr/share/zsh/site-functions/_faker ] && rm -rf /usr/share/zsh/site-functions/_fal
18     [ -L /usr/local/share/zsh/site-functions/_faker ] && rm -rf /usr/local/share/zsh/site-fu
19     ;;
20
21  upgrade | deconfigure)
22      #
23      ;;

```



```

24 abort-upgrade | failed-upgrade)
25     # if [ -e "/etc/sudoers.pre-conf" ]; then
26     # mv /etc/sudoers.pre-conf /etc/sudoers
27     # fi
28     ;;
29
30 *)
31     echo "unknown argument --> $1" >&2
32     exit 0
33     ;;
34 esac
35
36

```

postrm 可能被传入的参数更多，但通常需要被我们管许的是 remove 和 purge 两个动作。

3.3.3. preinst 和 prerm

我们也介绍以下 preinst 和 prerm，他们在安装前和卸载前被提前执行。

3.3.3.1. preinst

```

1  #!/bin/sh -e
2
3  case "$1" in
4      install|upgrade)
5          if [ -n "$2" ] && dpkg --compare-versions "$2" le "1.7.4p4-4"; then
6
7              SUDOERS="/etc/sudoers"
8
9              # if [ -e "$SUDOERS" ]; then
10             # md5sum="$(md5sum $SUDOERS | sed -e 's/ .*//')"
```

3.3.3.2. prerm

```

1  #!/bin/sh

```

```

2
3 set -e
4
5 check_password() {
6     if [ ! "$SUDO_FORCE_REMOVE" = "yes" ]; then
7         :
8     fi
9 }
10
11 case $1 in
12     remove)
13     check_password;
14     ;;
15     *)
16     ;;
17 esac
18
19 exit 0

```

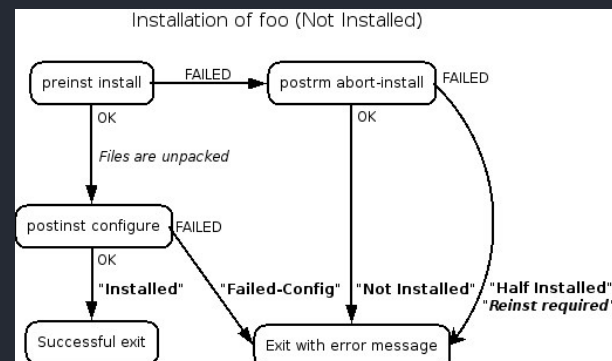
3.4. 维护脚本与安装行为间的关系

3.4.1. Install

安装一个 deb 可以通过:

```
1 | dpkg -i package.deb
```

这个过程图示如下:



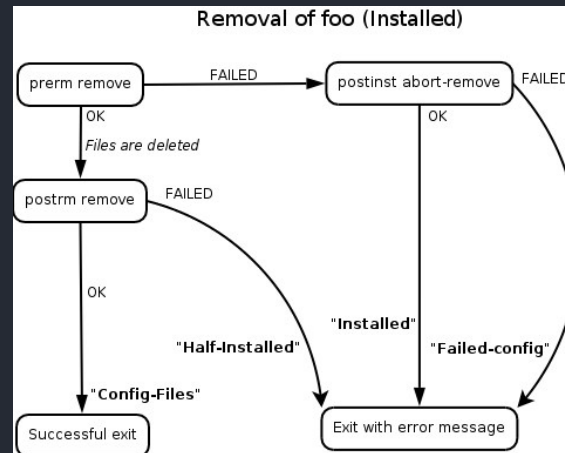
对于官方仓库而言, 安装是通过 apt install 来进行的, 本质不变。

3.4.2. Removal

卸载一个 package:

```
1 | dpkg -r <package-name>
```

其流程如下：



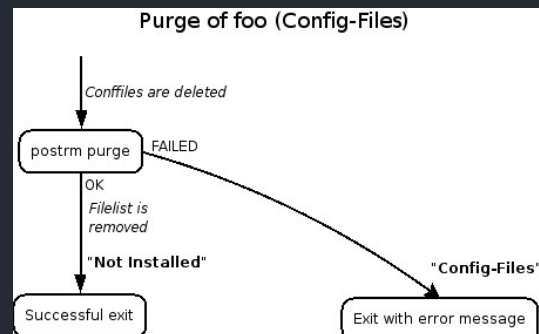
也可以通过 apt remove 的方式来卸载。

3.4.3. Purging

彻底清除软件包的存在痕迹, 是一个增强的特性, 软件包维护人员可以藉此消除自己的一切额外生成的内容。

```
1 | dpkg -P <package-name>
```

其流程如下：



也可以通过 apt purge 来达到目的。

3.4.4. More

还存在更多的软件包行为流程, 请查阅：


来获得较完整的表述。

3.5. 文件布局

经过上面的一系列工作, 已经准备好的 binary package 文件夹是这样的:

