**Python**记录文档

引言

我对python并不抱有好感，不过反感归反感，该用的地方还是得用的，笑。本文档主要对python的安装以及一些模块的简单使用做一些记录。

目录

[Python记录文档 I](#_Toc25697)

[引言 I](#_Toc3644)

[目录 II](#_Toc30227)

[1. 环境 1](#_Toc10504)

[2. python安装 2](#_Toc16553)

[2.1 前置说明 2](#_Toc22652)

[2.2 libffi安装 3](#_Toc25682)

[2.3 openssl安装 3](#_Toc14477)

[2.4 python安装 3](#_Toc16276)

[2.5 环境变量添加 4](#_Toc7822)

[2.6 安装完成 4](#_Toc26598)

[3. 模块安装 5](#_Toc20922)

[3.1 网络超时 5](#_Toc6294)

[3.2 --user参数 5](#_Toc29990)

[4. pyinstaller 6](#_Toc20861)

[4.1 引言 6](#_Toc5650)

[4.2 使用 6](#_Toc27772)

[5. trash-cli 8](#_Toc10736)

[5.1 引言 8](#_Toc26602)

[5.2 使用 8](#_Toc25002)

[5.3 配置 9](#_Toc21841)

[6. setuptools 10](#_Toc9180)

[6.1 引言 10](#_Toc5929)

[6.2 使用 10](#_Toc20004)

[6.2.1 目录层级 10](#_Toc11646)

[6.2.2 setup.py 11](#_Toc7610)

[6.2.3 command 11](#_Toc4253)

[6.2.4 不使用from 12](#_Toc25057)

**1.** 环境

Linux一般自带python，不过版本可能比较低。如果有网甚至有root，那么python的安装会变得格外简单。不过实际工作中，公司不会给开发者提供root权限，一般也不会有外网，最多提供一个pip的代理地址。

如果根目录下的python版本新一些还好，如果只有python2之类的，那么情况会变得非常蛋疼。这时候将新版本的python安装到主目录成了唯一的办法。

**2. python**安装

**2.1** 前置说明

下载python源码后直接安装其实是可以直接通过的，但是执行make时可能会有一些提示，一般会提示ssl、tkinter、zlib等模块未找到。其中值得注意的就是ssl和zlib这两个模块。比如如果没有ssl模块，终端会提示：

Could not build the ssl module!

Python requires a OpenSSL 1.1.1 or newer

SSL是一种加密协议，python中经常会使用到SSL加密以确保安全通信。比如使用pip安装，可能需要通过HTTPS连接到PyPI，此时就需要SSL。

其实我至今不太理解为什么通过源码安装python时需要额外安装一次ssl，通过源码安装python是附带pip的，然后pip经常会使用到ssl，但ssl又需要额外安装。至少看起来多少有些不合理。截止到python3.12.2，我都遇到了这种情况，希望未来安装时可以再简单一些，不用这么折腾吧。

在python3.10之后，libressl将不再被支持，所以这里使用openssl提供ssl模块。安装openssl前，需要先安装libffi。

综上，安装前，需要至少先获取libffi、openssl、python三份源码。

libffi官方地址如下，这里我使用的是libffi-3.4.5版本：

<https://sourceware.org/libffi/>

<https://github.com/libffi/libffi/releases/download/v3.4.5/libffi-3.4.5.tar.gz>

openssl官方地址如下，切记，截止至python-3.12.6，只能使用openssl-1.1.1的版本，我尝试使用openssl-3.3.2，即使安装成功，也会报错无法找到ssl模块。这里我使用的是openssl-1.1.1w版本：

<https://www.openssl.org/source/>

<https://www.openssl.org/source/old/1.1.1/index.html>

<https://www.openssl.org/source/old/1.1.1/openssl-1.1.1w.tar.gz>

zlib我的环境中已经存在了，后续不再赘述安装流程，和其他模块如openssl安装的流程是一样的。zlib官方地址如下，这里我使用的是python-1.3.1版本：

