LINUX: créer un paquet deb/rpm pour Linux

Table of Contents

RPM et DEB sont des formats d'emballage et de distribution de logiciels binaires utilisés respectivement par les distributions RedHat/Debian. Les deux prennent la source du paquet en amont (généralement une archive) et un ensemble de scripts/règles pour construire un paquet binaire.

L'article aborde la construction de ces paquets :

- RPM avec rpmdev/mock
- DEB avec dpkg-buildpackage/debuild/pbuilder
- RMP/DEB avec FPM

RPM

rpmbuild

- **rpmbuild** est utilisé pour construire des packages RPM . Il prend un fichier de spécifications avec les steps/snippets et un arbre de construction avec les fichiers et génère un rpm (et éventuellement srpm) à partir de la source dans une archive.
- **rpmdev-newspec** est utilisé pour créer un nouveau modèle de fichier de spécifications
- **rpmdev-setuptree/rpmdev-wipetree** sont utilisés pour créer/effacer une arborescence de génération RPM.

installation

```
$ sudo yum group info "Development Tools" (includes gcc, rpmdevtools, mock)
```

usage

```
rpmbuild -bSTAGE|-tSTAGE [ rpmbuild-options ] FILE ...
```

- **-t** utilise les spécifications à l'intérieur de l'archive
- **-b** utilise un fichier de spécification fourni
- **–showrc** affiche les macros rpm utilisées dans les fichiers de spécifications

-bSTAGE, -tSTAGE construit jusqu'à l'étape STAGE par exemple :

- **-ba** construit les paquets source et binaire (après avoir effectué les étapes %prep, %build et %install)
- **-bb** construit un paquet binaire (après avoir effectué les étapes %prep, %build et %install)
- **-bp** exécute l'étape “%prep” à partir du fichier de spécifications
- **-bc** effectue l'étape “%build” à partir du fichier de spécifications (après l'étape %prep)
- **-bi** effectue l'étape “% install” à partir du fichier de spécifications (après avoir effectué les étapes %prep et %build)
- **-bl** effectue une “vérification de liste”, la section “%files” du fichier de spécifications est une macro développée
- **-bs** construit uniquement le paquet source

exemple

```
$(myusername) rpmdev-setuptree
$ cd ~/rpmbuild/SOURCES
$ wget http://ftp.gnu.org/gnu/hello/hello-2.8.tar.gz
$ cd ~/rpmbuild/SPECS
$ rpmdev-newspec hello
```

comme on utilise des fichiers i18, ajouter ensuite 'gettext' comme condition de construction, utiliser

'%find_lang%{name}' pour les fichiers dans '%install' et **'%files -f%{name}.lang'** pour trouver les fichiers

commme on utilise également des fichiers d'informations, utiliser **'install-info'** dans '%post/%preun' pour installer/désinstaller du système.

```
$ cat hello.spec
Name:           hello
Version:        2.8
Release:        1%{?dist}
Summary:        The "Hello World" program from GNU
License:        GPLv3+
URL:            http://ftp.gnu.org/gnu/%{name}
Source0:        http://ftp.gnu.org/gnu/%{name}/%{name}-%{version}.tar.gz
BuildRequires: gettext
Requires(post): info
Requires(preun): info
%description
The "Hello World" program, done with all bells and whistles of a proper FOSS
project,
including configuration, build, internationalization, help files, etc.
%prep
%setup -q
%build
%configure
make %{?_smp_mflags}
%install
%make_install
%find_lang %{name}
rm -f %{buildroot}/%{_infodir}/dir
%post
/sbin/install-info %{_infodir}/%{name}.info %{_infodir}/dir || :
%preun
if [ $1 = 0 ] ; then
/sbin/install-info --delete %{_infodir}/%{name}.info %{_infodir}/dir || :
fi
%files -f %{name}.lang
%doc AUTHORS ChangeLog COPYING NEWS README THANKS TODO
%{_mandir}/man1/hello.1.gz
%{_infodir}/%{name}.info.gz
%{_bindir}/hello
%changelog
* Tue Sep 06 2011 The Coon of Ty  2.8-1
- Initial version of the package

$ rpmbuild -ba hello.spec
...
~/rpmbuild/SRPMS/hello-2.8-1.el7.centos.src.rpm
```

