

PacVis: 可视化 pacman 本地数据库



目录

- [我为什么要做 PacVis](#)
- [PacVis的老前辈们](#)
 - [pactree](#)
 - [pacgraph](#)
- [于是就有了 PacVis](#)
- [PacVis 的图例和用法](#)
- [从 PacVis 能了解到的一些事实](#)
 - [依赖层次](#)
 - [循环依赖](#)
 - [有些包没有依赖关系](#)
 - [只看依赖关系的话 Linux 内核完全不重要](#)
 - [pacman -Qtd 不能找到带有循环依赖的孤儿包](#)
- [PacVis 的未来](#)



PacVis

Install Size (-R)

Max Level: 1000 Max Required-By: 10000

pacvis-git

Visualize pacman local database using Vis.js, inspired by pacgraph

INFO DEP 3 REQ-BY 0 OPT-DEP 0

python-tornado pyalpm python-setuptools



我为什么要做 PacVis

我喜欢 Arch Linux，大概是因为唯有 Arch Linux 能给我对整个系统「了如指掌」的感觉。在 Arch Linux 里我能清楚地知道我安装的每一个包，能知道系统里任何一个文件是来自哪个包，以及我为什么要装它。或许对 Debian/Fedora/openSUSE 足够熟悉了之后也能做到这两点，不

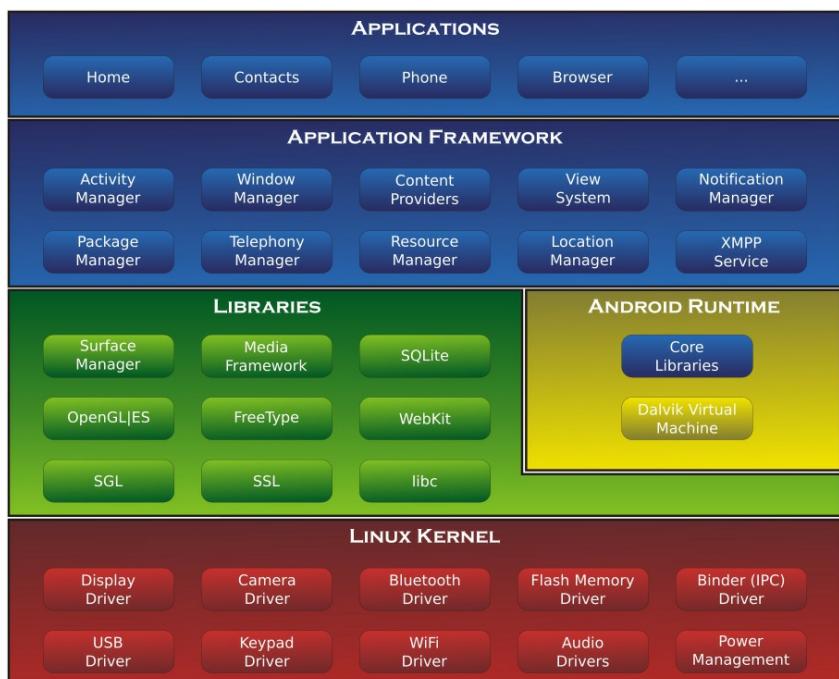
过他们的细致打包的结果通常是包的数量比 Arch 要多个 3 到 10 倍，并且打包的细节也比 Arch Linux 简单的 PKGBUILD 要复杂一个数量级。

每一个装过 Arch Linux 的人大概都知道，装了 Arch Linux 之后得到的系统非常朴素，按照 ArchWiki 上的流程一路走下来的话，最关键的一条命令就是 pacstrap /mnt base ，它在 /mnt 里作为根调用 pacman -S base 装上了整个 base 组，然后就没有然后了。这个系统一开始空无一物，你需要的任何东西都是后来一点点用 pacman 手动装出来的，没有累赘，按你所需。