<https://www.zlib.net/>

<https://www.zlib.net/zlib-1.3.1.tar.gz>

python3官方地址如下，这里我使用的是python-3.12.6版本：

<https://www.python.org/ftp/python/>

<https://www.python.org/ftp/python/3.12.6/Python-3.12.6.tar.xz>

**2.2 libffi**安装

libffi最好和python安装到同目录下，否则后续可能需要额外的环境配置，安装步骤如下：

tar -zxvf libffi-3.4.5.tar.gz

cd libffi-3.4.5

./configure --prefix=$HOME/.local/x86/python-3.12.6

make -j && make install

**2.3 openssl**安装

openssl最好不要和python安装到同目录下，否则安装可能无法成功，当然也可以尝试修改python源码中的Module/Setup等方式解决，这里不多赘述。注意，Linux下，运行的时config文件而不是Configure。安装步骤如下：

tar -zxvf openssl-1.1.1w.tar.gz

cd openssl-1.1.1w

./config --prefix=$HOME/.local/x86/openssl-1.1.1w

make -j && make install

**2.4 python**安装

每个版本的python安装识别的参数可能都有些不一样，比如这个--with-openssl参数，我也有些混乱，保险起见，环境变量和参数都加上比较好。python安装步骤如下：

tar -zxvf Python-3.12.6.tar.xz

cd Python-3.12.6

export PKG\_CONFIG\_PATH=\

$HOME/.local/x86/python-3.12.6/lib/pkgconfig:$PKG\_CONFIG\_PATH

export PKG\_CONFIG\_PATH=\

$HOME/.local/x86/openssl-1.1.1w/lib/pkgconfig:$PKG\_CONFIG\_PATH

export LD\_LIBRARY\_PATH=\

$HOME/.local/x86/python-3.12.6/lib:$LD\_LIBRARY\_PATH

export LD\_LIBRARY\_PATH=\

$HOME/.local/x86/openssl-1.1.1w/lib:$LD\_LIBRARY\_PATH

export C\_INCLUDE\_PATH=\

$HOME/.local/x86/python-3.12.6/include:$C\_INCLUDE\_PATH

./configure --prefix=$HOME/.local/x86/python-3.12.6 \

--with-openssl=$HOME/.local/x86/openssl-1.1.1w \

--enable-shared

make -j && make install

**2.5** 环境变量添加

因为python库和可执行文件安装在$HOME下，所以添加环境变量是必要的，步骤如下：

vim ~/.bash\_profile

export PATH=$HOME/.local/x86/python-3.12.6/bin:$PATH

export LD\_LIBRARY\_PATH=\

$HOME/.local/x86/python-3.12.6/lib:\

$HOME/.local/x86/openssl-1.1.1w/lib:$LD\_LIBRARY\_PATH

source ~/.bash\_profile

**2.6** 安装完成

一般完成上述流程python的安装就完成了。使用如下命令查看python是否安装成功。

which python3

python3 --version

最好再通过pip安装一些模块检验ssl是否存在问题。

**3.** 模块安装

**3.1** 网络超时

使用pip安装模块时，可能会遇到socket.timeout报错；也可能是hash值不匹配报错：

ERROR: THESE PACKAGES DO NOT MATCH THE HASHES FROM THE REQUIREMENTS FILE。

在pip命令后添加--default-timeout和--upgrade参数，一般可以解决此问题：

pip3 --default-timeout=6666 install --upgrade paramiko

**3.2 --user**参数

在pip命令后添加--user参数，默认情况下，模块会被安装在~/.local/目录中，包括bin、lib等等。

在pip命令后不添加--user参数，默认情况下，模块会被安装在python-3.12.6目录下，模块的bin目录和python-3.12.6/bin目录共用；模块的lib安装在python-3.12.6/lib/python3.12/site-packages下。

**4. pyinstaller**

**4.1** 引言

python直接运行源文件实际上已经非常方便了，那么为什么还要用第三方工具将源代码进行打包呢？原因是多方面的。比如出于对源代码的保密，要求仅提供一个可执行的包；又或者当我的某个脚本需要在其它环境中运行，别的环境中没有我需要python配置。

因此，一些对python源码打包的工具应运而生，比如pyinstaller、cxfreeze等等。这里以pyinstaller为例。通过以下命令安装pyinstaller：

pip install pyinstaller

**4.2** 使用

pyinstaller官方使用文档地址如下：

<https://pyinstaller.org/en/stable/usage.html>

关于使用pyinstaller命令时的参数，实际上官方文档已经说明的非常清楚了，这里记录一些常用的参数选项。

(1) --distpath <dist\_dir>

指定打包后程序的存放目录，不指定默认为./dist

(2) --clean

pyinstaller在打包时会产生一系列缓存文件，如果希望打包前删除上一次的缓存文件，可以使用此参数。

(3) -F/--onefile

只生成一个可执行文件。不指定的话默认生成一个目录，其中包含很多文件。

(4) -n binary\_file\_name/--name binary\_file\_name

指定可执行文件的文件名，不指定默认使用打包时的第一个脚本文件名。

(5) --add-data=<file\_source\_path>:<file\_packed\_dest\_dir>

添加其它数据文件。例如脚本中需要对./image/img.png图片文件做处理。那么默认情况下img.png文件是不会被打包到程序中的。此时需要按如下方式指定打包参数：

