

由浅入深说 PKGBUILD 打包



Table of Contents

Contents

- 包到用时方恨少，码非写过不知难
- EASY
 - 获取现有的 PKGBUILD
 - PKGBUILD 的构成要素

- 包的命名
- 包版本号
- 包元数据
- 包间关系

包到用时方恨少，码非写过不知难

即便是 Arch Linux 初学者，也鲜有完全靠官方源中的软件包就能存活的，或多或少都得依赖一些第三方源或者 AUR 中的额外软件包补充可用软件库。最近发现一个非常有意思的项目 [Repology](#)，总结了诸多自由开源软件发行版们软件库中软件包，其中还有一张总结性的图示，横总对比各个软件源的包数量和新旧程度。

Repology 软件源对比图（放大）（来源）



可以在 Repology 的软件源对比图中找找代表 Arch Linux 和 AUR 的两个 Arch 蓝小圆点 ●。Arch Linux 的小圆点 ● 位于斜对角线左侧，AUR 的小圆点 ● 则远居于整张图的最右侧。从中可以看出 Arch Linux 官方源的包数量相对较少（部分是因为粗放打包策略）而更新相对及时，另一边 AUR 的包数量就非常丰富，但更新却不那么及时了。官方软件源提供了一个更新及时并且足够稳定的基础，在此基础上利用 AUR 补充可用软件包，这对于 Arch Linux 用户来说也即是常态。

AUR 不同于二进制软件源的是，它只是一个提供 PKGBUILD 脚本的共享网站。对于二进制软件源而言，可以直接在 `/etc/pacman.conf` 中添加，然后就可安装其中的软件，而对于 AUR 中的软件包，需要自行下载 PKGBUILD 并将之编译成二进制包。由于 AUR 上软件包维护者众多，于是打包质量也参差不齐，难免遇到一些

包需要使用者手动修改 PKGBUILD 才可正常打包。于是对于 AUR 用户而言，**理解并可修改** PKGBUILD 以定制打包过程非常重要。即便使用 AUR Helpers 帮助自动完成打包工作，对 PKGBUILD 的理解也不可或缺。

是可谓：**包到用时方恨少，码非写过不知难**

包到用时方恨少： 很多时候想要用的包不在官方源，去 AUR 一搜虽有，却不知其打包的质量如何，更新的频度又怎样。

码非写过不知难： 于是从 AUR 下载下来的 PKGBUILD，用编辑器打开，却不知从何看起，如何定制。

经常有人让我写写关于 Arch Linux 中打包的经验和技巧，因此本文就旨在由浅入深地梳理一下我个人对 PKGBUILD 打包系统的理解和体会。社区中有如 felixonmars 这样掌管 Arch Linux 软件包近半壁江山的老前辈，有如 lilydjwtg 这样创建出 lilac 自动打包机器人并统领 [archlinuxcn] 软件源的掌门人，也有各编程语言各编译环境专精的行家里手，说起打包的经验实在不敢班门弄斧地宣称涵盖打包细节的方方面面，只能说是我个人的理解和体会。

再者，Arch Linux 的打包系统也是时时刻刻在不断变化、不断发展的，本文发布时所写的内容，在数月乃至数年之后可能不再适用。任何具体的打包细节都请参阅 PKGBUILD 和 makepkg 的相关手册页，以及

archwiki 上相应的 PKGBUILD 。本文的目的仅限于对上述官方文档提供一条易于入门的脉络，不能作为技术性参考或补充。

好，废话码了一屏，尚未见半句干货，就跟我一起从最基础的部分开始吧。

EASY

讲解 PKGBUILD 之前，想先大概看看 PKGBUILD 长什么样子。AUR 上有大量 PKGBUILD 可以下载，Arch Linux 官方源中打包的官方包也有提供公开渠道下载打包时采用的 PKGBUILD ，这些都可以拿来作为参考。于是作为第一步，如何获取 AUR 或者官方源中包的 PKGBUILD 呢？