```
1 hz@u20s:~/deb/faker.work$ tree faker_0.2-5_amd64
2 faker_0.2-5_amd64
3 |— DEBIAN
4 |   |— changelog
5 |   |— compat
6 |   |— control
7 |   |— copyright
8 |   |— postinst
9 |   |— postrm
10 |— usr
11 |   |— lib
12 |       |— faker
13 |           |— LICENSE
14 |           |— bin
15 |               |— faker
16 |           |— completions
17 |               |— bash
18 |                   |— faker
19 |               |— zsh
20 |                   |— _faker
21 |           |— etc
22 |               |— faker
23 |                   |— certs
24 |                       |— cert.key
25 |                           |— cert.pem
26 |                   |— conf.d
27 |                       |— 10.base.yml
28 |                           |— 11.logger.yml
29 |                   |— faker.yml
30 |           |— man
31 |               |— man1
32 |           |— faker-addr.1.gz
33 |           |— ... (more gz here)
34 |           |— faker.1.gz
```

在这里, faker 被放置在 `./usr/lib/faker` 之中, 这也意味着它将被安装到目标机器的 `/usr/lib/faker` 里。

在 `./usr/lib/faker/etc` 中, 包含 faker 的配置文件。按照 [cmdr](#)  的习惯做法, 你可以将主配置文件 `faker.yml` 放在 faker 执行文件的同一目录, 或者是

`/etc/faker` 中。作为安装包来说, 我们当然会选择 `/etc/faker` 作为配置文件的标准位置, 这会符合 Linux 文件夹规范(以及 GNU 文件夹规范)。

在 `./usr/lib/faker/man` 之中包含 `faker` 的 `man1` 手册。手册页是通过 `faker gen man` 生成的。手册页未来会被链接到 `/usr/share/man/man1` 之中, 然后你就可以使用 `man faker` 或者 `man faker-addr` 来查阅 `faker` 及其子命令的手册页了。

在 `./usr/lib/faker/completions` 之中包含了 `bash/zsh` 自动完成脚本, 安装指令将会检测目标机器的环境并制作适当的符号链接, 从而使能 `faker` 的命令行自动完成功能。同样地, 这些完成脚本也是通过 [cmdr](#) 提供的生成器生成的(`faker gen sh --zsh|--bash`)。

3.6. 生成 deb

很简单:

```
1 | dpkg-deb --build --root-owner-group faker_0.2-5_amd64
```

这里要给出的是 `bianry package` 的文件夹名称:`faker_0.2-5_amd64`。

执行完成后, `faker_0.2-5_amd64.deb` 将被构建在当前文件夹中。

3.6.1. Linux 账户

在构建时, 必须带上 `--root-owner-group`, 这个选项使得 `bianry package` 的文件树的所有者被修正为 `root` 用户。这个修正动作能够防止 `deb` 安装时你打包的账户名在目标系统中不存在的问题, 那样会导致不恰当的所有者以及权限。

3.6.2. 面对 deb

你可以展开这个 `deb`:

```
1 | dpkg-deb -X <deb> <directory>
```

查看 软件包 包含的文件清单(如果包已经被成功安装):

```
1 | dpkg -L <package-name>
```

查看 `deb` 包含的文件清单:

```
1 | dpkg-deb -c <deb>
```

查看 deb 的描述信息(主要是源于 DEBIAN/control 的各种字段信息):

```
1 | dpkg -I <deb>
```

查看 deb 中的 DEBIAN 文件夹中所包含的文件清单:

```
1 | dpkg-deb --ctrl-tarfile sudo_1.8.31-1ubuntu1.2_amd64.deb | tar -t
2 | # 也可以借助 ar 来实现
```

更多功能可以通过:

```
1 | dpkg --help
2 | dpkg-deb --help
```

来获取。

3.6.3. 调试和除错

在试制过程中, 如果你的维护脚本写的不正确, 那么安装 deb 可能会有意料之外的失败。这不是奇怪的事, 问题在于此时 Debian 的包管理系统可能会进入到一种错误的状态。

如果安装一个 deb 导致了意外的终止, 你可以采用下面的序列来做清理:

```
1 | sudo mv /var/lib/dpkg/info/<packagename>.* /tmp/
2 | sudo dpkg --remove --force-remove-reinstreq <packagename>
```

有时候, 你可能疑惑于安装流程失败的原因。此时可以打开安装时的详细输出开关, 例如:

```
1 | sudo dpkg -D2 -i <deb-file>
```

man dpkg 能够查阅到 -D 的可能的取值:

1	Number	Description
2	1	Generally helpful progress information
3	2	Invocation and status of maintainer scripts
4	10	Output for each file processed
5	100	Lots of output for each file processed
6	20	Output for each configuration file
7	200	Lots of output for each configuration file
8	40	Dependencies and conflicts
9	400	Lots of dependencies/conflicts output
10	10000	Trigger activation and processing

11	20000	Lots of output regarding triggers
12	40000	Silly amounts of output regarding triggers
13	1000	Lots of drive! about e.g. the dpkg/info dir
14	2000	Insane amounts of drive!

多数情况下, 你可以直接使用 `-D2`, 它足够用了。

注意上表中的 Number 是八进制数值, 使用时原样照录即可。

3.6.4. 外部依赖

对于 binary package 的打包来说, 外部依赖需要被手工管理。但这可能是个足够麻烦的任务。

本文中我们使用的 `golang executable` 是不需要额外申明外部依赖关系的。

如果你打包的对象是其它可执行文件, 一个偷懒的手段是利用 `dpkg-shlibdeps` 工具程序, 这是来自元 `dh-make` 的小工具, 可以扫描执行文件的 `so` 依赖并建立 `Depends` 字段所需要的依赖关系声明。

