由淺入深說 PKGBUILD 打包

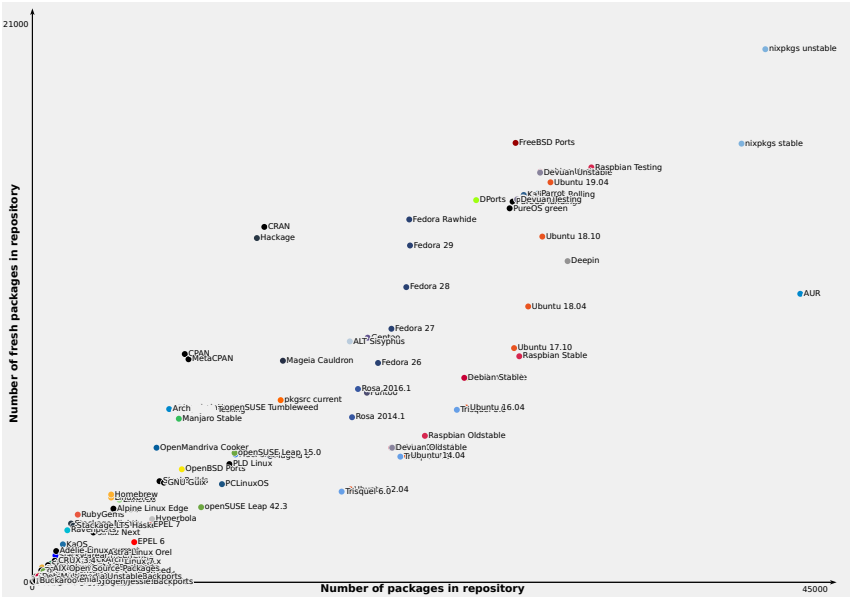
目次

- 包版本號
- 包元數據
- 包間關係

包到用時方恨少，碼非寫過不知難

即便是 Arch Linux 初學者，也鮮有完全靠官方源中的軟件包就能存活的， 或多或少都得依賴一些第三方源或者 AUR 中的額外軟件包補充可用軟件庫。 最近發現一個非常有意思的項目 Repology ， 總結了諸多自由開源軟件發行版們軟件庫中軟件包，其中還有一張總結性的圖示， 橫總對比各個軟件源的包數量和新舊程度。

Repology 軟件源對比圖 （放大） （來源）



可以在 Repology 的軟件源對比圖中找找代表 Arch Linux 和 AUR 的兩個 Arch 藍小圓點 🔵。 Arch Linux 的小圓點 🔵 位於斜對角線左側， AUR 的小圓點 🔵 則遠居於整張圖的最右側。 從中可以看出 Arch Linux 官方源的包數量相對較少（部分是因爲粗放打包策略）而更新相對及時， 另一邊 AUR 的包數量就非常豐富，但更新卻不那麼及時了。 官方軟件源提供了一個更新及時並且足夠穩定的基礎，在此之上利用 AUR 補充可用軟件包，這對於 Arch Linux 用戶來說也即是常態。

AUR 不同於二進制軟件源的是，它只是一個提供 PKGBUILD 腳本的共享網站。 對於二進制軟件源而言，可以直接在 `/etc/pacman.conf` 中添加，然後就可安裝其中的軟件， 而對於 AUR 中的軟件包，需要自行下載 PKGBUILD 並將之編譯成二進制包。 由於 AUR 上軟件包維護者衆多，於是打包質量也參差不齊，難免遇到一些

包需要使用者手動修改 PKGBUILD 纔可正常打包。於是對於 AUR 用戶而言， **理解並可修改** PKGBUILD 以定製打包過程非常重要。即便使用 AUR Helpers 幫助自動完成打包工作，對 PKGBUILD 的理解也不可或缺。

是可謂： **包到用時方恨少，碼非寫過不知難**

包到用時方恨少: 很多時候想要用的包不在官方源，去 AUR 一搜雖有，卻不知其打包的質量如何，更新的頻度又怎樣。

碼非寫過不知難: 於是從 AUR 下載下來的 PKGBUILD，用編輯器打開，卻不知從何看起，如何定製。

經常有人讓我寫寫關於 Arch Linux 中打包的經驗和技巧，因此本文就旨在由淺入深地梳理一下我個人對 PKGBUILD 打包系統的理解和體會。社區中有如 felixonmars 這樣掌管 Arch Linux 軟件包近半壁江山的老前輩，有如 lilydjwg 這樣創建出 lilac 自動打包機器人並統領 `[archlinuxcn]` 軟件源的掌門人， 也有各編程語言各編譯環境專精的行家裏手， 說起打包的經驗實在不敢班門弄斧地宣稱涵蓋打包細節的方方面面，只能說是我個人的理解和體會。

