探索 uv

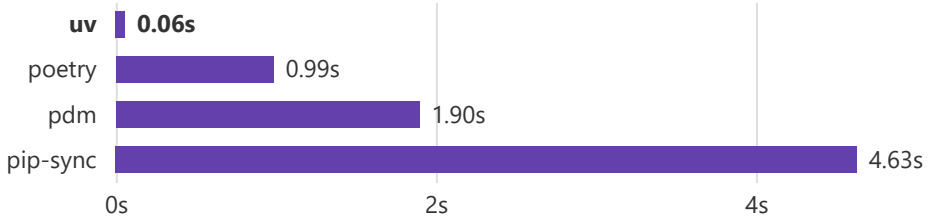
> 为啥我找不到一个完整的关于 uv 的中文文档或者教程。
>
> 是 pip 的黑恶势力太强大了吗？不，应该只是人们对于接受未知的事物抱有恐惧心理。
>
> 记录我的探索过程和路线，所以会超级乱。

什么是 uv?

你把一个立方体的表面沿着一条线剪开，把所有的面展开，就得到了 uv 映射。你可以利用 uv 贴图来给模型贴上纹理。

...

呸呸呸，这里我们要讲的 uv，是一个包管理器。<br



这是 uv 和 pip 的安装速度对比。

我们将在这里探索到一些 uv 的基础用法，取代 pip 的几种方式。以及会在后面尝试编译我自己的第一个 python package。（能够从 Github 安装并且运行即可）

呃另外，你可能会对我的记录方式感到不舒服，或者看得很难受，我理解，因为我按照我探索的路线写的，想到哪里实验一下，然后写下来。这也是我学的整个思路了。

后续可能会整理一个用法。

Refrence:

- 包管理工具 UV 使用指南：全面替代 conda
- UV 官方文档

uv.lock + pyproject.toml vs requirements.txt

uv.lock 是在手动运行 `uv lock` 后生成的，根据 `pyproject.toml`，并且一般情况下不应该被手动修改：

```
uv.lock is a human-readable TOML file but is managed by uv and should not be
edited manually.
uv.lock 是一个人类可读的 TOML 文件，但由 uv 管理，不应手动编辑。
```

一开始我以为 uv.lock 可以替代 pyproject.toml 的 python dependency 配置，后来发现是增强了。

用 `uv add package` -> `uv lock` -> `uv run` / `uv sync` -> `uv run`：

可以快速修改 `dependency` 并且验证是否兼容。

同时也支持从 Github 仓库直接安装第三方库，让包依赖处理非常精细。

中途更新 uv.lock 也是先更新 `pyproject.toml` 然后再 `uv lock`.

此外，uv.lock 给人满满的安全感，如果你特地去看过 uv.lock 的内容，它精确到 source 从哪里安装。相比起来 requirements.txt 就非常潦草了。

> Unlike the pyproject.toml, which is used to specify the broad requirements of your project, the lockfile contains the exact resolved versions that are installed in the project environment. This file should be checked into version control, allowing for consistent and reproducible installations across machines.
> 与用于指定项目总体要求的 pyproject.toml 文件不同，lockfile 文件包含项目环境中安装的精确解析版本。该文件应检查到版本控制中，以便在不同机器上进行一致和可重复的安装。

对于`pyproject.toml`+`uv.lock`的形式，人类可阅的部分是`pyproject.toml`,而`uv.lock`给机器阅读，而对于`pip freeze`生成的 requirements 来说，人也看不懂，机器也很头疼。

如果通过`uv add`的形式，每个包的版本和库都是一点点搭建和确定起来的，而不是直接 freeze 的。

pyproject.toml 中会列出必须的包的以及版本，而依赖包会自动下载。这也是建议在`uv add`的时候只加入必要包而非依赖包。

而`pip freeze`的 requirements.txt 全是依赖包，根本不是给人读的。

有时候我会考虑手写依赖包，而手写 requirements.txt 却很考验耐心。

而当我懒得的时候：

我提供的 requirements.txt 是这样的:

```
pyaml
gradio
```

而且你必须重新创建一个干净的环境然后`pip install -r requirements.txt`才能验证这个是否完整。

相对于`uv add` -> `uv lock` -> `uv run script`所有的依赖都可以快速确定是否支持环境运行。因为`uv run`只根据`uv.lock`运行。

以及如果我用上述手写的requirements.txt（无版本标注）,我确定我的使用者会在三个月后的某一天发现好像某些 api 被弃用或者最新版本的互相不兼容?

