上一章里，我们通过 ppw 生成了一个规范的 python 项目，对初学者来说，许多闻所未闻、见所未见的概念和名词扑面而来，不免让人一时眼花缭乱，目不暇接。然而，如果我们不从头讲起，可能读者也无从理解，ppw 为何要应用这些技术，又倒底解决了哪些问题。

在 2021 年 3 月的某个孤独的夜晚，我决定创建一个创建一个 python 项目以打发时间，这个项目有以下文件：

```
├── foo
│   ├── foo
│   │   ├── bar
│   │   │   └── data.py
│   └── README.md
```

当然，作为一个有经验的开发人员，我的机器上已经有了好多个其它的 python 项目，这些项目往往使用不同的 Python 版本，彼此相互冲突。所以，从一开始，我就决定通过虚拟开发环境来隔离这些不同的工程。这一次也不例外：我通过 conda 创建了一个名为 foo 的虚拟环境，并且始终在这个环境下工作。

我们的程序将会访问 postgres 数据库里的 users 表。一般来说，我们都会使用 sqlalchemy 来访问数据库，而避免直接使用特定的数据库驱动。这样做的好处是，万一将来我们需要更换数据库，那么这种迁移带来的工作量将轻松不少。

在 2021 年 3 月，python 的异步 io 已经大放异彩。而 sqlalchemy 依然不支持这一最新特性，这不免让人有些失望——这会导致在进行数据库查询时，python 进程会死等数据库返回结果，从而无法有效利用 CPU 时间。好在有一个名为 Gino 的项目弥补了这一缺陷：

```
$ pip install gino
```

> **Warning**
>
> 在那个孤独的夜晚，上述命令将安装 gino 1.0 版本。如果读者想运行这里的程序，请将 gino 的版本改为 1.0.1，即运行 pip install gino==1.0.1

做完这一切准备工作，开始编写代码，其中 data.py 的内容如下：

```
# 运行以下代码前，请确保本地已安装 POSTGRES 数据库，并且创建了名为 GINO 的数据库。

import asyncio
from gino import Gino

db = Gino()

class User(db.Model):
    __tablename__ = 'users'

    id = db.Column(db.Integer(), primary_key=True)
    nickname = db.Column(db.Unicode(), default='noname')

async def main():
    # 请根据实际情况，添加用户名和密码
    # 示例：POSTGRESQL://ZILLIONARE:123456@LOCALHOST/GINO
    # 并在本地 POSTGRES 数据库中，创建 GINO 数据库。
    await db.set_bind('postgresql://localhost/gino')
    await db.gino.create_all()

    # FURTHER CODE GOES HERE

    await db.pop_bind().close()

asyncio.run(main())
```

作为一个对代码有洁癖的人，我坚持始终使用`black`来格式化代码：

```
$ pip install black
$ black .
```

一切 ok，现在运行一下：

```
$ python foo/bar/data.py
```

检查数据库，发现 users 表已经创建。一切正常。

我希望这个程序在 macos, windows 和 linux 等操作系统上都能运行，并且可以运行在从 python 3.6 到 3.9 的所有版本上。

这里出现第一个问题。我需要准备 12 个环境：三个操作系统，每个操作系统上 4 个 python 版本，而且还要考虑如何进行"可复现的部署"的问题。在通过 ppw 创建的项目中，这些仅仅是通过修改 tox.ini 和.github\dev.yaml 中相关配置就可以做到了。但在没有使用 ppw 之前，我只能这么做：

在三台分别安装有 macos， windows 和 ubuntu 的机器上，分别创建 python 3.6 到 python 3.9 的虚拟环境，然后安装相同的依赖。首先，我通过`pip freeze`把开发机器上的依赖抓取出来：

```
$ pip freeze > requirements.txt
```

然后在另一台机器上的准备好的虚拟环境中，运行安装命令：

```
$ pip install -r requirements.txt
```

这里又出现了第二个问题。`black`纯粹是只用于开发目的，为什么也需要在测试/部署环境上安装呢？因此，在制作`requirements.txt`之前，我决定将`black`卸载掉：

```
$ pip uninstall -y black && pip freeze > requirements.txt
```

然而，仔细检查 requirements.txt 之后发现，`black`是被移除了，但仅仅是它自己。它的一些依赖，比如`click`， `tomli`等等，仍然出现在这个文件中。

> **Info**
>
> 这里的 click 就是我们前面提到的 Pallets 开发的那个 click。 black 作为格式化工具，它既可以作为 API 被其它工具调用，也可以作为独立应用，通过命令行来运行。black 就使用了 click 来进行命令行参数的解析。

于是，我不得不抛弃 pip freeze 这种作法，只在 requirements.txt 中加上直接依赖（在这里， black 是直接依赖，而 click 是间接依赖，由 black 引入），并且，将这个文件一分为二，将 black 放在 requirements_dev.txt 中。

```
# REQUIREMENTS.TXT
gino==1.0
```

```
# REQUIREMENTS_DEV.TXT
black==18.0
```

现在，在测试环境下，我们将只安装 requirements.txt 中的那些依赖。不出所料，项目运行得很流畅，目标达成，放心地去睡觉了。但是，gino 还依赖于 sqlalchemy 和 asyncpg。后二者被称为传递依赖。我们锁定了 gino 的版本，但是 gino 是否正确锁定了 sqlalchemy 和 asyncpg 的版本呢？这一切仍然不得而知。

第二天早晨醒来，sqlalchemy 1.4 版本发布了。突然地，当我再安装新的测试环境并进行测试时，程序报出了以下错误：

```
Traceback (most recent call last):
  File "/Users/aaronyang/workspace/best-practice-python/code/05/foo/foo/bar/
    from gino import Gino
  File "/Users/aaronyang/miniforge3/envs/bpp/lib/python3.9/site-packages/gin
    from .engine import GinoEngine, GinoConnection  # NOQA
  File "/Users/aaronyang/miniforge3/envs/bpp/lib/python3.9/site-packages/gin
    class GinoConnection:
  File "/Users/aaronyang/miniforge3/envs/bpp/lib/python3.9/site-packages/gin
    schema_for_object = schema._schema_getter(None)
AttributeError: module 'sqlalchemy.sql.schema' has no attribute '_schema_get
```