获取现有的 PKGBUILD

Arch Linux 老用户们已经很熟悉 AUR helpers 和 Arch Build System 那一套了，每个人可能都有两三个自己趁手的常用 AUR helper 自动化打包。不过其实，大概从3年前 AUR web v4 发布开始，已经不需要专用工具，直接用 git 就可以很方便地下载到官方源和 AUR 中的 PKGBUILD 。

对于 Arch Linux 官方源中的软件包，根据它是来自 [core]/[extra] 还是来自 [community] 我们可以用以下方式获取对应的 PKGBUILD：

```
1 # 获取 core/extra 中包名为 glibc 的包，
   写入同名文件夹
2 git clone https://git.archlinux.org/
  svntogit/packages.git/ -b packages/glib
  c --single-branch glibc
3 # 获取 community 中包名为 pdfpc 的包，
   写入同名文件夹
4 git clone https://git.archlinux.org/
  svntogit/community.git/ -b packages/pdf
  pc --single-branch pdfpc
```

对于官方源中的包，以上方式 clone 到的目录结构是这样：

```
1 pdfpc
2 |— repos
3 |   └─ community-x86_64
4 |       └─ PKGBUILD
5 └─ trunk
6     └─ PKGBUILD
```

其中 trunk 文件夹用于时机打包， repos 文件夹则用于跟踪这个包发布在哪些具体仓库中。由于区分仓库状态和架构，以前还在支持 i686 的时候，打出的包可能

位于 `community-testing-i686` 或者 `community-staging-x86_64` 这样的文件夹中。这些细节不需要关心，我们只需要 `trunk` 中的文件就可以打包了。

对于 AUR 中的软件包，可以直接用以下方式获取 PKGBUILD：

```
1 # 获取 AUR 中包名为 pdfpc-git 的包，写入同名文件夹
2 git clone aur@aur.archlinux.org:pdfpc-git.git
```

不同于官方源，AUR 中包没有深层的目录结构，直接在文件夹中放有 PKGBUILD：

```
1 pdfpc-git
2 └─ PKGBUILD
```

为了方便键入，在我的 `bash/zsh` 配置中提供了几
个函数 `Ge Gc Ga` 分别用于获取 `[core]/[extra]`，
`[community]` 或是 AUR 中的 PKGBUILD，需要的可以自己取用，对于 `zsh` 用户还有这些命令的 自动补全包名。
。