然而时间长了，系统中难免会有一些包，是你装过用过然后忘记了，然后这些包就堆在系统的角落里，就像家里陈年的老家具，占着地，落着灰。虽然 pacman -Qtd 能方便地帮你找出所有 **曾经作为依赖被装进来，而现在不被任何包依赖** 的包，但是对于那些你手动指定的包，它就无能为力了。

于是我就一直在找一个工具能帮我梳理系统中包的关系，方便我：

1. 找出那些曾经用过而现在不需要的包
2. 找出那些体积大而且占地方的包
3. 厘清系统中安装了的包之间的关系



Android 系统架构

关于最后一点「厘清包的关系」，我曾经看到过 [macOS 系统架构图](#) 和 Android 的系统架构图，对其中的层次化架构印象深刻，之后就一直在想，是否能画出现代 Linux 桌面系统上类似的架构图呢？又或者 Linux 桌面系统是否会展现完全不同的样貌？从维基百科或者别的渠道能找到 Linux 内核、或者 Linux 图形栈，或者某个桌面环境的架构，但是没有找到覆盖一整个发行版的样貌的。于是我便想，能不能从包的依赖关系中自动生成这样一张图呢。

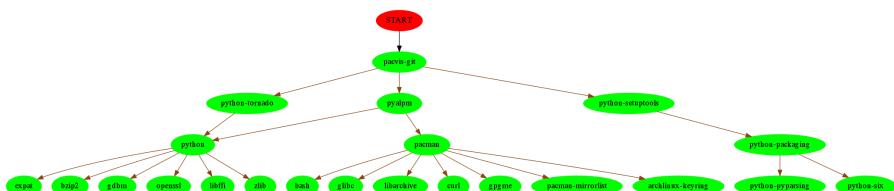
PacVis的老前辈们

在开始写 PacVis 之前，我试过一些类似的工具，他们都或多或少能解决一部分我的需要，又在某些方面有所不足。这些工具成为了 PacVis 的雏形，启发了 PacVis 应该做成什么样子。

pactree

pactree 曾经是一个独立的项目，现在则是 pacman 的一部分了。从手册页可以看出，pactree 的输出是由某个包开始的依赖树。加上 --graph 参数之后 pactree 还能输出 dot 格式的矢量图描述，然后可以用 dot 画出依赖图：

```
pactree pacvis-git -d3 --graph | dot -Tpng >pacvis-pactree.png
```



```
1 $ pactree pacvis-git -d3
2 pacvis-git
3   └── python-tornado
4     └── python
5       ├── expat
6       ├── bzip2
7       ├── gdbm
8       ├── openssl
9       ├── libffi
10      └── zlib
11    └── pyalpm
12      └── python
13    └── pacman
14      ├── bash
15      ├── glibc
16      ├── libarchive
17      ├── curl
18      ├── gpgme
19      ├── pacman-mirrorlist
20      └── archlinux-keyring
21    └── python-setuptools
22      └── python-packaging
23      └── python-pyparsing
24      └── python-six
25 $ pactree pacvis-git -d3 --graph | dot -Tpng >pacvis-pactree.png
```

从画出的图可以看出，因为有共用的依赖，所以从一个包开始的依赖关系已经不再是一棵 图论意义上的树(Tree) 了。最初尝试做 PacVis 的早期实现的时候，就是试图用 bash/python 脚本解析 pactree 和 pacman 的输出，在 pactree 的基础上把整个系统中所有安装的包全都包含到一张图里。当然后来画出的结果并不那么理想，首先由于图非常巨大，导致 dot 的自动布局要耗费数小时，最后画出的图也过于巨大基本上没法看。