我差不多花了整整两天才弄明白发生了什么。我的程序依赖于 gino, 而 gino 又依赖于著名的 SQLAlchemy。gino 1.0 是这样锁定 SQLAlchemy 的版本的：

```
$pip install gino==1.0
Looking in indexes: https://pypi.jieyu.ai/simple, https://pypi.org/simple
Collecting gino==1.0
  Downloading gino-1.0.0-py3-none-any.whl (48 kB)
     |████████████████████████████████| 48 kB 129 kB/s
Collecting SQLAlchemy<2.0,>=1.2
  Downloading SQLAlchemy-1.4.0.tar.gz (8.5 MB)
     |████████████████████████████████| 8.5 MB 2.3 MB/s
```

> **Info**
>
> 上述文本是在 2021 年 3 月安装 gino 1.0 时的输出。如果您现在运行 `pip install gino==1.0`，会安装 SQLAlchemy 1.4.46 版本，这是它在 1.x 下的最后一个版本。

从 pip 的安装日志可以看到，gino 声明能接受的 SQLAlchemy 的最小版本是 1.2，最大版本则是不到 2.0。因此，当我们安装 gino 1.0 时，只要 SQLAlchemy 存在超过 1.2，且小于 2.0 的最新版本，它就一定会选择安装这个最新版本，最终，SQLAlchemy 1.4.0 被安装到环境中。

SQLAlchemy 在 2020 年也意识到了 asyncio 的重要性，并计划在 1.4 版本时转向 asyncio。然而，这样一来，调用接口就必须发生改变 -- 也就是，之前依赖于 SQLAlchemy 的那些程序，不进行修改是无法直接使用 SQLAlchemy 1.4 的。1.4.0 这个版本发布于 2021 年 3 月 16 日。

原因找到了，最终问题也解决了。最终，我把这个错误报告给了 gino，gino 的开发者承担了责任，发布了 1.0.1，将 SQLAlchemy 的版本锁定在">1.2,<1.4"这个范围内。

```
pip install gino==1.0.1
Looking in indexes: https://pypi.jieyu.ai/simple, https://pypi.org/simple
Collecting gino==1.0.1
  Using cached gino-1.0.1-py3-none-any.whl (49 kB)
Collecting SQLAlchemy<1.4,>=1.2.16
  Using cached SQLAlchemy-1.3.24-cp39-cp39-macosx_11_0_arm64.whl
```

在这个案例中，我并没有要求升级并使用 SQLAlchemy 的新功能，因此，新的安装本不应该去升级这样一个破坏性的版本；但是如果 SQLAlchemy 出了新的安全更新，或者 bug 修复，显然，我们也希望我们的程序在不进行更新发布的情况下，就能对依赖进行更新（否则，如果任何一个依赖发布安全更新，都将导致主程序不得不发布更新的话，这种耦合也是很难接受的）。因此，是否存在一种机制，使得我们的应用在指定直接依赖时，也可以恰当地锁定传递依赖的版本，并且允许传递依赖进行合理的更新？这是我们这个案例提出来的第三个问题。

现在，似乎是我们将产品发布的时候了。我们看到其它人开发的开源项目发布在 pypi 上，这很酷。我也希望我的程序能被千百万人使用。这就需要编写 MANINFEST.in, setup.cfg, setup.py 等文件。

MANIFEST.in 用来告诉 setup tools 哪些额外的文件应该被包含在发行包里，以及哪些文件则应该被排除掉。当然在我们这个简单的例子中，这个文件是可以被忽略的。

setup.py 中需要指明依赖项、版本号等等信息。由于我们已经使用了 requirements.txt 和 requirements_dev.txt 来管理依赖，所以，我们并不希望在 setup.py 中重复指定 -- 我们希望只更新 requirements.txt，就可以自动更新 setup.py：

```
from setuptools import setup

with open('requirements.txt') as f:
    install_requires = f.read().splitlines()
with open('requirements_dev.txt') as f:
    extras_dev_requires = f.read().splitlines()

# SETUP 是一个有着庞大参数体的函数，这里只显示了部分相关参数
setup(
    name='foo',
    version='0.0.1',
    install_requires=install_requires,
    extras_require={'dev': extras_dev_requires},
    packages=['foo'],
)
```

看上去还算完美。但实际上，我们每一次发布时，还会涉及到修改版本号等问题，这都是容易出错的地方。而且，它还不涉及打包和发布。通常，我们还需要编写一个 makefile，通过 makefile 命令来实现打包和发布。

这些看上去都是很常规的操作，为什么不将它自动化呢？这是第四个问题，即如何简化打包和发布。

这四个问题，就是我们这一章要讨论的主题。我们将以 Poetry 为主要工具，结合 semantic versioning 来串起这一话题的讨论。

1. SEMANTIC VERSIONING（基于语义的版本管理）

在软件开发领域中，我们常常对同一软件进行不断的修补和更新，每次更新，我们都保留大部分原有的代码和功能，修复一些漏洞，引入一些新的构件。

有一个古老的思想实验，被称之为忒修斯船（The Ship of Theseus）问题，它描述的正是同样的场景：

忒修斯船问题最早出自公元一世纪普鲁塔克的记载。它描述的是一艘可以在海上航行几百年的船，只要一块木板腐烂了，它就会被替换掉，以此类推，直到所有的功能部件都不是最开始的那些了。现在的问题是，最后的这艘船是原来的那艘忒修斯之船呢，还是一艘完全不同的船？如果不是原来的船，那么从什么时候起它就不再是原来的船了？

忒修斯船之问，发生在很多领域。象 IBM 这样的百年老店，不仅 CEO 换了一任又一任，就连股权也在不停地变更。可能很少人有在意，今天的 IBM，跟百年之前的 IBM 还是不是同一家 IBM，就象我们很少关注，人类是从什么时候起，不再是智人一样。又比如，如果有一家创业公司，当初吸引你加入，后来创始人变现走人了，尽管公司名字可能没换，但公司新进了管理层和新同学，业务也可能发生了一些变化。这家公司，还是你当初加入的公司吗？你是要选择潇洒的离开，还是坚持留下来？

在软件开发领域中，我们更是常常遇到同样的问题。每遇到一个漏洞（bug），我们就更换一块"木板"。随着这种修补和替换越来越多，软件也必然出现忒修斯船之问：现在的软件还是不是当初的软件，如果不是，那它是在什么时候不再是原来的软件了呢？

忒修斯船之问有着深刻的哲学内涵。我们在软件领域中，尽管也会遇到类似的问题，但回答就容易很多：

软件应该如何向外界表明它已发生了实质性的变化；生态内依赖于该软件的其它软件，又应该如何识别软件的蜕变呢？

为了解决上述问题，Tom Preston-Werner（Github 的共同创始人）提出 Semantic versioning 方案，即基于语义的版本管理。Semantic version 表示法提出的初衷是：