contrôle de conformité (rpmlint)

```
$ rpmlint hello.spec ../SRPMS/hello* ../RPMS/*/hello*
...
$ rpmlint -I description-line-too-long
Your description lines must not exceed 80 characters. If a line is exceeding
this number, cut it to fit in two lines.
```

mock

mock prend un srpm et crée un environnement de construction chroot garantissant que les exigences de construction sont correctes. On peut également l'utiliser pour construire dans un environnement différent ou pour tester les versions de paquet.

```
$ sudo usermod -a -G mock myusername
$ newgrp mock
```

compiler dans un env chrooté

Pour compiler dans le répertoire /var/lib/mock/epel-6-x86_64/root

```
$ export MOCK_CONFIG=/etc/mock/epel-6-x86_64.cfg
$ /usr/bin/mock -r $MOCK_CONFIG --verbose --rebuild ../SRPMS/*.src.rpm
$ ls /var/lib/mock/epel-6-x86_64/result
build.log                hello-2.8-1.el6.x86_64.rpm            root.log
hello-2.8-1.el6.src.rpm  hello-debuginfo-2.8-1.el6.x86_64.rpm  state.log
$ /usr/bin/mock -r $MOCK_CONFIG --clean
```

utilisation des packages externes

si on a besoin de paquets qui ne sont pas dans le repo, installer le package dans l'env chrooté puis copier le SRPM du package dans le répertoire SRPM du paquet à construire avant de compiler:

```
$ /usr/bin/mock -r $MOCK_CONFIG --init
$ /usr/bin/mock -r $MOCK_CONFIG --install PACKAGE_NAME_OR_PATH_TO_RPM
$ /usr/bin/mock -r $MOCK_CONFIG --no-clean /PATH/TO/SRPM
$ /usr/bin/mock -r $MOCK_CONFIG --copyin /PATH/TO/SRPM /tmp
$ mock -r $MOCK_CONFIG --shell
$ cd ; rpmbuild --rebuild /tmp/SRPM_NAME
```

Accélération

```
$ cat /etc/mock/site-defaults.cfg
```

```
config_opts['macros']['%_smp_mflags'] = "-j17"
```

DEB

Hiérarchie des commandes/outils

- **script debian/rules** pour la construction du paquet
- **dpkg-buildpackage** outil de construction de paquet
- **debuild** dpkg-buildpackage + lintian pour construire avec des variables d'environnement assainies
- **pbuilder** pour utiliser un environnement chrooté Debian
- **pdebuild** pbuilder + dpkg-buildpackage pour construire dans le chroot
- **cowbuilder** pour accélérer l'exécution de constructeur
- **git-pbuilder** la syntaxe de ligne de commande facile à utiliser pour pdebuild (utilisée par gbp buildpackage)
- **gbp** gère le source Debian sous le dépôt git
- **gbp buildpackage** pbuilder + dpkg-buildpackage + gbp

dpkg-buildpackage

- **dpkg-buildpackage** est utilisé pour construire des packages binaires ou sources à partir de sources.
- **dh_make** est utilisé pour créer l’arborescence liée Debian à partir d’une arborescence source normale,
- **dpkg-deb** est utilisé pour manipuler l’archive de paquetages .deb
- **dpkg** est utilisé pour gérer/installer/désinstaller des paquetages.

installation

```
$ sudo apt-get install dh-make debhelper devscripts fakeroot
```

usage

```
dpkg-buildpackage [...]
```

- **-F** (default) builds binaries and sources
- **-g,-G** source and arch-indep/specific build
- **-b,-B** binary-only, no-source/arch-specific/arch-indep
- **-S** source-only
- **-tc,-nc** clean/dont-clean source trees when finished
- **-us,-uc** unsigned source package / changes file.

exemple

Récupérer les sources

```
$ mkdir hello ; cd $_
$ wget http://ftp.gnu.org/gnu/hello/hello-2.6.tar.gz
$ tar zxf hello-*.tar.gz ; cd hello-*
```

