Package Tutorials

杨资璋（翻译）

2021 年 10 月 9 日

1 对于 Python 中的 Packaging 的概述

作为一个宽泛目的的编程语言，Python 被设计为通过多种方式使用。你可以创建网站或者产业机器人或者你朋友玩游戏，所有都使用相同的核心技术。

由于 Python 的灵活性，所以所有 Python 项目第一步要思考的是项目的受众和项目对应的运行环境。在写代码之前思考 packing 似乎是奇怪的，但是这个过程对于避免以后的头痛有奇效。

这篇概述提供了一个解释 Python 过多 packing 选项的宽泛意义决定树。继续阅读来为你的下一个项目选择最好的技术。

1.1 考虑部署

packages 之所以存在是为了被安装（或者部署）的，所以在你 package 任何东西之前，你将会思考以下问题的答案：

- 你的软件用户是谁？你的软件会被其他开发者用来开发软件、被数据中心的操作人员安装或者会被不怎么知道软件的组安装吗？
- 你的软件是准备在服务器上运行还是桌面、移动客户端（手机、平板等等）或者是嵌入在专用机器上？
- 你的软件是被个体安装还是被大规模安装？

packaging 完全就是目标环境和开发经验。上述问题有很多答案，每一个组合情形都有它对应的解决办法。有了这些信息，接下来的概述将会指导你找到最适合你的项目的 packaging 技术。

1.2 打包 Python 库和工具

你可能听说过 PyPI，setup.py 和 wheel 文件。这只是 Python 生态提供给开发者分发 Python 代码的一小部分工具，为了了解这些，你可以阅读打包和分发项目。

接下来打包的方法是为在开发设定下的技术受众使用的库和工具准备的。如果你正在寻找为了非技术受众或者产品设定的 Python 打包方法，跳至打包 Python 应用。

1.2.1 Python 模块

只依赖标准库的可重分发、重使用的 Python 文件。你还需要确保它是为了正确的 Python 版本写的，并仅仅依赖标准库。

如果想要分享简单脚本和代码段的双方有兼容的 Python 版本（例如通过电子邮件，StackOverflow 或者 GitHub gists），这是极好的。甚至有一些完整的库将这提供为一个选项，例如 bottle.py 和 boltons。

然而，这种模式不能扩展至包含多个文件、需要额外的库或者需要特定 Python 版本的项目，因此有了以下选项。

1.2.2 Python 源码分发

如果你的代码包含多个 Python 文件，它通常会被组织为一个目录结构。任何一个包含 Python 文件的目录可以构成一个导入包。

由于包包含多个文件，所以分发它们更加困难。许多协议支持每次传输一个文件（上次你点击一个下载多个文件的链接是什么时候?）。它更容易得到不完整的传输并且更难保证分发的代码完整性。

只要你的代码只包含 Python 代码，并且你知道你的部署环境支持的 Python 版本，那么你就可以使用 Python 的原生打包工具来创建一个 source Distribution Package，简写为 sdist。

Python 的 sdist 是一个包含一个或多个包或模块的压缩档案（实际上是.tar.gz 文件）。如果你的代码是纯 Python 的，并且你只依赖于其他的 Python 包，你可以在这里学到更多。

如果你依赖于一些非 Python 代码，或者非 Python 包（例如 lxml 中的 libxml2，或者 numpy 中的 BLAS 库），你将会需要接下来章节详述的格式，它对于纯 Python 库也有一些好处。