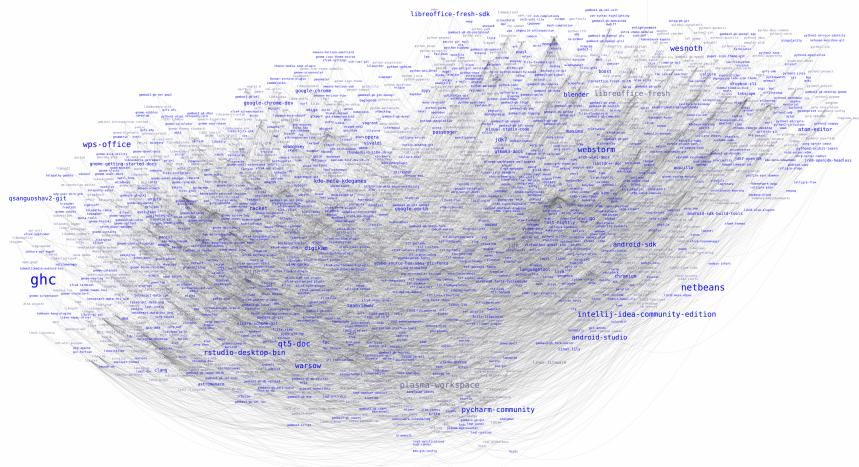
然而不得不说没有 pactree 就不会有 PacVis，甚至 pacman 被分离出 alpm 库也和 pactree 用 C 重写的过程有很大关系，而 PacVis 用来查询 pacman 数据库的库 pyalpm 正是 alpm 的 Python 绑定。因为 pactree 的需要而增加出的 alpm 库奠定了 PacVis 实现的基石。

pacgraph

pacgraph 的输出

29726MB

android-ndk



pacgraph 是一位 Arch Linux 的 Trusted User keenerd 写的程序，和 PacVis 一样也是用 Python 实现的。比起 pactree，pacgraph 明显更接近我的需求，它默认绘制整个系统的所有安装包，并且用聪明的布局算法解决了 dot 布局的性能问题。

pacgraph 的输出是一个富有艺术感的依赖图，图中用不同的字体大小表示出了每个包占用的磁盘空间。通过观察 pacgraph 的输出，我们可以清楚地把握系统全局的样貌，比如一眼看出这是个桌面系统还是个服务器系统，并且可以很容易地发现那些占用磁盘空间巨大的包，考虑清理这些包以节约空间。

更棒的是 pacgraph 还提供了一个交互性的 GUI 叫做 pacgraph-tk，显然通过 tk 实现。用这个 GUI 可以缩放观察整幅图的细节，或者选中某个包观察它和别的包的依赖关系。

pacgraph 还支持通过参数指定只绘制个别包的依赖关系，就像

pactree 那样。

不过 pacgraph 也不是完全满足我的需要。如我前面说过，我希望绘制出的图能反应 **这个发行版的架构面貌**，而 pacgraph 似乎并不区别「该包依赖的包」和「依赖该包的包」这两种截然相反的依赖关系。换句话说 pacgraph 画出的是一张无向图，而我更想要一张有向图，或者说是 **有层次结构的依赖关系图**。

于是就有了 PacVis

PacVis 刚打开的样子



总结了老前辈们的优势与不足，我便开始利用空余时间做我心目中的 PacVis。前后断断续续写了两个月，又分为两个阶段，第一阶段做了基本的功能和雏形，第二阶段套用上 <https://getmdl.io/> 的模板，总算有了能拿得出手给别人看的样子。