> **Quote**

> 在软件管理的领域里存在着被称作“依赖地狱”的死亡之谷，系统规模越大，加入的包越多，你就越有可能在未来的某一天发现自己已深陷绝望之中。
>
> 在强依赖的系统中发布新版本包可能很快会成为噩梦。如果依赖关系过强，可能面临版本控制被锁死的风险（必须对每一个依赖包改版才能完成某次升级）。而如果依赖关系过于松散，又将无法避免版本的混乱（假设兼容于未来的多个版本已超出了合理数量）。当你专案的进展因为版本依赖被锁死或版本混乱变得不够简便和可靠，就意味着你正处于依赖地狱之中。

Semantic versioning 简单地说，就是用版本号的变化向外界表明软件变更的剧烈程度。要理解 Semantic versioning，我们首先得了解软件的版本号。

当我们说起软件的版本号时，我们通常会意识到，软件的版本号一般由主版本号 (major)，次版本号 (minor)，修订号 (patch) 和构建编号 (build no.) 四部分组成。由于 Python 程序没有其它语言通常意义上的构建，所以，对 Python 程序而言，一般只用三段，即 major.minor.patch 来表示。

> **Info**
>
> 实际上，出于内部开发的需要，我们仍然可能给 Python 程序的版本用上 build no，特别是在 CI 集成中。当我们向仓库推送一个 commit 时，CI 都需要进行一轮构建和自动验证，此时并不会修改正式版本号，因此，一般倾向于使用构建号来区分不同的 commit 导致的版本上的不同。在 python project wizard 生成的项目中，其 CI 就实现了这个逻辑。

上述版本表示法没有反映出任何规则。在什么情况下，你的软件应该定义为 0.x，什么时候又应该定义为 1.x，什么时候递增主版本号，什么时候则只需要递增修订号呢？如果不同的软件生产商对以这些问题没有共识的话，会产生什么问题吗？

实际上，由于随意定义版本号引起的问题很多。在前面我们提到过 SQLAlchemy 的升级导致许多 Python 软件不能正常工作的例子。在讲述那个例子时，我指出，是 gino 的开发者承担了责任，发行了新的 gino 版本，解决了这个问题。但实际上，责任的根源在 SQLAlchemy 的开发者那里。

从 1.3.x 到 1.4.x, 出现了接口的变更，这是一种破坏性的更新，此时，新的 1.4 已不再是过去的忒修斯之船了，使用者如果不修改他们的调用方式，就无法使用 SQLAlchemy 的

问题。gino 的开发者认为（这也是符合 semantic versioning 思想的），SQLAlchemy 从 1.2 到 2.0 之间的版本，可以增加接口，增强性能，修复安全漏洞，但不应该变更接口；因此，它声明为依赖 SQLAlchemy 小于 2.0 的版本是安全的。但可惜的是，SQLAlchemy 并没有遵循这个约定。

Sematic versioning 提议用一组简单的规则及条件来约束版本号的配置和增长。首先，你规划好公共 API，在此后的新版本发布中，通过修改相应的版本号来向大家说明你的修改的特性。考虑使用这样的版本号格式：X.Y.Z（主版本号. 次版本号. 修订号）：修复问题但不影响 API 时，递增修订号；API 保持向下兼容的新增及修改时，递增次版本号；进行不向下兼容的修改时，递增主版本号。

我们在前面提到过 SQLAlchemy 从 1.x 升级到 1.4 的例子。实际上，由于引入了异步机制，这是个不能向下兼容的修改，因此，SQLAlchemy 本应该启用 2.x 的全新版本序列号，而把 1.4 留作 1.x 的后续修补发布版本号使用。如此一来，SQLAlchemy 的使用者就很容易明白，如果要使用最新的 SQLAlchemy 版本，则必须对他们的应用程序进行完全的适配和测试，而不能象之前的升级一样，简单地把最新版本安装上，就仍然期望它能像之前一样工作。不仅如此，一个定义了良好依赖关系的软件，还能自动从升级中排除掉升级到 SQLAlchemy 2.x，而始终只在 1.x，甚至更小的范围内进行升级。

> **Info**
>
> SQLAlchemy 的错误并非孤例。一个影响范围更广的例子涉及到 python 的 cryptography 库。这是一个广泛使用的密码学相关的 python 库。为了提升性能，许多代码最初是使用 c 写的。有一天作者意识到，使用 c 会存在很多安全问题，而安全性又是 cryptography 的核心。于是，在 2021 年 2 月 8 日前后，他们改用 rust 来进行实现。这导致安装 cryptography 库的人，必须在本机上有 rust 的编译工具链 -- 事实是，rust 与 c 和 python 相比，还是相当小众的，很多人的机器上显然不会有这套工具链。
>
> 需要指出的是，cryptography 改用 rust 实现，并没有改变它的 python 接口。相反，其 python 接口完全保持着一致。因此，cryptography 的作者也既没有重命名 cryptography，也没有变更主版本号。
>
> 但是这一小小的改动，仍然掀起了轩然大波。一夜之间，它摧毁了无数的 CI 系统，无数 docker 镜像必须被重构，抱怨声如潮水般涌向作者。在短短几个小时，作者就收到了 100 条激烈的评论，最终作者不得不关掉了这个 issue

一个正确地使用 semantic versioning 的例子是 aioredis 从 1.x 升级到 2.0。尽管 aioredis 升级到 2.0 时，大多数 API 并没有发生改变--只是在内部进行了性能增强，但它的确改变了初始化 aioredis 的方式，从而使得你的应用程序，不可能不加修改就直接更新到 2.0 版本。因此，aioredis 在这种情况下，将版本号更新为 2.0 是非常正确的。

事实上，如果你的程序的 API 发生了变化（函数签名发生改变），或者会导致旧版的数据无法继续使用，你都应该考虑主版本号的递增。

此外，从 0.1 到 1.0 之前的每一个 minor 版本，都被认为在 API 上是不稳定的，都可能是破坏性的更新。因此，如果你的程序使用了还未定型到 1.0 版本的第三方库，你需要谨慎地声明依赖关系。而我们自己如果作为开发者，在软件功能稳定下来之前，不要轻易地将版本发布为1.0。

2. POETRY：简洁清晰的项目管理工具

[Poetry] 是一个依赖管理和打包工具。Poetry 的作者解释开发 Poetry 的初衷时说：

> **Quote**
>
> Packaging systems and dependency management in Python are rather convoluted and hard to understand for newcomers. Even for seasoned developers it might be cumbersome at times to create all files needed in a Python project: setup.py, requirements.txt, setup.cfg, MANIFEST.in and the newly added Pipfile. So I wanted a tool that would limit everything to a single configuration file to do: dependency management, packaging and publishing.