Créer les fichiers de contrôle avec 'dh_make'

note: ne jamais relancer dh_make

```
$ dh_make -s -y --copyright gpl -f ../hello-*.tar.gz
$ cd debian
$ rm -f rm *.ex *.EX README.Debian info docs
$ tree
.
├── changelog
├── compat
├── control
├── copyright
├── README.source
├── rules
└── source
    └── format
```

mettre à jour changelog

```
$ dch -i "Added README.Debian"
```

Editer le fichier de contrôle

Ajouter les 'Build-Depends' supplémentaire; 'depend' est automatiquement ajouté

```
$ cat control
Source: hello
Section: unknown
Priority: optional
Maintainer: Rui Coelho
Build-Depends: debhelper (>= 9), autotools-dev
Standards-Version: 3.9.5
Homepage:
#Vcs-Git: git://anonscm.debian.org/collab-maint/hello.git
#Vcs-Browser: http://anonscm.debian.org/?p=collab-maint/hello.git;a=summary
```

```
Package: hello
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Description:
```

Lancer le build

Utiliser le fichier rules pour appeler Makefile

```
$ cd .. ; dpkg-buildpackage -rfakeroot
```

Contrôler les résultats du build

```
$ cd .. ; ls
hello-2.6  hello_2.6-1_amd64.changes  hello_2.6-1_amd64.deb
hello_2.6-1.debian.tar.xz  hello_2.6-1.dsc  hello_2.6.orig.tar.gz
hello-2.6.tar.gz
```

1. **.dsc** est le contenu du code source sumamry, utilisé lors de la décompression de la source avec 'dpkg-source'
2. **.debian.tar.xz** est le répertoire 'debian', chaque mise à jour est stockée comme patch dans 'debian/patches'
3. **.deb** est un binaire complet, utiliser 'dpkg' pour l'installer/le supprimer
4. **.changes** décrit les modifications apportées, en partie générées à partir de changelog et '.dsc'

```
  $ lintian hello_*.dsc
  $ lintian hello_*.deb
  $ lesspipe hello_*.deb
```

Installer le paquet

```
$ sudo dpkg --install hello_*.deb
$ /usr/bin/hello
Hello, world!
$ sudo apt-get remove hello
```

Re-ceration du paquet "from scratch"

Utiliser '.dsc,.orig,.debian'

```
$ dpkg-source -x *.dsc
$ cd hello* ; dpkg-buildpackage -rfakeroot
```

debuild

On peut automatiser le processus de construction en utilisant la commande dpkg-buildpackage et empaqueter ensuite avec la commande debuild :

- **debuild** est un wrapper autour de dpkg-buildpackage + lintian utilisé pour empaqueter les nouvelles versions amont.
- **uupdate** est utilisé pour mettre à niveau un package de code source à partir d'une révision en amont.

La commande debuild exécute la commande lintian pour une analyse statique après la construction du paquet Debian. La commande lintian peut être personnalisée dans ~/.devscripts ou dans /etc/devscripts.conf comme suit :

```
DEBUILD_DPKG_BUILDPACKAGE_OPTS="-us -uc -I -i"
DEBUILD_LINTIAN_OPTS="-i -I --show-overrides"
```

usage

```
debuild [debuild options] binary|binary-arch|binary-indep|clean ...
```

exemple

```
$ cd hello ; wget http://ftp.gnu.org/gnu/hello/hello-2.7.tar.gz
$ cd hello-2.6 ; uupdate -u hello-2.7.tar.gz
$ cd ../hello-2.7; debuild -rfakeroot
$ ls ../*2.7*
hello_2.7-0ubuntu1_amd64.build    hello_2.7-0ubuntu1.debian.tar.xz
hello-2.7.tar.gz
hello_2.7-0ubuntu1_amd64.changes  hello_2.7-0ubuntu1.dsc
hello_2.7-0ubuntu1_amd64.deb      hello_2.7.orig.tar.gz
$ debuild clean
```