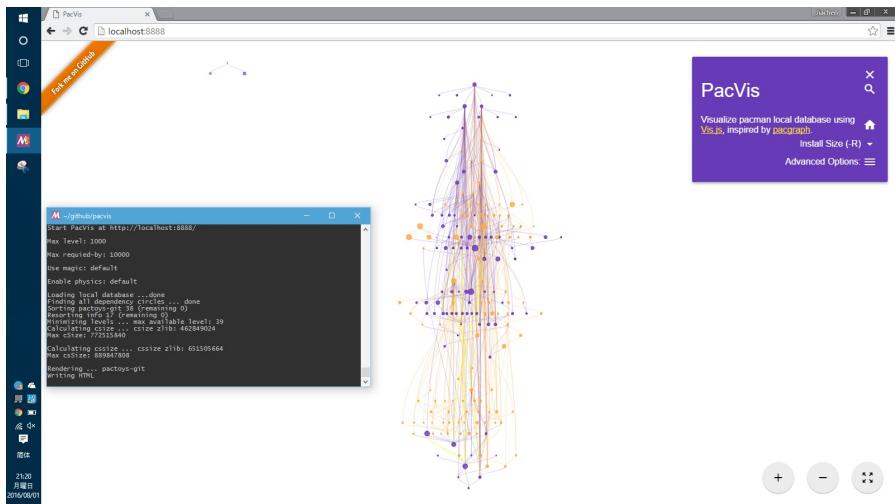
于是乎前两天在 AUR 上给 pacvis 打了个 pacvis-git 包，现在想在本

地跑 pacvis 应该很方便了，用任何你熟悉的 aurhelper 就可以安装，也可以直接从 aur 下载 PKGBUILD 打包：

- 1 ~\$ git clone aur@aur.archlinux.org:pacvis-git.git
- 2 ~\$ cd pacvis-git
- 3 ~/pacvis-git\$ makepkg -si
- 4 ~/pacvis-git\$ pacvis
- 5 Start PacVis at <http://localhost:8888/>

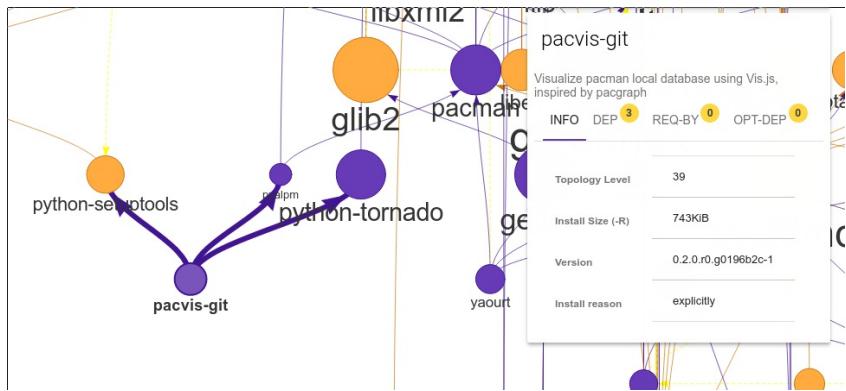
按照提示说的，接下来打开浏览器访问 <http://localhost:8888/> 就能看到 PacVis 的样子了。仅仅作为尝试也可以直接打开跑在我的服务器上的 demo: <https://pacvis.farseerfc.me/>，这个作为最小安装的服务器载入速度大概比普通的桌面系统快一点。

在 Windows msys2 跑 PacVis



另外补充一下，因为 PacVis 只依赖 pyalpm 和 tornado，所以在别的基于 pacman 的系统上跑它应该也没有任何问题，包括 Windows 上的 msys2 里（尽管在 msys2 上编译 tornado 的包可能要花些功夫）。

操作上 PacVis 仿照地图程序比如 Google Maps 的用法，可以用滚轮或者触摸屏的手势 缩放、拖拽，右上角有个侧边栏，不需要的话可以点叉隐藏掉，右下角有缩放的按钮和回到全局视图的按钮，用起来应该还算直观。



pacvis-git 包的依赖

先解释图形本身，整张图由很多小圆圈的节点，以及节点之间的箭头组成。一个圆圈就代表一个软件包，而一条箭头代表一个依赖关系。缩放到细节的话，能看到每个小圆圈的下方标注了这个软件包的名字，鼠标悬浮在圆圈上也会显示响应信息。还可以点开软件包，在右侧的边栏里会有更详细的信息。

比如图例中显示了 pacvis-git 自己的依赖，它依赖 pyalpm, python-tornado 和 python-setuptools，其中 pyalpm 又依赖 pacman。图中用 **紫色** 表示手动安装的包，**橙色** 表示被作为依赖安装的包，箭头的颜色也随着包的颜色改变。