> 翻译：Python 的打包系统和依赖管理相当复杂，对新人来讲尤其费解。要正确地创建 Python 项目所需要的文件：setup.py, requirements.txt, setup.cfg, MANIFEST.in 和新加入的 pipfile，有时候即使对一个有经验的老手，也是有一些困难的。因此，我希望创建一种工具，只用一个文件就实现依赖管理、打包和发布。

通过前面的案例，我们已经提出了一些问题。但不止于此。

当您将依赖加入到 requirements.txt 时，没有人帮你确定它是否与既存的依赖能够和平共处，这个过程要比我们想象的复杂许多，不仅仅是直接依赖，还需要考虑彼此的传递依赖是否也能彼此兼容；所以一般的做法是，先将它们加进来，完成开发和测试，在打包之前，运行`pip freeze > requirements.txt`来锁定依赖库的版本。但我们也在前面的案例中提到，这种方法可能会将不必要的开发依赖打入到发行版中；此外，它也会过度锁定版本，从而使得一些活跃的第三方库失去自动更新热修复和安全更新的机会。

项目的版本管理也是一个问题。在老旧的 Python 项目中，一般我们使用 bumpversion 来管理版本，它需要使用三个文件。在我的日常使用时，它常常会出现各种问题，最常见的是单双引号导致把`__version__=0.1`当成一个版本号，而不是`0.1`。这样打出来的包名也会奇怪地多一个无意义的 version 字样。单双引号则是因为你的 format 工具对字符串常量应该使用什么样的引号规则有自己的意见。

项目进行打包和发布需要准备太多的文件，正如 Poetry 的开发者所说，要确保这些文件的内容完全正确，对一个有经验的开发者来说，也不是轻而易举的事。

Poetry 解决了所有这些问题（除了案例中的第一个，该问题要通过 tox 和 CI 来解决）。它提供了版本管理、依赖解析、构建和发布的一站式服务，并将所有的配置，集中到一个文件中，即 pyproject.toml。此外，Poetry 还提供了一个简单的工程创建向导。不过这个向导的功能仍然过于简单，我们的推荐则是使用上一章介绍的 python project wizard。

> **Info**
>
> 实际上 Poetry 还会用到另一个文件，即 poetry.lock。这个文件并非独立文件，而是 Poetry 根据 pyproject.toml 生成的、锁定了依赖版本的最终文件。它的主要作用，是在一组开发者之间，帮助其它开发者省去依赖解析的时间。
>
> 因此，当你通过 poetry 向项目中增加（或者移除）了新的依赖时，该文件会被更新。您应该把该文件也提交到代码仓库中。但是，该文件并不会发布给最终用户。

现在，让我们看一眼 sample 项目中的 pyproject.toml 文件：

```
[tool]
[tool.poetry]
name = "sample"
version = "0.1.0"
homepage = "https://github.com/zillionare/sample"
description = "Skeleton project created by Python Project Wizard (ppw)."
authors = ["aaron yang <aaron_yang@jieyu.ai>"]
readme = "README.md"
license =  "MIT"
classifiers=[
    'Development Status :: 2 - Pre-Alpha',
    'Intended Audience :: Developers',
    'License :: OSI Approved :: MIT License',
    'Natural Language :: English',
    'Programming Language :: Python :: 3',
    'Programming Language :: Python :: 3.7',
    'Programming Language :: Python :: 3.8',
    'Programming Language :: Python :: 3.9',
    'Programming Language :: Python :: 3.10',
]
packages = [
    { include = "sample" },
    { include = "tests", format = "sdist" },
]

[tool.poetry.dependencies]
python = ">=3.7.1,<4.0"
fire = "0.4.0"

black  = { version = "^22.3.0", optional = true}
isort  = { version = "5.10.1", optional = true}
flake8  = { version = "4.0.1", optional = true}
flake8-docstrings = { version = "^1.6.0", optional = true }
pytest  = { version = "^7.0.1", optional = true}
pytest-cov  = { version = "^3.0.0", optional = true}
tox  = { version = "^3.24.5", optional = true}
virtualenv  = { version = "^20.13.1", optional = true}
pip  = { version = "^22.0.3", optional = true}
mkdocs  = { version = "^1.2.3", optional = true}
mkdocs-include-markdown-plugin  = { version = "^3.2.3", optional = true}
mkdocs-material  = { version = "^8.1.11", optional = true}
```

```
mkdocstrings  = { version = "^0.18.0", optional = true}
mkdocs-material-extensions  = { version = "^1.0.3", optional = true}
twine  = { version = "^3.8.0", optional = true}
mkdocs-autorefs = {version = "^0.3.1", optional = true}
pre-commit = {version = "^2.17.0", optional = true}
toml = {version = "^0.10.2", optional = true}
livereload = {version = "^2.6.3", optional = true}
pyreadline = {version = "^2.1", optional = true}
mike = { version="^1.1.2", optional=true}

[tool.poetry.extras]
test = [
    "pytest",
    "black",
    "isort",
    "flake8",
    "flake8-docstrings",
    "pytest-cov"
    ]

dev = ["tox", "pre-commit", "virtualenv", "pip", "twine", "toml"]

doc = [
    "mkdocs",
    "mkdocs-include-markdown-plugin",
    "mkdocs-material",
    "mkdocstrings",
    "mkdocs-material-extension",
    "mkdocs-autorefs",
    "mike"
    ]

[tool.poetry.scripts]
sample = 'sample.cli:main'

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"

[tool.black]
line-length = 88
```

```
include = '\.pyi?$'
exclude = '''
/(
    \.eggs
  | \.git
  | \.hg
  | \.mypy_cache
  | \.tox
  | \.venv
  | _build
  | buck-out
  | build
  | dist
)/
'''
[tool.isort]
profile = "black"
```

我们简单地解读一下这个文件：
在 [tool.poetry] 那一节，定义了包的名字（这里是 sample）、版本号（这里是 0.1.0）和其它的一些字段，比如 classifiers，这是打包和发布时需要的。如果您熟悉 python setup tools，那么对这些字段将不会陌生。packages 字段指明了打包时需要包含的文件。在示例中，我们要求在以.whl 格式发布的包中，将 sample 目录下的所有文件打包发布；而以 sdist 格式（即.tar.gz) 发布的包中，还要包含 tests 目录下的文件。