再者， Arch Linux 的打包系統也是時時刻刻在不斷變化、不斷發展的，本文發佈時所寫的內容， 在數月乃至數年之後可能不再適用。任何具體的打包細節都請參閱 PKGBUILD 和 makepkg 的相關手冊頁，以及

archwiki 上相應的 PKGBUILD 。 本文的目的僅限於對上述官方文檔提供一條易於入門的脈絡，不能作爲技術性參考或補充。

好，廢話碼了一屏，尚未見半句乾貨，就跟我一起從最基礎的部分開始吧。

EASY

講解 PKGBUILD 之前，想先大概看看 PKGBUILD 長什麼樣子。 AUR 上有大量 PKGBUILD 可以下載， Arch Linux 官方源中打包的官方包也有提供公開渠道下載打包時採用的 PKGBUILD ，這些都可以拿來作爲參考。於是作爲第一步，如何獲取 AUR 或者官方源中包的 PKGBUID 呢？

獲取現有的 PKGBUILD

Arch Linux 老用戶們已經很熟悉 AUR helpers 和 Arch Build System 那一套了，每個人可能都有兩三個自己趁手的常用 AUR helper 自動化打包。不過其實，大概從3年前 AUR web v4 發佈開始，已經不需要專用工具，直接用 `git` 就可以很方便地下載到 官方源和 AUR 中的 PKGBUILD 。

對於 Arch Linux 官方源中的軟件包，根據它是來自 [core]/[extra] 還是來自 [community] 我們可以用以下方式獲取對應的 PKGBUILD ：

```
# 獲取 core/extra 中包名爲 glibc 的包，寫入同名文件夾
git clone https://git.archlinux.org/svntogit/packages.git/ -b packages/glibc --single-branch glibc
# 獲取 community 中包名爲 pdfpc 的包，寫入同名文件夾
git clone https://git.archlinux.org/svntogit/community.git/ -b packages/pdfpc --single-branch pdfpc
```

對於官方源中的包，以上方式 clone 到的目錄結構是這樣：

```
pdfpc
├── repos
│   └── community-x86_64
│       └── PKGBUILD
└── trunk
    └── PKGBUILD
```

其中 trunk 文件夾用於時機打包， repos 文件夾則用於跟蹤這個包發佈在哪些具體倉庫中。 由於區分倉庫狀態和架構，以前還在支持 i686 的時候，打出的包可能

位於 `community-testing-i686` 或者 `community-staging-x86_64` 這樣的文件夾中。這些細節不需要關心，我們只需要 `trunk` 中的文件就可以打包了。

對於 AUR 中的軟件包，可以直接用以下方式獲取 PKGBUILD ：

```
# 獲取 AUR 中包名爲 pdfpc-git 的包，寫入同名文件夾
git clone aur@aur.archlinux.org:pdfpc-git.git
```

不同於官方源， AUR 中包沒有深層的目錄結構，直接在文件夾中放有 PKGBUILD ：

```
pdfpc-git
└── PKGBUILD
```

爲了方便鍵入， 在我的 bash/zsh 配置中 提供了幾個函數 `Ge` `Gc` `Ga` 分別用於獲取 [core]/[extra]，[community] 或是 AUR 中的 PKGBUILD ，需要的可以自己取用，對於 zsh 用戶還有這些命令的 自動補全包名 。

PKGBUILD 的構成要素

拿到了 PKGBUILD，就先用文本編輯器打開它看一眼吧，以 pdfpc 的 PKGBUILD 爲例：

```
 1  # Maintainer: Jiachen Yang <farseerfc@archlinux.org>
 2
 3  pkgname=pdfpc
 4  pkgver=4.2.1
 5  pkgrel=1
 6  pkgdesc='A presenter console with multi-monitor support for PDF files'
 7  arch=('x86_64')
 8  url='https://pdfpc.github.io/'
 9  license=('GPL')
10
11  depends=('gtk3' 'poppler-glib' 'libgee' 'gstreamer' 'gst-plugins-base')
12  makedepends=('cmake' 'vala')
13
14  source=("$pkgname-$pkgver.tar.gz::https://github.com/pdfpc/pdfpc/archive/v$pkgver.tar.gz")
15  sha256sums=('f67eedf092a9bc275dde312f3166063a2e88569f030839efc211127245be6df8')
16
17  build() {
18      cd "$srcdir/$pkgname-$pkgver"
19      cmake  -DCMAKE_INSTALL_PREFIX="/usr/" -DSYSCONFDIR="/etc" .
20      make
```

```
21  }
22
23  package() {
24      cd "$srcdir/$pkgname-$pkgver"
25      make DESTDIR="$pkgdir/" install
26  }
```