值得注意的是图中大多数箭头都是由下往上指的，这是因为 PacVis 按照包的依赖关系做了拓扑排序，并且给每个包赋予了一个拓扑层级。比如 pacvis-git 位于 39 层，那么它依赖的 pyalpm 就位于 38 层，而 pyalpm 依赖的 pacman 就位于 37 层。根据层级关系排列包是 PacVis 于 pacgraph 之间最大的不同之处。

除了手动缩放，PacVis 还提供了搜索框，根据包名快速定位你感兴趣的包。以及在右侧边栏中的 Dep 和 Req-By 等页中，包的依赖关系也是做成了按钮的形式，可以由此探索包和包之间的关联。

最后稍微解释一下两个和实现相关的参数：

Max Level

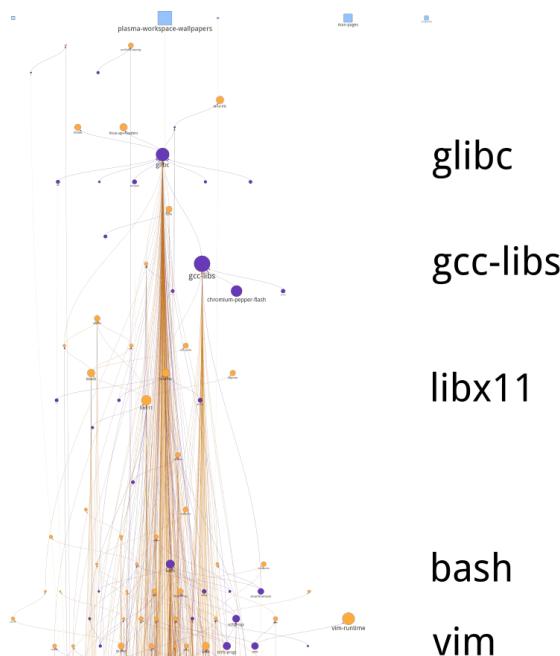
这是限制 PacVis 载入的最大拓扑层。系统包非常多的时候 PacVis 的布局算法会显得很慢，限制层数有助于加快载入，特别是在调试 PacVis 的时候比较有用。