接下来是 [tool.poetry.dependencies] 一节，这是我们声明项目依赖的地方。首先是项目要求的 python 版本声明。这里我们要求必须在 3.7.1 以上，4.0 以下的 python 环境中运行。因此，python 3.7.1，3.8， 3.9， 3.10 都是恰当的 python 版本，但 4.0 则不允许。

接下来就是工程中需要用到的其它第三方依赖，有运行时的（即当最终用户使用我们的程序时，必须安装的那些第三方依赖），也有开发时的（即只在开发和测试过程中使用到的，比如文档工具类 mkdocs，测试类 tox, pytest 等）。

我们对运行时和开发时需要的依赖进行了分组。对开发时需要的依赖，我们分成 dev, test 和 doc 三组，通过 [tool.poetry.extras] 中进行分组声明。对于归入到 dev, test 和 doc 分组中的依赖，我们在 [tool.poetry.dependencies] 中，将其声明为 optional 的，这样在安装最终分发包时，这些声明为 optional 的第三方依赖将不会安装到用户环境中。

再接下来，[tool.poetry.scripts] 声明了一个 console script 入口。Console script 是一种特殊的 Python 脚本，它使得您可以象调用普通的 shell 命令一样来调用这个脚本。

```
[tool.poetry.scripts]
sample = 'sample.cli:main'
```

> **Info**
>
> 能制作 console scrip 是 Python 另一个优势 -- 这在 Linux/Mac 上尤其明显。这样一来，我们就可以很容易地通过 Python 往 shell 中增加各种命令，使得它们还可以串行起来。此外，如果我们通过 Python script 来提供了服务，我们就很需要通过命令来管理服务，比如启动、停止和显示服务状态。

当 sample 包被安装后，就往安装环境里注入了一个名为`sample`的shell 命令。它可以接受各种参数，最终将交给 sample\cli.py 中的 main 函数来执行。

接下来就是关于如何构建的相关指示，在 [build-system] 中。如果你的程序中只包含纯粹的 Python 代码，那么这部分可不做任何修改。如果你的程序包含了一些原生的代码（比如 c 的），那么就需要自己定义构建脚本。

在示例代码中，还有 [tool.black] 和 [tool.isort] 两个小节，分别是 black（代码格式化工具）和 isort（将导入进行排序的工具）的配置文件。它们是对 pyproject.toml 的扩展，并不是 poetry 所要求的。

2.1. 版本管理

poetry 为我们的 package 提供了基于语义 (semantic version) 的版本管理功能。它通过`poetry version`这个命令，让我们查看 package 的版本，并且实现版本号的升级。

假设您已经使用 [python project wizard] 生成了一个工程框架，那么应该可以在根目录下找到 pyproject.toml 文件，其中有一项：

```
version = 0.1
```

如果您现在运行`poetry version`这个命令，就会显示`0.1`这个版本号。

Poetry 使用基于语义的版本 (semantic version) 表示法。

在 Poetry 中，当我们需要修改版本号时，并不是直接指定新的版本号，而是通过`poetry version semver`来修改版本。`semver`可以是`patch`，`minor`，`major`，`prepatch`，`preminor`，`premajor`和`prerelease`中的一个。这些关键字定义在规范 PEP 440 中。

将`semver`与您当前的版本号相结合，通过运算，就得出了新的版本号：

rule	before	after
major	1.3.0	2.0.0
minor	2.1.4	2.2.0
patch	4.1.1	4.1.2
premajor	1.0.2	2.0.0-alpha.0
preminor	1.0.2	1.1.0-alpha.0
prepatch	1.0.2	1.0.3-alpha.0
prerelease	1.0.2	1.0.3-alpha.0
prerelease	1.0.3-alpha.0	1.0.3-alpha.1
prerelease	1.0.3-beta.0	1.0.3-beta.1

可以看出，poetry 对版本号的管理是完全符合 semantic version 的要求的。当你完成了一个小的修订（比如修复了一个 bug，或者增强了性能，或者修复了安全漏洞），此时只应该递增 package 的修订号，即 x.y.z 中的'z'，这时我们就应该使用命令：

```
$ poetry version patch
```

如果之前的版本是 0.1.0，那么运行上述命令后，版本号将变更为 0.1.1。
如果我们的 package 新增加了一些功能，而之前提供的功能（API）都还能不加修改，继续使用，那么我们应该递增次版本号，即 x.y.z 中的'y'。这时我们应该使用命令：

```
$ poetry version minor
```

如果之前的版本是 0.1.1，那么运行上述命令后，版本号将变更为 0.2.0

如果我们的 package 进行了大幅的修改，并且之前提供的功能（API）的签名已经变掉，从而使得调用者必须修改他们的程序才能继续使用这些 API，又或者新的版本不再能兼容老版本的数据格式，用户必须对数据进行额外的迁移，那么，我们就认为这是一次破坏性的更新，必须升级主版本号：

```
$ poetry version major
```

如果之前的版本号是 0.3.1, 那么运行上述命令之后，版本号将变更为 1.0.0；如果之前的版本号是 1.2.1，那么运行上述命令之后，版本号将变更为 2.0.0。

除此之外，poetry 还提供了预发布版本号的支持。比如，上一个发布的版本是 0.1.0，那么我们在正式发布 0.1.1 这个修订之前，可以使用 0.1.1.a0 这个版本号：

```
$ poetry version prerelease
Bumping version from 0.1.0 to 0.1.1a0
```

如果需要再出一个 alpha 版本，则可以再次运行上述命令：

```
$ poetry version prerelease
Bumping version from 0.1.1a0 to 0.1.1a1
```

如果 alpha 版本已经完成，可以正式发布，运行下面的命令：

```
$ poetry version patch
Bumping version from 0.1.1a1 to 0.1.1
```

poetry 暂时还没有提供从 alpha 转到 beta 版本系列的命令。如果有此需要，您需要手工编辑 pyproject.toml 文件。

除了 poetry version prerelease 之外，我们还注意到上面列出的 premajor, preminor 和 prepatch 选项。它们的作用也是将版本号修改为 alpha 版本系列，但无论你运行多少次，它们并不会象 prerelease 选项一样，递增 alpha 版本号。所以在实际的 alpha 版本管理中，似乎只使用`poetry version prerelease`就可以了。