PKGBUILD 文件的格式本質上是 bash 腳本，語法遵從 bash 腳本語言，只不過有些預先確定好的內容需要撰寫。 粗看上面的 PKGBUILD 大體可以分爲兩半，前一半 3~15 行定義了很多變量和數組，後一半 17~26 行定義了一些函數。也即是說， PKGBUILD 包含兩大塊內容：

1. 該包是什麼，也即包的元數據(metadata)
2. 當如何打包，也即打包的過程

其中包的元數據又可大體分爲三段：

1. 對包的描述性數據。對應上面 3~9 行的內容。這裏寫這個包叫什麼名字，版本是什麼，協議用什麼……
2. 這個包與其它包的關係。對應上面 11,12 行。這裏寫這個包依賴哪些包，提供哪些虛包，位於什麼包組……
3. 包的源代碼位置。對應上面 14,15 行。這裏描述這個包從什麼地方下載，下載到的文件校驗，上游簽名……

這些元數據以 bash 腳本中定義的 變量(variable) 和數組(array) 的方式描述。應當定義哪些， 每個數據的含義，在 手冊頁 和 PKGBUILD 都有詳盡介紹，下文要具體說明的內容也會相應補充。

隨後打包過程則是以確定名稱的 bash 函數(function) 的形式描述。在函數體內直接書寫腳本。 一個包至少需要定義一個 `package()` 函數，它用來寫「安裝」文件的步驟。 如果是用編譯型語言編寫的軟件，那麼也應該有 `build()` 函數，用來寫 配置(configure) 和編譯的步驟。

PKGBUILD 一開始有一行註釋以 `Maintainer:` 開頭，這裏描述這個 PKGBUILD 的維護者信息，算作是記錄對打包貢獻，同時也在打包出問題時留下聯絡方式。如果 PKGBUILD 經手多人，通常當前的維護者寫在 `Maintainer:` 中，其餘的貢獻者寫作 `Contributor:`。這些信息雖然在 AUR 網頁界面中也有所記錄，不過留下註釋也可算作補充。

關於自定義變量

> PKGBUILD 中定義的 bash 變量和函數是導出給 makepkg 負責讀取的，於是這些變量名和函數名 的具體含義有所規定，不能隨便亂寫。不過如果有重複定義的內容，那麼還是可以自定義變量。 通常自定義變量名函數名會以下劃線 (_) 開頭，以和 makepkg 需要的變量名函數名區分。

例如， `pkgname` 需要加前綴後綴的情況下，通常常見的是定義一個 `_pkgname` 作爲項目上游的名稱，然後讓 `pkgname=${_pkgname}-git` 。再如，上游

包的命名

第3行 `pkgname` 定義了包的名字，這個變量的值應當和 AUR 上提交的軟件包相同， 也應儘量符合上游對項目的命名。定義包名同時也應儘量符合 Arch Linux 中現有軟件包的命名方式， 並且在 AUR 上提交的軟件包名還有些額外約定俗成的規則：

- 如果是編譯自版本控制系統(VCS, Version Control System)中檢出的最新源代碼， 應該在上游項目名後添加 `-vcs` 後綴。比如由 `git clone` 得到的 GitHub 上寄宿的上游軟件通常會有 `-git` 這樣的後綴。
- 如果是對現有二進制做重新打包，應該在上游項目名後添加 `-bin` 後綴。比如上游發佈了用於 Debian 系統的二進制包，想要重新打包成可用於 Arch Linux 的包，則要加 `-bin` 後綴。
- 對於特定語言需要的庫，通常會有語言名作爲前綴。不過這個規則的特例是，如果這個庫同時也在 `/usr/bin` 中提供可執行的命令，那麼包名可以沒

有前綴，或者對包進行拆包， 把庫和可執行命令分列在不同的包裏。一個例子是 powerline 包提供可執行程序，而它依賴的 python-powerline 則提供 python 的庫。

- 對於 Arch Linux 官方源中已經有的軟件包，如果想稍作修改之後將修改版共享在 AUR ，那麼通常 AUR 上的包名會是在官方源中對應包的包名，加上簡短的單詞描述所做的修改。 比如 telegram-desktop-systemqt-notoemoji 就是對官方源中 telegram-desktop 基礎上換用 NotoEmoji 的修改。 並且實際上官方源的 telegram-desktop 曾經在 AUR 中叫 telegram-desktop-systemqt ，因爲有來自 Debian 的 SystemQt 補丁。在被移入官方源之後去掉了 `-systemqt` 後綴。