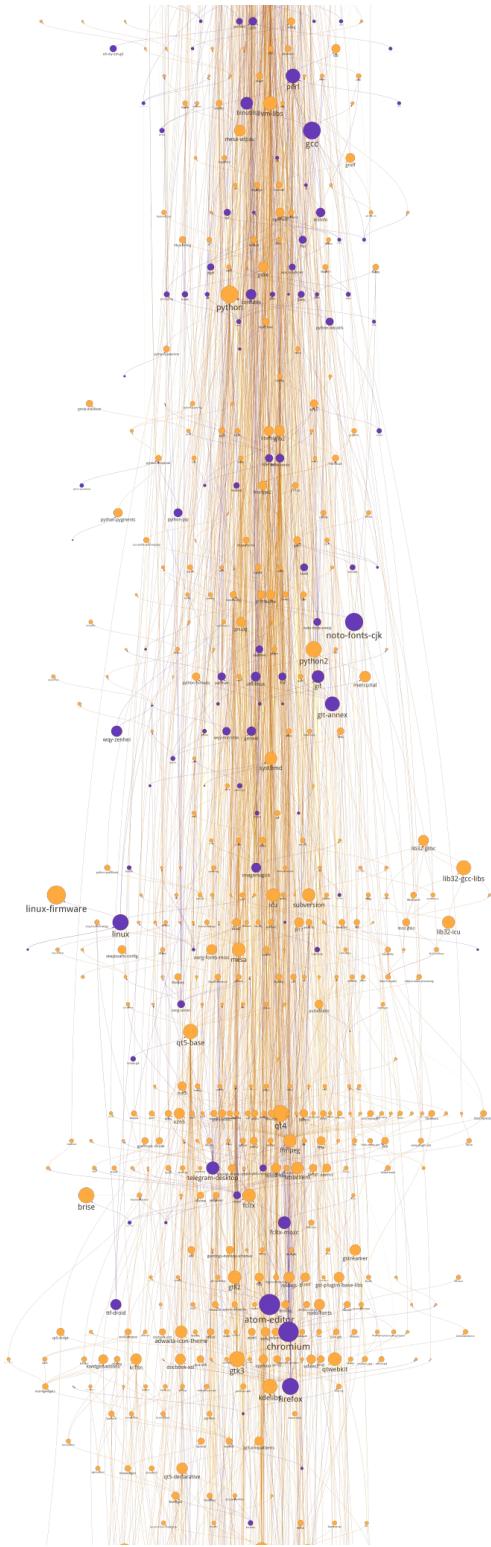
Max Required-By

这是限制 PacVis 绘制的最大被依赖关系。稍微把玩一下 PacVis 就会发现系统内绝大多数的包都直接依赖了 glibc 或者 gcc-libs 等个别的几个包，而要绘制这些依赖的话会导致渲染出的图中有大量长直的依赖线，不便观察。于是可以通过限制这个值，使得 PacVis 不绘制被依赖太多的包的依赖关系，有助于让渲染出的图更易观察。

从 PacVis 能了解到的一些事实

一个 KDE 桌面的 PacVis 结果全图，[放大 \(17M\)](#)





perl
gcc

guile
python coreutils

glib2
freetype2

pacman

systemd

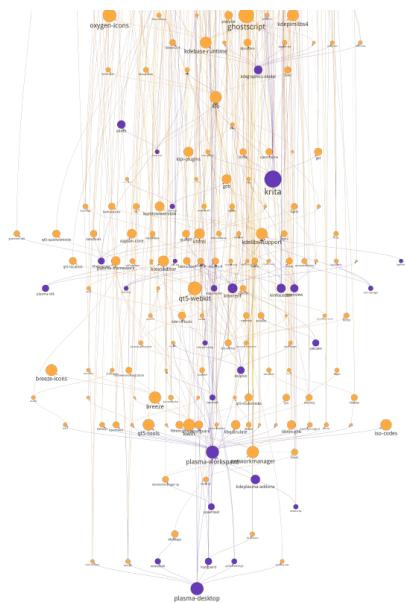
linux

qt5-base

qt4

gtk2
chromium
gtk3
firefox

qt5-declarative



kdegraphics-okular
sddm
krita
plasma-framework

plasma-workspace

plasma-desktop

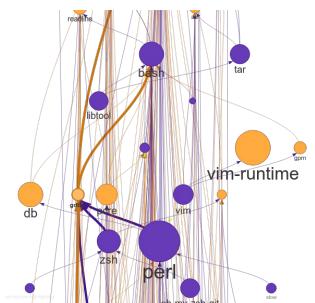
稍微玩一下 PacVis 就能发现不少有趣现象，上述「绝大多数包依赖 glibc」就是一例。除此之外还有不少值得玩味的地方。

依赖层次

系统中安装的包被明显地分成了这样几个层次：

- glibc 等 C 库
- Bash/Perl/Python 等脚本语言
- coreutils/gcc/binutils 等核心工具
- pacman / systemd 等较大的系统工具
- gtk{2,3}/qt{4,5} 等 GUI toolkit
- chromium 等 GUI 应用
- Plasma/Gnome 等桌面环境

大体上符合直观的感受，不过细节上有很多有意思的地方，比如 zsh 因为 gdbm 间接依赖了 bash，这也说明我们不可能在系统中用 zsh 完全替代掉 bash。再比如 python（在 Arch Linux 中是 python3）和 python2 和 pypy 几乎在同一个拓扑层级。