2.2. 依赖管理

2.2.1. 实现依赖管理的意义

我们已经通过大量的例子说明了依赖管理的作用。总结起来，依赖管理不仅要检查项目中声明的直接依赖之间的冲突，还要检查它们各自的传递依赖之间的彼此兼容性。

2.2.2. Poetry 进行依赖管理的相关命令

在 Poetry 管理的工程中，当我们向工程中加入（或者更新）依赖时，总是使用`poetry add`命令，比如：`poetry add pytest`

这里可以指定，也可以不指定版本号。命令在执行时，会对`pytest`所依赖的库进行解析，直到找到合适的版本为止。如果您指定了版本号，该版本与工程里已有的其它库不兼容的话，命令将会失败。

我们在添加依赖时，一般要指定较为准确的版本号，界定上下界，从而避免意外升级带来的各种风险。在指定依赖库的版本范围时，有以下各种语法：

```
$ poetry add SQLAlchemy                # 使用最新的版本
```

使用通配符语法：

```
# 使用任意版本，无法锁定上界，不推荐
$ poetry add SQLAlchemy=*

# 使用>=1.0.0, <2.0.0 的版本
$ poetry add SQLAlchemy=1.*
```

使用插字符 (caret) 语法：

```
# 使用>=1.2.3, <2.0.0 的版本
$ poetry add SQLAlchemy^1.2.3

# 使用>=1.2.0, <2.0.0 的版本
$ poetry add SQLAlchemy^1.2

# 使用>=1.0.0, <2.0.0 的版本
$ poetry add SQLAlchemy^1
```

使用波浪符 (Tilde) 语法：

```
# 使用>=1.2.0,<1.3 的版本
$ poetry add SQLAlchemy~1.2

# 使用>=1.2.3，<1.3 的版本
$ poetry add SQLAlchemy~1.2.3
```

使用不等式语法（及多个不等式）:

```
# 使用>=1.2,<1.4 的版本
$ poetry add SQLAlchemy>=1.2,<1.4
```

最后，精确匹配语法：

```
# 使用 1.2.3 版本
$ poetry add SQLAlchemy==1.2.3
```

如果有可能，我们推荐总是使用波浪符或者不等式语法。它们有助于在可升级性和可匹配性上取得较好的平衡。比如，如果在增加对 SQLAlchemy 的依赖时，如果使用了插字符语法，已经发行出去的安装包，则会在安装时自动采用直到 2.0.0 之前的 SQLAlchemy 的最新版本。因此，如果你的安装包是在 SQLAlchemy 1.4 之前被安装，此后用户不再升级，则它们将可以正常运行；而如果是在 SQLAlchemy 1.4 发布之后被安装，pip 将自动使用 1.4 及以后最新的 SQLAlchemy，于是 这个跟之前版本不兼容的1.4版本就被安装上了，导致你的程序崩溃；除非发行新的升级包，而你将不会有任何办法来解决这一问题。

这也看出来 SQLAlchemy 的发行并不符合 Semantic 的标准。一旦出现 API 不兼容的情况，是需要对主版本升级的。如果 SQLAlchemy 不是将版本升级到 1.4，则是升级到

2.0，则不会导致程序出现问题。

始终遵循社区规范进行开发，这是每一个开源程序开发者都应该重视的问题。

指定过于具体的版本也会有它的问题。在向工程中增加依赖时，如果我们直接指定了具体的版本，有可能因为依赖冲突的原因，无法指定成功。此时可以指定一个较宽泛一点的版本范围，待解析成功和测试通过后，再改为固定版本。另外，如果该依赖发布了一个紧急的安全更新，通常会使用递增修订号的方式来递增版本。使用指定的版本号会导致你的应用无法快速获得此安全更新。

在上一章里，我们已经提到了依赖分组。我们的应用程序会依赖许多第三方库，这些第三方库中，有的是运行时依赖，因此它们必须随我们的程序一同被分发到终端用户那里；有的则只是开发过程中需要，比如象 pytest，black，mkdocs 等等。因此，我们应该将依赖进行分组，并且只向终端用户分发必要的依赖。

这样做的益处是显而易见的。一方面，依赖解析并不容易，一个程序同时依赖的第三方库越多，依赖解析就越困难，耗时越长，也越容易失败；另一方面，我们向终端用户的环境里注入的依赖越多，他们的环境中就越容易遇到依赖冲突问题。

最新的 Python 规范允许你的程序使用发行依赖（在最新的 poetry 版本中，被归类为 main 依赖）和 extra requirements。在上一章向导创建的工程中，我们把 extra reuqirement 分为了三个组，即 dev, test, doc。

```
[tool.poetry.dependencies]
black  = { version = "20.8b1", optional = true}
isort  = { version = "5.6.4", optional = true}
flake8  = { version = "3.8.4", optional = true}
flake8-docstrings = { version = "^1.6.0", optional = true }
pytest  = { version = "6.1.2", optional = true}
pytest-cov  = { version = "2.10.1", optional = true}
tox  = { version = "^3.20.1", optional = true}
virtualenv  = { version = "^20.2.2", optional = true}
pip  = { version = "^20.3.1", optional = true}
mkdocs  = { version = "^1.1.2", optional = true}
mkdocs-include-markdown-plugin  = { version = "^1.0.0", optional = true}
mkdocs-material  = { version = "^6.1.7", optional = true}
mkdocstrings  = { version = "^0.13.6", optional = true}
mkdocs-material-extensions  = { version = "^1.0.1", optional = true}
twine  = { version = "^3.3.0", optional = true}
mkdocs-autorefs = {version = "0.1.1", optional = true}
pre-commit = {version = "^2.12.0", optional = true}
toml = {version = "^0.10.2", optional = true}

[tool.poetry.extras]
test = [
    "pytest",
    "black",
    "isort",
    "flake8",
    "flake8-docstrings",
    "pytest-cov",
    "twine"
    ]

dev = ["tox", "pre-commit", "virtualenv", "pip",  "toml"]

doc = [
    "mkdocs",
    "mkdocs-include-markdown-plugin",
    "mkdocs-material",
    "mkdocstrings",
    "mkdocs-material-extension",
```

```
    "mkdocs-autorefs"
    ]
```

这里 tox， pre-commit 等是我们开发过程中使用的工具；pytest 等是测试时需要的依赖；而 doc 则是构建文档时需要的工具。通过这样划分，可以使 CI 或者文档托管平台只安装必要的依赖；同时也容易让开发者分清每个依赖的具体作用。