--add-data=./image/img.png:./image/

添加img.png后，打包完成运行时不一定找得到img.png文件，因为脚本代码中可能使用的是相对路径，而实际使用pyinstaller打包过程中，pyinstaller会创建一个临时目录，此时通过--add-data指定的文件将无法被添加。所以，若脚本中要对其它数据文件做处理，尽量使用这些数据文件的绝对路径。

--add-data参数可多次使用。Linux中，file\_source\_path和file\_packed\_dest\_dir之间用符号:分隔；Windows中，file\_source\_path和file\_packed\_dest\_dir之间用符号;分隔。

(6) --add-binary=<lib\_source\_path>:<lib\_packed\_dest\_dir>

打包时添加二进制文件。一些动态库在打包时可能不会被引用到程序中，此选项可以手动添加二进制文件。它的用法和--add-data类似。lib\_packed\_dest\_dir一般为模块名称。比如添加zbar模块，那么lib\_packed\_dest\_dir一般就指定./zbar。

(7) --icon <ico\_file>

一般用于Windows、MacOs，用于添加可执行文件的图标。

(8) --noconsole

一般用于Windows、MacOs，可执行文件运行时将不会产生黑框窗口。

**5. trash-cli**

**5.1** 引言

试问在Linux下删除文件或目录要怎么做？当然是使用rm命令，于是顺手敲击键盘，输入rm -rf <file\_dir>。一般-r和-f这两个参数足以，脚本中可能加个-v打印删除的文件信息。一开始我甚至不曾了解-r和-f的意思，只知道要删除目录，那就加一个-r。

直到有一天，我的某个目录被我不小心删没了，还没有git。输入删除命令后按下回车那一刹那我拼命地敲击ctrl+c，然而为时已晚。后续经历的风雨就不想多回忆了。实际上-f表示--force，即强制删除；-r表示--recursive，即递归删除。加上这两个参数，尤其是-f，表示删除文件的保险没有了，删除任何文件时，只要有权限，都直接删除，不会做出提示。那时候有人告诉我在rm命令后加上-i参数会保险一些，-i表示--interactive，即删除时向用户确认。不过在此之前我都没有听说过-i这个参数，估计以后也经常懒得用。

事后一开始我是想自己搞一个类似Windows的回收站的，不过实际想来并不轻松，文件重名、文件的恢复等都还是比较麻烦的。不过为这种东西头疼的肯定不止我一个，于是trash-cli工具应运而生。

这是一款开源的文件删除工具，官方地址如下：

<https://github.com/andreafrancia/trash-cli>

这款工具很好地解决了Linux下，执行rm命令后，文件一去不复返的问题。trash-cli是用python开发的，绝大多数的Linux发行版都提供这款软件，如果没有root权限问题也不大，通过以下命令安装trash-cli：

pip install trash-cli

**5.2** 使用

trash-cli的基本使用也非常简单，常用命令如下：

(1) trash-put

将文件或目录放入回收站。

(2) trash-empty

清空回收站，终端会询问是否清空。

(3) trash-list

列出回收站中的文件或目录。

(4) trash-restore

恢复回收站中的文件或目录。

(5) trash-rm

删除回收站中的目录或文件

一般被删除的文件位于~/.local/share/Trash下，并且不需要担心文件重名问题，trash-cli会自动处理。使用trash-restore恢复文件时，trash-cli会询问具体恢复哪个文件，防止重名文件恢复混淆。恢复后的文件和删除前是完全一样的。

**5.3** 配置

如果每次删除文件都要输入trash-put，那还是不现实的，删文件如此频繁的操作要用这么长的命令，那还是放弃得了。工具开发时，开发人员肯定也意识到了这个问题。所以trash-cli比较友好的地方就在于它的命令参数是完美适配rm命令的，比如rm的-i、-v等参数trash-cli也有。

所以可以在环境初始化中添加如下语句：

alias rm="trash-put"

此时执行rm后，文件便进入了回收站。还有一点是，trash-put删除目录时是不需要加-r参数的，不过即使习惯性的加了也没什么问题。添加以上环境变量后，若想真正的使用rm命令而不是trash-put，可以在rm命令前加上符号\或者使用unalias rm解除对rm命令的修改。此后，时不时运行一下trash-empty命令清空一下回收站就可以了。