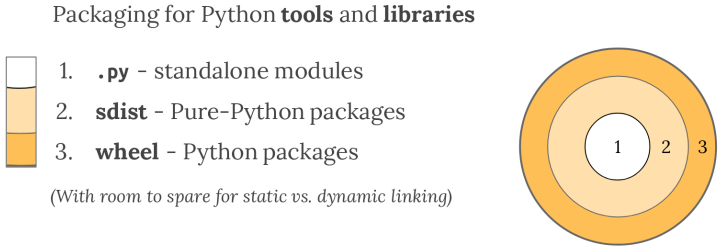
1.2.3 Python 二进制分发

Python 的许多实际能力来自它可以整合软件生态的能力，特别是用 C、C++、Fortran、Rust 和其他语言写的库。

不是所有的开发者都有正确的工具或经验来构建用这些编译好的语言写得组件，因此 Python 创建了 Wheel——一种被设计用来运载有编译好文物的库的包格式。实际上，Python 的包安装器，pip，一直更喜欢 wheel，因为安装更快。所以即便是纯 Python 包也和 wheel 工作的更好。

当分发须需要与源分发匹配时，二进制分发是最好的。即使你没有为每一个操作系统上传代码的 wheel，通过上传 sdist，你使得其他平台的用户可以自己构建包。默认情况下会一起发布 sdist 和 wheel，除非你在为一种非常特殊的使用场景——你知道你的接受者只需要某一种，创建文物。

Python 和 PyPI 使得一起上传 wheels 和 sdists 很容易。只要按照打包 Python 项目向导就可以了。



1.3 打包 Python 应用

到现在为止，我们仅仅讨论了 Python 的原生分发工具。基于我们的介绍，你应该成功地推断出这些自带方法仅仅着眼于有 Python 并且受众知道如何安装 Python 包的环境。

随着操作系统、设置和使用者的变化，这样的假设尽在目标是开发者受众时是合理的。

Python 的原生打包大部分是为了开发者间分发可复用代码，也就是库创建的。你可以使用例如setuptools entry points的技术在 Python 库打包的基础上背驼**工具**或者针对开发者的基本应用。

库是构建模块，不是完整的应用。对于分发应用，那是一个完全新的世界。

接下来的几个部分根据目标环境的依赖组织应用打包选项，所以你可以选择对于你的项目正确的部分。

1.4 What about...

以上部分只能总结如此了，你可能会好奇一些明显的断层。

1.4.1 操作系统包

正如前面所提及到的，一些操作系统有它们自己的包管理器。如果你十分了解你的目标操作系统，你可以直接依赖与例如 deb 或 RPM 的格式，并且使用那个自建的包管理器来负责安装甚至部署。你甚至可以使用 FPM 从相同的源生成 deb 和 FPMs。

在大多数部署流水线上，操作系统的包管理器只是帮助你理解的一部分。

1.4.2 virtualenv

virtualenvs 已经是多代 Python 开发者不可或缺的工具，但是由于它们被更高层级的工具封装，它们正在退出人们的视野。特别的，对于打包，virtualenvs 在 the dh-virtualenv tool 和 osnap 中被用作原型，这两个都通过一种自洽的方式封装了 virtualenvs。

对于产品部署，不要依赖 python -m pip install 来将包装入虚拟环境，正如人们可能自开发环境中做得。上面的概述都是更好的解决办法。

1.4.3 安全性

你越向下走，更新你的包的组件就越难。所有的东西都联系的越来越紧密。

例如，如果出现了一个内核安全性问题，并且你在部署容器，那么宿主系统的内核可能在不要求构造应用的新代表的情况下更新。如果你不是虚拟机镜像，你将需要一个新的构建。这种动态是否使得某种选择更加安全仍然是一个古老的争论，返回仍未挺停息的问题静态 vs 动态链接。

1.5 总结

Python 中的打包由于困难有一定的声望。这种印象大部分是 Python 的多功能性的副产品。一旦你理解了各种打包策略的自然边界，你开始意识到，变化的景象是 Python 程序员为了使用最平衡之一的灵活语言的小代价。

2 安装包

这个部分包含如何安装 Python 包的基本知识。

很重要的一点是注意这个语境下术语 “包” 是用来描述一捆要安装的软件（也就是，分发的同义词）。它不代表那种你在 Python 源代码中导入的包（也就是，一些模块的容器）。在 Python 社区中，使用术语 “包” 来代表分发是很常见的。由于术语 “分发” 可能会与 Linux 版本或者其他大一些的软件版本例如 Python 自身混淆，所以使用术语 “分发” 通常并不被青睐。