```
1  $ mkdir debian
2
3  $ cat > debian/control <<EOF
4  Source: debhello
5  Section: devel
6  Priority: optional
7  Maintainer: Hedzr Yeh <hedzrz@gmail.com>
8  Build-Depends: debhelper (>=11~)
9  Standards-Version: 4.1.4
10 Homepage: https://hedzr.github.io/devhello/
11
12 Package: debhello
13 Architecture: any
14 Multi-Arch: foreign
15 Depends: $(misc:Depends), $(shlibs:Depends)
16 Description: auto-generated package by debmake
17 This Debian binary package was auto-generated by the
18 debmake(1) command provided by the debmake package.
19 EOF
20
21 $ dpkg-shlibdeps -e/usr/bin/nano -O
22 dpkg-shlibdeps: warning: binaries to analyze should already be installed in their package's
23 shlibs:Depends=libc6 (>= 2.27), libncursesw6 (>= 6), libtinfo6 (>= 6)
```

这里扫描了 `nano` 的依赖信息, 得到了最后一行的输出内容, 所以我们可以据此改写 `DEBIAN/control` 中的 `Depends` 字段为:

```
1 | Depends: libc6 (>= 2.27), libcursesw6 (>= 6), libtinfo6 (>= 6)
```

为了能顺利运行 `dpkg-shlibdeps` 工具，我们首先要在当前文件夹中准备一个 `debian/control` 文件，然后为其填写合法的字段内容，然后就可以执行 `dpkg-shlibdeps -e<executable> -O` 了。这个 `debian/control` 文件采用的是 `source package` 构建时的格式，详情可以精确地参考 [Debian 新维护者手册](#)，但直接取用我们给出的实例即可，`dpkg-shlibdeps` 只需要该 `debian/control` 存在且语法合规就行，并不在乎这个文件的语义也是正确的。

4. 后记

在上面建立的骨架里，你需要做的工作基本上就只在规划自己的文件结构，以及编写维护脚本上面了。

有关 `go-faker` 打包的全部代码在 [hedzr/deb-build-tutor](#) 可以找到。

5. 参考

- [Creating a custom Debian package - Leaseweb labs](#)
- [Creating a Debian Package \(.deb\)](#)
- [Building binary deb packages: a practical guide](#)
- [如何製作「deb檔\(Debian Package\)」](#)
- [Debian GNU/Linux Installation Guide](#)
- [Debian Policy Manual](#)
- [6. Package maintainer scripts and installation procedure — Debian Policy Manual v4.6.0.1](#)
- Debian documentation - [Chapter 7. Basics of the Debian package management system](#)
- [Debian 新维护者手册](#)
- [Ubuntu Packaging Guide](#)
- [Introduction to Debian packaging](#)
- Debian Packaging Tutorial: [pdf](#)
- [5.2. Package Meta-Information - The Debian Administrator's Handbook](#)



标签: [binary package](#) [deb](#) [packaging](#)

分类: [deb](#) [packaging](#)

更新时间: December 17, 2021

许可信息: 本文采用知识共享署名-非商业性使用-相同方式共享 4.0 国际许可协议进行许可, 转载请注明出处 [hedzr](#) 或出处链接。

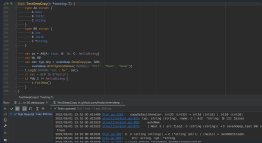
分享

[Twitter](#) [Facebook](#) [LinkedIn](#)

[向前](#) [向后](#)

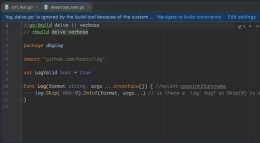
留下评论

猜您还喜欢



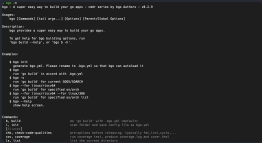
容易定制的 deepcopy 反射库 evendeep

7 分钟阅读



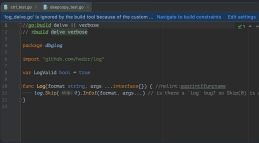
[Go-Lib] 实现基本类型之上的类型扩展

3 分钟阅读



gochecknoinits - init() 不是真的很要不得

少于 1 分钟阅读



在 Go 中实现更好的埋点日志功能

2 分钟阅读

evendeeep 提供 deep 系列工具:
deepcopy, deepdiff 以及
deepequal ...

类型扩展, 与库作者谈研发 ...

关于 lint 时遇到的 ...

埋点日志, 与库作者谈研发 ...

关注: [TWITTER](#) [GITHUB](#) [FEED](#)

© 2022 hedzr. 技术来自于 Jekyll & Minimal Mistakes.