值得一提的是，执行脚本或可执行程序等，子进程不会继承父进程的alias，alias是shell的一个命令，并不是一种环境变量。通常子进程会继承父进程的环境变量，但不会继承父进程的alias。所以在shell中执行rm命令，或在Makefile中编写伪目标clean等情况下，还是要多加小心的。

**6. setuptools**

**6.1** 引言

假设在编写python代码时，需要用到一个自定义模块中的函数，若此时模块文件不在当前目录下，那么单纯的import是搜索不到模块路径的。通常的方法是通过以下方式指定模块的搜索路径：

sys.path.append(selfModulePath)

如果这个自定义模块的通用性比好，许多源文件中都会用到它，那每次都通过上述的方式指定搜索路径就比较麻烦了。如果可以的话，像sys、os这些可以直接import似乎更友好一些。

setuptools这个工具就能很好地完成上述要求。它可以自定义创建python模块，并将这些模块像其它模块一样安装到site-packages中。如此，自定义的模块用起来就可以像其它模块一样方便了。

实际上setuptools的功能非常丰富，也被许多其它模块作为依赖。自己使用时，通常了解一些基础功能即可。通过以下命令安装setuptools：

pip install setuptools

**6.2** 使用

setuptools是一款开源工具，github地址如下

<https://github.com/pypa/setuptools>

setuptools官方使用文档地址如下：

<https://setuptools.pypa.io/en/latest/setuptools.html>

**6.2.1** 目录层级

假设打包根目录是$HOME/python/common，这里打包根目录用$pRoot代替。我希望打包一个名为fileModule的模块作用通用模块。

那么首先，需要在$pRoot下创建一个名为fileModule的目录，这个目录的名称不要修改，后续import或者from导入模块时用的就是这个名称。将有关fileModule模块的源文件放入这个目录中，假设我这里有一个源文件fileModule.py，那么我将其放入fileModule目录中。此外，还需要在fileModule目录中创建一个\_\_init\_\_.py文件，这个文件可以是空内容，但必须要有。setuptools会搜索同级目录中，带有\_\_init\_\_.py文件的其它源文件，然后进行打包。若没有\_\_init\_\_.py文件，打包后，会找不到对应的fileModule。

其次，在$pRoot目录下，需要创建一个名为setup.py的打包文件，这个文件中指定的是打包的一些信息。综上，打包根目录的层级如下：

$pRoot

├── fileModule

│   ├── fileModule.py

│   └── \_\_init\_\_.py

└── setup.py

**6.2.2 setup.py**

setup.py文件内容如下：

import setuptools

setuptools.setup(

name="fileModule",

version="1.0",

description="File processing",

author="Hu\_Yihua\_UsadaYu",

author\_email="UsadaYu.yh@gmail.com",

packages=setuptools.find\_packages("./"),

)

(1) name：指定包名称，这个名称是任意，不同于上述的fileModule目录名称。不过最好将其和上述的目录名称设置为一样。

(2) version：指定版本号，pip install --upgrade一般根据这个参数寻找目标版本。

(3) packages：指定源文件搜索路径，在此基础上寻找上述的\_\_init\_\_.py文件。

**6.2.3 command**

(1) python3 setup.py sdist

上述命令会根据setup.py的内容以sdist的形式进行打包，Linux下，打包后生成的一版是一个tar.gz形式的压缩文件。此时打包根目录的层级如下：

$pRoot

├── dist

│   └── fileModule-1.0.tar.gz

├── fileModule

│   ├── fileModule.py

│   └── \_\_init\_\_.py

├── fileModule.egg-info

│   ├── dependency\_links.txt

│   ├── PKG-INFO

│   ├── SOURCES.txt

│   └── top\_level.txt

└── setup.py

(2) pip install dist/fileModule-1.0.tar.gz

运行上述命令可将fileModule模块安装到site-packages中，此时site-packages下一般会多出两个目录，为：fileModule和fileModule-1.0.dist-info。可以看出这个很多其它模块的形式一样了。此时用pip list可以查看到fileModule已被安装；使用pip show fileModule也可以看到setup.py中指定的一些打包信息。

(3) from fileModule import fileModule

表示从fileModule目录中导入fileModule模块，第二个fileModule对应的是fileModule.py。也就是说site-packages为第一级搜索目录。

**6.2.4** 不使用**from**

上述在使用fileModule模块时，相比import os这种直接导入，多了一段from。如果希望可以直接import fileModule，那么在打包前的\_\_init\_\_.py文件中添加如下内容即可：

from .fileModule import \*

上述的符号.不要遗漏，这段内容在打包后的site-packages中的\_\_init\_\_.py文件里添加也是有用的。