当你使用 poetry add 命令，不加任何选项时，该依赖将被添加为发行依赖（在 1.3 以上的 poetry 中，被归为 main 组），即安装你的包的最终用户，他们也将安装该依赖。但有一些依赖只是开发者需要，比如象 mkdocs， pytest 等，它们不应该被分发到最终用户那里。

在 python proejct wizard 开发时，poetry 还只支持一个 dev 分组，这样的粒度当然是不够的，因此，python project wizard 借用了 extras 字段来向项目添加可选依赖分组，其它工具，比如 tox 也支持这样的语法。

现在最新的 poetry 已经完全支持分组模式，并且从文档可以看出，它建议至少使用 main, docs 和 test 三个分组。后续 python project wizard 生成的项目框架，也将完全使用最新的语法，但仍然保留四个分组，即 main, dev, docs 和 test。

通过 poetry 向项目增加分组及依赖，语法是：

```
$ poetry add pytest --group test
```

这样，生成的 pyproject.toml 片段如下：

```
[tool.poetry.group.test.dependencies]
pytest = "*"
```

一般地，我们应该将其指定为 optional。目前最新版本的 poetry 仍然不支持通过命令行直接将 group 指定为 optional，您可能需要手工编辑这个文件。

```
[tool.poetry.group.test]
optional = true
```

Info

> 注意，通过上述命令生成的 toml 文件的内容可能与 python project wizard 当前版本生成的有所不同。但 python project wizard 的未来版本最终将使用同样的语法。

2.2.3. poetry 依赖解析的工作原理

在上一节，我们简单地介绍了如何使用 poetry 来向我们的项目中增加依赖。我们强调了依赖解析的困难，但并没有解释 poetry 是如何进行依赖解析的，它会遇到哪些困难，可能遭遇什么样的失败，以及应该如何排错。对于初学者来说，这往往是配置 poetry 项目时最困难和最耗时间的部分。

现在，我们往项目中增加一个新的依赖，通常我们使用`poetry add xxx`来往项目中增加依赖。为了一窥 poetry 依赖解析的究竟，这次我们加上详细信息输出：

```
$ poetry add gino -vvv
```

输出会很长很长。我们摘要读一下跟 gino 相关的一些解析过程：

首先，poetry 注意到 sample 0.1.0 依赖到 gino(>=1.0.1, < 2.0.0)，以及其它一些依赖，生成了第一步的解析结果：

```
   1: fact: sample is 0.1.0
   1: derived: sample
   1: selecting sample (0.1.0)
   1: derived: gino (>=1.0.1,<2.0.0)
   1: derived: mike (>=1.1.2,<2.0.0)
    ...
```

接下来，下载 gino，解析出下面的依赖：

```
1 packages found for gino >=1.0.1,<2.0.0
   1: fact: gino (1.0.1) depends on SQLAlchemy (>=1.2.16,<1.4)
   1: fact: gino (1.0.1) depends on asyncpg (>=0.18,<1.0)
   1: selecting gino (1.0.1)
   1: derived: asyncpg (>=0.18,<1.0)
   1: derived: SQLAlchemy (>=1.2.16,<1.4)
```

再接下来，它找到 sqlalchemy 的 29 个版本：

```
Source (ali): 14 packages found for asyncpg >=0.18,<1.0
Source (ali): 29 packages found for sqlalchemy >=1.2.16,<1.4
```

接下来比较幸运，当 poetry 查找 asyncpg 和 sqlalchemy 的传递依赖时，没有找到它们有更多的传递依赖，解析结束，这样，poetry 就顺利地选择了 29 个版本中，最新的一个，即 SQLAlchemy-1.3.24。这个版本又有 linux, windows 和 mac 等好几个包，poetry 最终选择跟当前环境中操作系统版本一致、python 版本一致的那个进行安装。

现在让我们看一看 poetry 最终解析出来的依赖树：

```
$ poetry show -t
black 22.12.0 The uncompromising code formatter.
├── click >=8.0.0
│   ├── colorama *
│   └── importlib-metadata *
│       ├── typing-extensions >=3.6.4
│       └── zipp >=0.5
├── mypy-extensions >=0.4.3
├── pathspec >=0.9.0
├── platformdirs >=2
│   └── typing-extensions >=4.4
├── tomli >=1.1.0
├── typed-ast >=1.4.2
└── typing-extensions >=3.10.0.0
gino 1.0.1 GINO Is Not ORM - a Python asyncio ORM on SQLAlchemy core.
├── asyncpg >=0.18,<1.0
└── sqlalchemy >=1.2.16,<1.4
mkdocs 1.2.4 Project documentation with Markdown.
├── click >=3.3
│   ├── colorama *
│   └── importlib-metadata *
│       ├── typing-extensions >=3.6.4
│       └── zipp >=0.5
├── ghp-import >=1.0
│   └── python-dateutil >=2.8.1
│       └── six >=1.5
├── importlib-metadata >=3.10
│   ├── typing-extensions >=3.6.4
│   └── zipp >=0.5
├── jinja2 >=2.10.1
│   └── markupsafe >=2.0
├── markdown >=3.2.1
│   └── importlib-metadata *
│       ├── typing-extensions >=3.6.4
│       └── zipp >=0.5
├── mergedeep >=1.3.4
├── packaging >=20.5
├── pyyaml >=3.10
├── pyyaml-env-tag >=0.1
│   └── pyyaml *
```

```
└── watchdog >=2.0
```

这个依赖树很长，这里只截取了一小部分，但大致上可以帮助我们了解 poetry 的工作原理。我们可以看到 `black` 和 `mkdocs` 都依赖了 `click` ，但 `black` 要求更新到 8.0 以上，而 `mkdocs` 则认为只要是 3.3 以上都可以。两者版本要求差距如此之大，也不免让人担心，8.0 的 `click` 与 3.3 的 `click` 还会是同一个 `click` 吗？

最终，关于 gino 和 sqlalchemy，poetry 安装的分别是 1.0.1 和 1.3.24，但是，上述解析树表明，如果存在 sqlalchemy 的 1.3.25 版本，它是可以自动升级的。我们许的愿，poetry 帮助实现了。

生成这棵依赖树可能要比你想像的困难得多。首先，PyPI 目前还没有给出它上面的某一个 package 的依赖树，这意味着 poetry 要知道 `black` 依赖哪些库，它必须先把 `black` 下载下来，打开它并解析才能知道。然后它从 `black` 中发现更多的依赖，这往往就需要它把这些依赖也下载下来，依次递归下去。