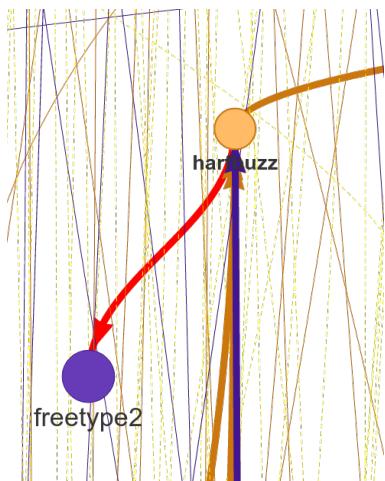


zsh 因为 gdbm 间接依赖了 bash

不过偶尔显示的依赖层级不太符合直观，比如 qt5-base < qt4 < gtk2 < gtk3。qt5 因为被拆成了数个包所以比 qt4 更低级这可以理解，而 gtk 系比 qt 系更高级这一点是很多人（包括我）没有预料到的吧。

循环依赖

有些包的依赖关系形成了循环依赖，一个例子是 freetype2 和 harfbuzz，freetype2 是绘制字体的库，harfbuzz 是解析 OpenType 字形的库，两者对对方互相依赖。另一个例子是 KDE 的 kio 和 kinit，前者提供类似 FUSE 的资源访问抽象层，后者初始化 KDE 桌面环境。



freetype2 和 harfbuzz 之间的循环依赖

因为这些循环依赖的存在，使得 PacVis 在实现时不能直接拓扑排序，我采用环探测 算法找出有向图中所有的环，并且打破这些环，然后再使用拓扑排序。因此我在图中用红色的箭头表示这些会导致环的依赖

关系。

有些包没有依赖关系



man-pages 和 licenses 没有依赖关系

有些包既不被别的包依赖，也不依赖别的包，而是孤立在整张图中，比如 *man-pages* 和 *licenses*。这些包在图中位于最顶端，拓扑层级是 0，我用 **蓝色** 正方形特别绘制它们。

只看依赖关系的话 Linux 内核完全不重要

所有用户空间的程序都依赖着 *glibc*，而 *glibc* 则从定义良好的 *syscall* 调用内核。因此理所当然地，如果只看用户空间的话，*glibc* 和别的 GNU 组件是整个 GNU/Linux 发行版的中心，而 *Linux* 则是位于依赖层次中很深的位置，甚至在我的 demo 服务器上 *Linux* 位于整个图中的最底端，因为它的安装脚本依赖 *mkinitcpio* 而后者依赖了系统中的众多组件。

pacman -Qtd 不能找到带有循环依赖的孤儿包



msys2 中带有循环依赖的孤儿包

这是我在 *msys2* 上测试 *PacVis* 的时候发现的，我看到在渲染的图中有一片群岛，没有连上任何手动安装的包。这种情况很不正常，因为我一直在我的所有系统中跑 *pacman -Qtd* 找出孤儿包并删掉他们。放大之

后我发现这些包中有一条循环依赖，这说明 pacman -Qtd 不能像语言的垃圾回收机制那样找出有循环依赖的孤儿包。

PacVis 的未来

目前的 PacVis 基本上是我最初开始做的时候设想的样子，随着开发逐渐又增加了不少功能。一些是迫于布局算法的性能而增加的（比如限制层数）。

今后准备再加入以下这些特性：

1. 更合理的 optdeps 处理。目前只是把 optdeps 关系在图上画出来了。
2. 更合理的 **依赖关系抉择**。有时候包的依赖关系并不是直接根据包名，而是 provides 由一个包提供另一个包的依赖。目前 PacVis 用 alpm 提供的方式抉择这种依赖，于是这种关系并没有记录在图上。
3. 目前的层级关系没有考虑包所在的仓库 (core/extr/community/...) 或者包所属的组。加入这些关系能更清晰地表达依赖层次。
4. 目前没有办法只显示一部分包的关系。以后准备加入像 pactree/pacgraph 一样显示部分包。

如果你希望 PacVis 出现某些有趣的用法和功能，也 请给我提 issue。

◦