PKGBUILD 的构成要素

拿到了 PKGBUILD ，就先用文本编辑器打开它看一眼吧，以 pdfpc 的 PKGBUILD 为例：

```
1 # Maintainer: Jiachen Yang <farseerf
c@archlinux.org>
2
3 pkgname=pdfpc
4 pkgver=4.2.1
5 pkgrel=1
6 pkgdesc='A presenter console with mu
lti-monitor support for PDF files'
7 arch=('x86_64')
8 url='https://pdfpc.github.io/'
9 license=('GPL')
10
11 depends=('gtk3' 'poppler-glib' 'libg
ee' 'gstreamer' 'gst-plugins-base')
12 makedepends=('cmake' 'vala')
13
14 source=("$pkgname-$pkgver.tar.gz::ht
tps://github.com/pdfpc/pdfpc/archive/v$
pkgver.tar.gz")
15 sha256sums=('f67eedf092a9bc275dde312
f3166063a2e88569f030839efc211127245be6d
f8')
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 文件的格式本质上是 bash 脚本，语法遵从 bash 脚本语言，只不过有些预先确定好的内容需要撰写。粗看上面的 PKGBUILD 大体可以分为两半，前半 3~15 行定义了很多变量和数组，后半 17~26 行定义了一些函数。也即是说，PKGBUILD 包含两大块内容：

1. 该包是什么，也即包的元数据(metadata)
2. 当如何打包，也即打包的过程

其中包的元数据又可大体分为三段：

1. 对包的描述性数据。对应上面 3~9 行的内容。这里写这个包叫什么名字，版本是什么，协议用什么……
2. 这个包与其它包的关系。对应上面 11,12 行。这里写这个包依赖哪些包，提供哪些虚包，位于什么包组……
3. 包的源代码位置。对应上面 14,15 行。这里描述这个包从什么地方下载，下载到的文件校验，上游签名……

这些元数据以 `bash` 脚本中定义的 变量(variable) 和 数组(array) 的方式描述。应当定义哪些，每个数据的含义，在手册页和 `PKGBUILD` 都有详尽介绍，下文要具体说明的内容也会相应补充。

随后打包过程则是以确定名称的 `bash` 函数(function) 的形式描述。在函数体内直接书写脚本。一个包至少需要定义一个 `package()` 函数，它用来写「安装」文件的步骤。如果是用编译型语言编写的软件，那么也应该有 `build()` 函数，用来写配置(configure) 和编译的步骤。

`PKGBUILD` 一开始有一行注释以 `Maintainer:` 开头，这里描述这个 `PKGBUILD` 的维护者信息，算是记录对打包贡献，同时也在打包出问题留下联络方式。如果 `PKGBUILD` 经手多人，通常当前的维护者写在 `Maintainer:` 中，其余的贡献者写作 `Contributor:`。这些信息虽然在 AUR 网页界面中也有所记录，不过留下注释也可算作补充。

关于自定义变量

`PKGBUILD` 中定义的 `bash` 变量和函数是导出给 `makepkg` 负责读取的，于是这些变量名和函数名的具体含义有所规定，不能随便乱写。不过如果有重复定义的内容，那么还是可以自定义变量。通常自定义变量名函数名会以下划线(`_`) 开头，以和 `makepkg` 需要的变量名函数名区分。

例如， `pkgname` 需要加前缀后缀的情况下，通常常见的是定义一个 `_pkgname` 作为项目上游的名称，然后让 `pkgname=${_pkgname}-git` 。再如，上游

包的命名

第3行 `pkgname` 定义了包的名字，这个变量的值应当和 AUR 上提交的软件包相同，也应尽量符合上游对项目的命名。定义包名同时也应尽量符合 Arch Linux 中现有软件包的命名方式，并且在 AUR 上提交的软件包名还有些额外约定俗成的规则：

- 如果是编译自版本控制系统(VCS, Version Control System)中检出的最新源代码，应该在上游项目名后添加 `-vcs` 后缀。比如由 `git clone` 得到的 GitHub 上寄宿的上游软件通常会有 `-git` 这样的后缀。
- 如果是对现有二进制做重新打包，应该在上游项目名后添加 `-bin` 后缀。比如上游发布了用于 Debian 系统的二进制包，想要重新打包成可用于 Arch Linux 的包，则要加 `-bin` 后缀。
- 对于特定语言需要的库，通常会有语言名作为前缀。不过这个规则的特例是，如果这个库同时也在 `/usr/bin` 中提供可执行的命令，那么包名可以没

有前缀，或者对包进行拆包，把库和可执行命令分列在不同的包里。一个例子是 powerline 包提供可执行程序，而它依赖的 python-powerline 则提供 python 的库。

- 对于 Arch Linux 官方源中已经有的软件包，如果想稍作修改之后将修改版共享在 AUR，那么通常 AUR 上的包名会是在官方源中对应包的包名，加上简短的单词描述所做的修改。比如 telegram-desktop-systemqt-notoemoji 就是对官方源中 telegram-desktop 基础上换用 NotoEmoji 的修改。并且实际上官方源的 telegram-desktop 曾经在 AUR 中叫 telegram-desktop-systemqt，因为有来自 Debian 的 SystemQt 补丁。在被移入官方源之后去掉了 `-systemqt` 后缀。

一些有趣的包名字符统计