2.1 安装包的要求

这个章节讲述了在安装其他 Python 包之前的步骤。

2.1.1 确保你可以在命令行运行 Python

在你继续之前，确保在你的命令行中有 Python 并且预期的版本是可用的。你可以通过运行如下命令检查：

```
python3 --version
```

你应该得到类似 Python 3.6.3 的输出。如果你没有 Python，请从 python.org 下载最新的 3.x 版本或者参考 Hitchhiker 的 Python 指导中的安装 Python 章节。

2.1.2 确保你可以在命令行运行 pip

额外的，你需要确保 pip 可用。你可以通过云心如下命令检查：

```
python3 -m pip --version
```

如果你从源安装 Python，通过 python.org 的安装器或者通过 Homebrew，你应该已经有 pip。如果你是在 Linux 上并且使用你的操作系统包管

理器来安装，你可能需要单独安装 pip，见通过 Linux 包管理器安装 pip/setuptools/wheel。

如果 pip 没有安装，那么首先尝试从标准库中牵引它：

```
python3 -m ensurepip --default-pip
```

如果这仍然不允许你运行 python -m pip：

- 安全下载 get-pip.py
- 运行 python get-pip.py。这将会下载或者升级 pip。额外的，如果 setuptools 和 wheel 还没有被安装，那么它们也会被安装。

2.1.3 确保 pip，setuptools 和 wheel 是最新版本

尽管 pip 独自便足以从与构建好的二进制档案安装，最新的 setuptools 和 wheel 项目拷贝对于确保你也可以从源档案安装十分有用。

```
python3 -m pip install --upgrade pip setuptools wheel
```

2.1.4 可选择的，创建一个虚拟环境

详细信息见下面的章节，但是这里有一些典型 Linux 系统下的基本 venv 命令：

```
python3 -m venv tutorial_env
source tutorial_env/bin/activate
```

这将会自 tutorial_env 目录中创建一个新的虚拟环境，并将当前 shell 的默认 python 环境设置为它。

2.2 创建虚拟环境

2.3 使用 pip 安装

pip 是推荐的安装器。接下来，我们将会覆盖最常见的使用场景。对于更多细节，见包含完整的参考指导的pip 文档。

2.4 从 PyPI 安装

pip 最常见的用法是从Python 包索引中使用要求指定器安装。大致上来说，一个要求指定器有一个项目名称跟随一个可选的版本号构成。PEP 440 中包含现在支持的限定符的完整规范。接下来是一些例子。

为了安装最新版本的 “SomeProject”

```
python3 -m pip install "SomeProject"
```

为了安装特定的版本:

```
python3 -m pip install "SomeProject==1.4"
```

为了安装大于或等于版本 1 同时小鱼另外一个的版本:

```
python3 -m pip install "SomeProject>=1,<2"
```

为了安装兼容特定版本的版本:

```
python3 -m pip install "SomeProject~=1.4.2"
```

在这种情况下，这表示安装任意">=1.4.2" 的"==1.4.*" 的版本。

2.5 源分发与 Wheels 的比较

pip 可以从 sdist 或者 Wheels 中安装，但是如果二者都存在于 PyPI，pip 将会倾向于兼容的 wheel。你可以通过例如–no-binary 选项来覆盖 pip 的默认行为。

Wheels 是一个与 sdist 相比提供了更快速安装的预链接好的分发格式，特别是当项目包含编译好的扩展。

如果 pip 没有发现 wheel，它将会在本地链接一个 wheel 并为将来的安装存储，而不是在将来重链接 dist。

2.6 更新包

将已经安装的 SomeProject 升级至 PyPI 的最新版本。

```
python3 -m pip install --upgrade SomeProject
```