一些有趣的包名字符統計

```
 1  $ # 三個官方源總體包數量
 2  $ pacman -Slq core extra communi
ty | wc -l
 3  10225
 4  $ # 除了小寫字母、數字、短橫、點之外有
別的字符的包名數量
 5  $ pacman -Ssq | grep "[^-a-z0-9.
]" -c
 6  192
 7  $ # 除了小寫字母、數字、短橫、點、下劃
線、加號之外有別的字符的包名數量
 8  $ pacman -Ssq | grep "[^-a-z0-9.
_+]" -c
 9  4
```

另外關於包名中可以使用的字符，在 PKGBUILD#pkgname 有說明可以用：

1. 英文大小寫字母
2. 數字
3. 這些符號： `@._+-`

一般來說，包名的字符會符合 Arch Linux 官方源中現有的包名的命名風格。 絕大多數包名是 **純小寫字母** 加上 **數字或者點(.)**，單詞之間用短橫(-) 分隔。 另外還有少數包名中出現大寫字母或者下劃線分隔，或者 C++ 相關的包名中出現加號(+)。

對包命名的基本原則是好記好搜，如果知道上游項目的名字，應該能很方便地搜到包對應的名字。 不那麼好搜的比如如果上游項目叫 `PyQt` 而 pkgname 叫 `python-qt` 那麼會讓搜索更加困難，所以請不要這樣命名。

另外關於包名中帶的版本號或者數字，除了個別情況之外一般而言 Arch Linux 打包不會給包名本身寫上 版本。一種特例是當某個上游庫發佈了新版本，一部分依賴該庫的程序還沒有兼容新版，這時通常的做法是 把老版本的庫的包名後面加上版本號，和新版區分，然後讓還沒有兼容的程序依賴帶版本包名的老版， 其餘依賴新版。比如官方源中的 lua 和 lua51 就是這樣的關係。 不過這種做法只是過渡，長期來看大部分包名中都不會有版本數字。

包版本號

版本號由3個變量描述 `epoch` ，`pkgver` ，`pkgrel` 。由 pacman 顯示時， 這三者顯示爲 `epoch:pkgver-pkgrel` 這種樣子。比如 toxcore 的版本號是 `1:0.2.8-1` 的話，也即是說 `epoch=1` ， `pkgver=0.2.8` ， `pkgrel=1` 。這三者中最重要的是 `pkgver` ，這與上游的版本號相對應，前後的 `epoch` 和 `pkgrel` 則是下游打包時指定的。

其中特殊而比較少見的是 `epoch` ，默認不寫時值是 0 ，這主要是用來應對上游改變了版本號命名方式的時候用。 比如一開始上游用日期 20190101 這樣的版本號，後來轉用 1.0 發佈了新版，讓 pacman 的 vercmp 判斷的話 `20190101 > 1.0` 會認爲 1.0 更老。這時候就需要加上 `epoch=1` 從而 `20190101 < 1:1.0` 。增加 epoch 需要相對謹慎，因爲一旦加上去，就很難再減下來了。 根據 man PKGBUILD , `epoch` 的值應該是一個自然數。

版本號主要部分 `pkgver` 來自上游，也就是說同樣一份源代碼，如果打幾次不同的包， 應該有相同的 pkgver 。 pkgver 不由打包者定義，於是可以用的字符比較自由， 不過一般是點隔開的數字的形式，可能會有字母。數字和字母混雜的版本號之間如何比較大小，在 pacman 有定義，可以參考 `man vercmp` 。值得注意 pacman 的定義可能和上游社區比如 PyPI 之類的地方的定義有所不同。

最后 `pkgrel` 是同一套源碼再次打包時遞增的發行版本號，通常是個正整數，可選得可以有小數。

包元數據

上面例子中 6 至 9 行定義了一些元數據。

pkgdesc: 通常來自上游項目的一行描述，會在

`pacman -Ss` 時顯示，其中出現的詞也會作爲搜索關鍵詞。

arch: 是計算機架構的一個數組。常見 `arch=('any')` 表示架構無關的包， `arch=('x86_64')` 表示 x86_64 架構下的二進制包。這個後文講述拆包時詳述。

url: 上游項目的 URL 地址，會在 Arch Linux 包列表的界面上顯示一個鏈接。

license: 採用的開源協議。協議名不能隨便寫，常用協議的協議名在 `/usr/share/licenses/common/` 有個完整列表，不在其中的協議就被認爲是不常用的協議。 雖然 MIT 和 BSD 這些開源協議很常見，但是不算在常用協議中。 不常用的協議 **要求** 在打包時同時安裝協議文件到 `/usr/share/licenses/<pkgname>/` 目錄中去。

包間關係

上面例子中 11 和 12 行定義了 depends 和 makedepends 兩個數組的包依賴關係。 包和包之間可以互相依存。