> **Info**
>
> 类似的系统在其它语言中已经存在了。比如Java有maven来保存各个开源库的依赖树。在依赖解析时，它不需要下载整个包，而只需要下载索引就可以进行解析，因此速度会更快一些。

更为糟糕的是，在这个过程中，某个库的好几个版本可能都需要依次下载下来 -- 因为它们的传递依赖不能兼容。我记得在某次解析中，poetry 把 numpy 的版本从 1.2.x 一直下载到了 0.1！最终还是失败了。

所以，如果你在添加某个依赖时，发现 poetry 耗时过长，不要慌张，很多人都有与你一样的经历。这种情况主要是 poetry 无法快速锁定某个 package 的正确版本，不得不向后一个个版本搜索下载所致。我们能做的，就是加快 poetry 下载的速度。

poetry 正常情况下，是从 pypi.org 上下载 package。如果遇到解析速度问题，我们可以临时添加一个源：

```
poetry source add ali https://mirrors.aliyun.com/pypi/simple --default
```

再次运行`poetry add`，这次你会发现解析速度快了很多。

> **Info**
>
> 早期 poetry 的依赖解析可以慢到 10 多个小时都做不完。这有两方面的原因，一是早期 poetry 的依赖解析还没有启用多线程下载优化；二是在特殊情况下，poetry 需要把某些 package 在 pypi 上所有的版本全部下载一次，才能得出无法（或者可以）加入该依赖的结论。随着 python 生态的变化，现在这种需要数小时的依赖解析的时代基本结束了。在添加国内源的情况下，慢的时候也往往是不到一刻钟就能完成解析。

现在我们来移除 gino：

```
$ poetry remove gino
Updating dependencies
Resolving dependencies... (1.2s)

Writing lock file

Package operations: 0 installs, 0 updates, 3 removals

  • Removing asyncpg (0.27.0)
  • Removing gino (1.0.1)
  • Removing sqlalchemy (1.3.24)
```

可以看出，不仅是 gino 本身被卸载，它的传递依赖 -- asyncpg 和 sqlalchemy 也被移除掉了。这是 pip 做不到的。

2.3. 虚拟运行时

Poetry 自己管理着虚拟运行时环境。当你执行`poetry install`命令时，Poetry 就会安装一个基于 venv 的虚拟环境，然后把项目依赖都安装到这个虚拟的运行环境中去。此后，当你通过 poetry 来执行其它命令时，比如`poetry pytest`，也会在这个虚拟环境中执行。反之，如果你直接执行`pytest`，则会报告一些模块无法导入，因为你的工程依赖并没有安装在当前的环境下。

我们推荐在开发过程中，使用 conda 来创建集中式管理的运行时。在调试 Python 程序时，都要事先给 IDE 指定解析器，这里使用集中式管理的运行时，可能更方便一点。Poetry 也允许这种做法。当 Poetry 检测到当前是运行在虚拟运行时环境下时，它是不会创建新的虚拟环境的。

但是 Poetry 的创建虚拟环境的功能也是有用的，主要是在测试时，通过 virtualenv/venv 创建虚拟环境速度非常快。

2.4. 构建发行包

2.4.1. Python 构建标准和工具的变化

在 poetry 1.0 发布之前，打包一个 python 项目，需要准备 MANIFEST.in, setup.cfg, setup.py，makefile 等文件。这是 PyPA(python packaging authority) 的要求，只有遵循这些要求打出来的包，才可以上传到 pypi.org，从而向全世界发布。

但是这一套系统也有不少问题，比如缺少构建时依赖声明，自动配置，版本管理。因此，PEP 517 被提出，然后基于 PEP 517, PEP 518 等一系列新的标准，Sébastien Eustace 开发了 poetry。

2.4.2. 基于 Poetry 进行发行包的构建

我们通过运行 `poetry build` 来打包，打包的文件约定俗成地放在 dist 目录下。

poetry 支持向 pypi 进行发布，其命令是 `poetry publish`。不过，在运行该命令之前，我们需要对 poetry 进行一些配置，主要是 repo 和 token。

```
# PUBLISH TO TEST PYPI
$ poetry config repositories.testpypi https://test.pypi.org/legacy/
$ poetry config testpypi-token.pypi my-token
$ poetry publish -r testpypi

# PUBLISH TO PYPI
$ poetry config pypi-token.pypi my-token
$ poetry publish
```

上面的命令分别对发布到 test pypi 和 pypi 进行了演示。默认地 Poetry 支持 PyPI 发布，所以有些参数就不需要提供了。当然，一般情况下，我们都不应该直接运

行`poetry publish`命令来发布版本。版本的发布，都应该通过 CI 机制来进行。这样的好处时，可以保证每次发布，都经过了完整的测试，并且，构建环境是始终一致的，不会出现因构建环境不一致，导致打出来的包有问题的情况。

2.5. 其它重要的 Poetry 命令

我们已经介绍了 poetry add, poetry remove, poetry show, poetry build, poetry publish, poetry version 等命令。还有一些命令也值得介绍。

2.5.1. poetry lock

该命令将进行依赖解析，锁定所有的依赖到最新的兼容版本，并将结果写入到 poetry.lock 文件中。通常，运行 poetry add 时也会生成新的锁定文件。

在对代码执行测试、CI 或者发布之前，务必要确保 poetry.lock 存在，并且这个文件也应该提交到代码仓库中，这样所有的测试，CI 服务器，你的同侪开发者构建的环境才会是完全一致的。

2.5.2. poetry export

```
$ poetry export -f requirements.txt --output requirements.txt
```

2.5.3. poetry config

我们可以通过 poetry config --list 来查看当前配置项：

```
cache-dir = "/path/to/cache/directory"
virtualenvs.create = true
virtualenvs.in-project = null
virtualenvs.options.always-copy = true
virtualenvs.options.no-pip = false
virtualenvs.options.no-setuptools = false
virtualenvs.options.system-site-packages = false
virtualenvs.path = "{cache-dir}/virtualenvs"  # /path/to/cache/directory/vi
virtualenvs.prefer-active-python = false
virtualenvs.prompt = "{project_name}-py{python_version}"
```

这里面比较重要的有配置 pypi-token，配置之后，就可以免登录进行项目发布。不过，我们建议对重要项目，不要在本地配置这个token， 我们应该只在CI/CD系统中配置这个token，以实现仅从CI/CD进行发布。