```
1 $ # 三个官方源总体包数量
2 $ pacman -Slq core extra communi
ty | wc -l
3 10225
4 $ # 除了小写字母、数字、短横、点之外有
别的字符的包名数量
5 $ pacman -Ssq | grep "[^-a-z0-9.
]" -c
6 192
7 $ # 除了小写字母、数字、短横、点、下划
线、加号之外有别的字符的包名数量
8 $ pacman -Ssq | grep "[^-a-z0-9.
_+]" -c
9 4
```

另外关于包名中可以使用的字符，在
PKGBUILD#pkgname 有说明可以用：

1. 英文大小写字母
2. 数字
3. 这些符号： @.__+ -

一般来说，包名的字符会符合 Arch Linux 官方源中现有的包名的命名风格。绝大多数包名是 **纯小写字母** 加上 **数字或者点(.)**，单词之间用短横(-) 分隔。另外还有少数包名中出现大写字母或者下划线分隔，或者 C++ 相关的包名中出现加号(+).

对包命名的基本原则是好记好搜，如果知道上游项目的名字，应该能很方便地搜到包对应的名字。不那么好搜的比如如果上游项目叫 PyQt 而 pkgname 叫 python-qt 那么会让搜索更加困难，所以请不要这样命名。

另外关于包名中带的版本号或者数字，除了个别情况之外一般而言 Arch Linux 打包不会给包名本身写上版本。一种特例是当某个上游库发布了新版本，一部分依赖该库的程序还没有兼容新版，这时通常的做法是把老版本的库的包名后面加上版本号，和新版区分，然后让还没有兼容的程序依赖带版本包名的老版，其余依赖新版。比如官方源中的 lua 和 lua51 就是这样的关系。不过这种做法只是过渡，长期来看大部分包名中都不会有版本数字。

包版本号

版本号由3个变量描述 epoch , pkgver , pkgrel 。由 pacman 显示时，这三者显示为 epoch:pkgver-pkgrel 这种样子。比如 toxcore 的版本号是 1:0.2.8-1 的话，也即是说 epoch=1 , pkgver=0.2.8 , pkgrel=1 。这三者中最重要的是 pkgver ，这与上游的版本号相对应，前后的 epoch 和 pkgrel 则是下游打包时指定的。

其中特殊而比较少见的是 `epoch` ，默认不写时值是 0 ，这主要是用来应对上游改变了版本号命名方式的时候用。比如一开始上游用日期 20190101 这样的版本号，后来转用 1.0 发布了新版，让 `pacman` 的 `vercmp` 判断的话 $20190101 > 1.0$ 会认为 1.0 更老。这时候就需要加上 `epoch=1` 从而 $20190101 < 1:1.0$ 。增加 `epoch` 需要相对谨慎，因为一旦加上去，就很难再减下来了。根据 `man PKGBUILD` ，`epoch` 的值应该是一个自然数。

版本号主要部分 `pkgver` 来自上游，也就是说同样一份源代码，如果打几次不同的包，应该有相同的 `pkgver` 。`pkgver` 不由打包者定义，于是可以用的字符比较自由，不过一般是点隔开的数字的形式，可能会有字母。数字和字母混杂的版本号之间如何比较大小，在 `pacman` 有定义，可以参考 `man vercmp` 。值得注意 `pacman` 的定义可能和上游社区比如 PyPI 之类的地方的定义有所不同。

最后 `pkgrel` 是同一套源码再次打包时递增的发行版本号，通常是个正整数，可选得可以有小数。

包元数据

上面例子中 6 至 9 行定义了一些元数据。

pkgdesc: 通常来自上游项目的一行描述，会在

pacman -Ss 时显示，其中出现的词也会作为搜索关键词。

arch: 是计算机架构的一个数组。常见 arch= ('any') 表示架构无关的包， arch= ('x86_64') 表示 x86_64 架构下的二进制包。这个后文讲述拆包时详述。

url: 上游项目的 URL 地址，会在 Arch Linux 包列表的界面上显示一个链接。

license: 采用的开源协议。协议名不能随便写，常用协议的协议名在 /usr/share/licenses/common/ 有个完整列表，不在其中的协议就被认为是不常用的协议。虽然 MIT 和 BSD 这些开源协议很常见，但是不算在常用协议中。不常用的协议 **要求** 在打包时同时安装协议文件到 /usr/share/licenses/<pkgname>/ 目录中去。

包间关系

上面例子中 11 和 12 行定义了 depends 和 makedepends 两个数组的包依赖关系。包和包之间可以互相依存。