Autre exemple (package sans make)

créer l'environnement

```
$ mkdir myapp-0.1 ; cd myapp*
$ echo -e '#!/bin/bash\necho Hello World' > hello.sh
$ dh_make -s --indep --createorig
$ rm debian/*.ex
```

spécifier les fichiers à installer

```
$ echo hello.sh /usr/bin > debian/install
```

modifier le format source

```
$ echo "1.0" > debian/source/format
```

ajouter les dépendances dans la section 'Depends' de 'debian/control'

```
$ cat debian/control
```

construire le paquet

```
$ debuild -us -uc
$ ls ../*deb
../myapp_0.1-1_all.deb
```

tester le paquet

```
$ sudo dpkg -i myapp_0.1-1_all.deb
$ /usr/bin/hello.sh
Hello World
$ sudo dpkg -r myapp
```

pbuilder

- **pbuilder** est utilisé pour créer et maintenir un environnement chroot et pour construire un paquet Debian dans un environnement chroot.
- **pdebuild** cumule les fonction de pbuilder et dpkg-buildpackage pour construire dans un chroot.

installation

```
# sudo apt-get install pbuilder
```

exemple

creer/mattre à jour la base '/var/cache/pbuilder/base.tgz' dans un env chrooté,

```
$ sudo pbuilder clean
# note: use '--distribution sid' to switch distro
$ sudo pbuilder create
$ sudo pbuilder --update
```

recompiler le package

```
$ sudo pbuilder --build hello_2.7-0ubuntu1.dsc
$ ls /var/cache/pbuilder/result/
```

```
hello_2.7-0ubuntu1_amd64.changes  hello_2.7-0ubuntu1.debian.tar.xz
hello_2.7.orig.tar.gz
hello_2.7-0ubuntu1_amd64.deb      hello_2.7-0ubuntu1.dsc
```

signer les fihiers '.dsc' and '.changes'

```
$ cd /var/cache/pbuilder/result/
$ debsign hello_2.7-0ubuntu1_amd64.changes
```

récupérer les sources

Dans l'env chrooté

```
$ sudo pbuilder --login --save-after-login
```

soit diredtement depuis sources.list

```
$ cat /etc/apt/sources.list
deb-src http://archive.ubuntu.com/ubuntu <ubuntu_version> main restricted
universe multiverse
```

soit en utilisant dget depuis 'http://packages.ubuntu.com/' ou 'http://packages.debian.org/'

```
$ dget http://ftp.de.debian.org/debian/pool/main/h/hello/hello_2.9-2.dsc
$ sudo pbuilder --build hello_2.9-2.dsc
$ ls /var/cache/pbuilder/result/hello_2.9*
```

Utiliser pdebuild dans l'env chrooté

```
$ apt-get source hello
$ cd hello-2.9 ; pdebuild
```

FPM

fpm «Effing Package Management» construit des packages pour plusieurs plates-formes (deb, rpm, etc.) avec une grande facilité et une sécurité absolue.

installation

```
$ sudo apt-get install ruby-dev gcc | sudo yum install ruby-devel gcc
$ sudo gem install fpm
```

usage

```
fpm -s <source type> -t <target type> [list of sources]...
```

- **-n name, -v version** package name and version
- **-C chdir** change directory before searching files
- **–prefix prefix** path to prefix when building
- **-d,–depends dep>version** dependencies
- **-a,–architecture arch** architecture, usually 'uname -m'
- **–inputs file** file with newline-separated list of files and dirs
- **–before/after-install/remove/upgrade file** script to run at given stage

exemples

créer un deb noarch à partir du contenu d'un répertoire

```
$ echo -e '#!/bin/bash\necho Hello World' > hello.sh
$ fpm -s dir -t deb -a all -n hello -v 1.0 --prefix /usr/bin hello.sh
$ dpkg -c *.deb
... ./usr/bin/hello.sh
$ sudo dpkg -i *.deb
$ . /usr/bin/hello.sh
Hello World
$ sudo dpkg -r hello
```

créer des fichiers deb à partir de python modulo avec easy_install

```
$ fpm -s python -t deb django
$ dpkg -c *.deb
... ./usr/local/lib/python2.7/dist-packages/django
```

convertir une source de paquet Python locale

```
$ fpm -s python -t deb myproj/setup.py
```

construire un paquet gnu

```
$ wget http://ftp.gnu.org/gnu/hello/hello-2.9.tar.gz
$ tar -zxf hello-*.tar.gz
$ cd hello* ; ./configure --prefix=/usr ; make ; make install
DESTDIR=/tmp/installdir
$ cd .. ; fpm -s dir -t deb -n hello -v 2.9 -C /tmp/installdir usr
$ dpkg -c hello*.deb
```

```
... ./usr/bin/hello
```