2.7 安装至用户站点

为了单独为当前的用户安装包，使用–user 标志：

```
python3 –m pip install ––user SomeProject
```

对于更多的信息，将 pip 文档中的用户安装章节。

注意–user 标志在虚拟环境内是无效的——所有的安装安装命令将会影响虚拟环境。

如果 SomeProject 定义了命令行脚本或者控制台入口点，–user 将会导致它们被安装在用户基础的二进制目录，它可能会也可能没有已经在你的 shell 的 PATH 中。（从版本 10 开始，当安装任意脚本到非 PATH 目录时，pip 会显示警告。）如果安装后脚本在你的 shell 中不可用，你需要将目录添加到你的 PATH 中。

- 在 Linux 和 macOS 中你可以通过运行 python -m site –user-base 来找到用户基础二进制目录。例如，这通常会打印 /.local（其中 将扩展为你的 home 目录的绝对路径）所以你需要在 PATH 中添加 /.local/bin。你可以通过修改 /.profile永久修改 PATH。

2.8 要求文件

安装要求文件中指定的要求列表。

```
python3 –m pip install –r requirements.txt
```

2.9 从 VCS 安装

在可编辑模式下从 VCS 安装项目。对于详细的语法，见 pip 的VCS 支持章节。

2.10 从其他索引安装

从一个备选索引下载

```
python3 –m pip install ––index–url https://my.package.repo/simple/ Somel
```

在 PyPI 之外，在安装过程中从一个额外的索引搜索

```
python3 –m pip install ––extra–index–url https://my.package.repo/simple/
```

2.11 从本地 src 树安装

在开发模式下从本地 src 安装，也就是，在这样一种方式下项目似乎被安装了，但是仍然可以从 src 树编辑。

```
python3 -m pip install -e <path>
```

你也可以从 src 正常安装

```
python3 -m pip install <path>
```

2.12 从本地档案安装

安装一个特定的源档案文件

```
python3 -m pip install ./downloads/SomeProject-1.0.4.tar.gz
```

从包含档案的本地目录安装（同时不检查 PyPI）

```
python3 -m pip install --no-index --find-links=file:///local/dir/ SomeP
python3 -m pip install --no-index --find-links=/local/dir/ SomeProject
python3 -m pip install --no-index --find-links=relative/dir/ SomeProject
```

2.13 从其他源安装

为了从其他数据源下载（例如亚马逊 S3 存储），你可以创建一个提供 PEP503 索引格式数据的助手函数，并使用–extra-index-url 标志来告诉 pip 使用这个索引

```
./s3helper --port=7777
python3 -m pip install --extra-index-url http://localhost:7777 SomeProje
```

2.14 下载预发布版本

除了稳定版本，找到预发布和开发者版本。默认的，pip 只寻找稳定版本。

```
python3 -m pip install --pre SomeProject
```

2.15 安装 Setuptools “额外”

安装setuptools extras

```
python3 -m pip install SomePackage[PDF]
python3 -m pip install SomePackage[PDF]==3.0
python3 -m pip install -e .[PDF]  # editable project in current director
```

3 创建文档

这个章节包含如何使用Sphinx创建文档以及将文档免费寄存在Read The Docs的基本内容。

3.1 安装 Sphinx

使用 pip 安装 Sphinx:

```
python3 -m pip install -U sphinx
```

对于其他安装方法，见 Sphinx 的安装指南。

3.2 开始使用 Sphinx

在你的项目下创建一个 docs 目录来存放你的文档:

```
python3 -m pip install -U sphinxcd /path/to/project
mkdir docs
```

在 docs 目录里运行 sphinx-quickstart 命令

```
cd docs
sphinx-quickstart
```

这将会创建一个源目录，带你走过一些基本设置，并创建一个 index.rst 文件和一个 conf.py 文件

你可以在 index.rst 文件中添加关于你的项目的信息，之后链接它们:

```
make html
```

链接过程的更多细节间 Read The Docs 的指南

3.3 其他资源

关于如何使用 Sphinx 和 reStructuredText 的更多详细指南，请查看 Hitchhiker 的 Python 指南的文档教程