在 windows 上，它会下载最新版本的 pyaml 和 gradio。如果我很久没更新，大概率出问题。

如果在 linux 上面，这个就更离谱了，它会把所有版本的 pyaml 和 gradio 都下载一遍。我不清楚是不是 bug，反正它下了至少 5 个版本的 gradio，最后安装了几个我不清楚。

机缘巧合之下我决定入门 uv,从它的官方文档，下这个决定，也让我不得不克服之前的心理阴影，就是问 gpt 啥也不懂，反而被误导了好多天。

它让我用`uv pip install uv.lock`.....我实在是。。。

不过在开始前，我还是需要自己摸一遍 uv，毕竟这个东西都找不到齐全点的中文文档。是 pip 的黑恶势力太强大了吗？不，应该只是人们对于未知的事物具有恐惧心理。我挺早就想了解一下，但是拖到现在，还是机缘巧合。

安装 uv 以及最终建议。

https://docs.astral.sh/uv/getting-started/installation/

我最终建议是全局安装 uv + 使用 pyproject.toml + uv.lock 的方式来管理项目（python lib）。

具体参考:

https://github.com/yutto-dev/yutto

https://github.com/MrXnneHang/yutto-uiya

或者图方便只是用来加速安装也可以简单用 uv+conda 来替代 pip+conda。用法上，就是把所有的:

`pip install` -> `uv pip install`.

从 uv run 开始： uv run python/yutto/... 的可执行文件位于哪里呢?

如果不考虑那个人博客里写的 uvv 那种全局做法，正常的 uv 的环境位于项目根目录的`.venv`下方。

写上 `uv run` 的就相当于之前在 conda 指定虚拟环境下可以直接用 python/yutto 去访问我们虚拟环境中的可执行文件。

python 一般在 `.venv/bin` 下方。

独立使用 uv 创建一个 venv。

值得一提。在 linux 上面安装 miniconda 之初，会让我们选择是否开启`auto_activate_base`，即每次进入终端的时候是否会默认激活 base 环境。当我们尝试对 uv 进行独立使用的时候，应该考虑先禁用掉这个选项:

不然你会发现，创建的 venv 都是当前虚拟环境的 python 版本。

```
conda config --set auto_activate_base false
```

然后新开一个终端，就不会自动激活 base 环境了。

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv venv
Using CPython 3.13.0
Creating virtual environment at: .venv
Activate with: source .venv/bin/activate
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ ls .venv/
bin  CACHEDIR.TAG  lib  lib64  pyvenv.cfg
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ ls .venv/bin/
activate      activate.csh   activate.nu   activate_this.py  pydoc.bat  python3
activate.bat  activate.fish  activate.ps1  deactivate.bat     python
python3.13
```

似乎直接创建了一个 3.13 的 python 基础环境。

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run python tmp.py
Hello UV!

xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run tmp.py
Hello UV!

xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run python
Python 3.13.0 (main, Oct 16 2024, 03:23:02) [Clang 18.1.8 ] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

可以看见似乎正常工作。

而且，我在创建环境的时候，速度快的离奇？半秒？

似乎是这个原因：

```
If Python is already installed on your system, uv will detect and use it without configuration. However, uv can also install and manage Python versions. uv automatically installs missing Python versions as needed — you don't need to install Python to get started.
如果系统中已经安装了 Python，uv 将检测并使用它，而无需配置。uv 会根据需要自动安装缺失的 Python 版本，因此无需安装 Python 即可开始使用。
```

参见:[Install Python]

我们也会利用这个特性来直接从 conda 中直接获取我们需要的 python 版本，具体见下一节。

如何安装指定版本的 python?

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv python install 3.12
Installed Python 3.12.9 in 53.73s
 + cpython-3.12.9-linux-x86_64-gnu
```

这个又安装了一遍。根据上面可以判断是因为我系统中不具有 3.12 的 python 版本。

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run python
Python 3.13.0 (main, Oct 16 2024, 03:23:02) [Clang 18.1.8 ] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()

xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run python3.12
Python 3.12.4 (main, Jul  9 2024, 09:31:23) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>

xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run python3.11
error: Failed to spawn: `python3.11`
  Caused by: No such file or directory (os error 2)

xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ ls .venv/bin/
activate      activate.csh   activate.nu   activate_this.py  pydoc.bat  python3
activate.bat  activate.fish  activate.ps1  deactivate.bat    python
python3.13
```

可以看到，它安装 3.12 似乎并不是安装在`.venv`下.当我删除了`.venv`之后，再次运行`uv run python3.12`，它依然可行。说明它是全局安装的。

这其实存在一些可怕的隐患，全局的 python 总是会存在一些可以钻的漏洞。比如我使用`uv run python -m pip install matplotlib`。我就可以直接污染全局环境。

所以，这里我个人觉得更为可行的做法是，利用 conda 来管理这种 3.10,3.11,3.12 的环境，当我们需要的时候，从 conda 那边复制环境到./venv.后来我们所有的环境更改都针对./venv。

为 venv 指定 python 版本：

因为相对于系统全局的 python， conda 的虚拟环境更为安全并且容易管理。~~我们甚至可以自定义不同的起始包，比如起始 selenium,webdrier 来做爬虫工作的，或者起始 numpy,matplotlib 来做数据分析工作的。然后在用到的时候复制到.venv 中做基础环境。~~

我后半段的想法是错的，因为`uv venv`只会继承 python，不会继承 package。所以，这个想法是不可行的。下一个内容会证明这一点。

这点上我和那个作者不谋而合：

```
conda activate 11
uv venv --seed -p 3.11
conda deactivate
uv pip install -r requirements.txt
source ./.venv/bin/activate
python main.py
```

参数解析:

```
-p, --python <PYTHON>
    The Python interpreter to use for the virtual environment. [env: UV_PYTHON=]
--seed
    Install seed packages (one or more of: `pip`, `setuptools`, and `wheel`) into
the virtual
    environment [env: UV_VENV_SEED=]
```

我发现并没有我想象的`复制`发生，可能并非复制环境，只是继承了 Python。

它利用了一点，就是在`uv venv`运行的时候，默认检测系统首选 python 作为 venv 的 python。而当我们`conda activate 11`之后，我们默认运行`python XXX`运行的是 conda 的 python。所以，`uv venv`就会继承这个 python。

至于它会不会继承 package.来测试一下。

测试 venv 创建是否会继承虚拟环境的 package

先说结论，不会，只能继承 python。

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ conda activate numpy
(numpy) xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ python
Python 3.10.16 (main, Dec 11 2024, 16:24:50) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> exit()
```

```
(numpy) xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ conda deactivate
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ conda activate numpy

(numpy) xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ python
Python 3.10.16 (main, Dec 11 2024, 16:24:50) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> numpy.__version__
'2.2.2'
>>> exit()

(numpy) xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv venv -p 3.10 --seed
Using CPython 3.10.16 interpreter at:
/home/xnne/miniconda3/envs/numpy/bin/python3.10
Creating virtual environment with seed packages at: .venv
 + pip==25.0.1
 + setuptools==75.8.0
 + wheel==0.45.1
Activate with: source .venv/bin/activate

(numpy) xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ conda deactivate
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run python
Python 3.10.16 (main, Dec 11 2024, 16:24:50) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'numpy'
>>>
```

`Using CPython 3.10.16 interpreter at: /home/xnne/miniconda3/envs/numpy/bin/python3.10` 这说明确实是用到了我们的 conda 环境的 python。但是也只用到了`Interpreter`.

使用`uv run pip install`和`uv pip install`的区别:

`uv run pip install` vs `pip install`:

先说结论，`uv run pip install`会安装到.venv 中,和 uv run python -m pip install 是一个效果。

而前面我们猜测，这么做和正常使用 pip install 几乎没有区别，只不过在我们使用 conda 的时候，虚拟环境中使用 pip install 并不会污染全局环境，因为前面提到当我们激活虚拟环境，虚拟环境的优先级高于系统环境，所以 pip install 会安装在虚拟环境中。

而我们的 uv 只创建了一个局部的.venv。

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run python
Python 3.10.16 (main, Dec 11 2024, 16:24:50) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'numpy'
>>> exit()
```

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run pip install numpy
Collecting numpy
  Using cached numpy-2.2.3-cp310-cp310-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (62 kB)
Using cached numpy-2.2.3-cp310-cp310-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (16.4 MB)
Installing collected packages: numpy
Successfully installed numpy-2.2.3

xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run python
Python 3.10.16 (main, Dec 11 2024, 16:24:50) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> numpy.__version__
'2.2.3'
>>>

xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ ls .venv/lib/python3.10/site-packages/
_distutils_hack           numpy.libs            __pycache__
_virtualenv.py
distutils-precedence.pth  pip                   setuptools
wheel
numpy                     pip-25.0.1.dist-info  setuptools-75.8.0.dist-info
wheel-0.45.1.dist-info
numpy-2.2.3.dist-info     pkg_resources         _virtualenv.pth
```

而当我们直接运行`pip install`时，会发现我的系统环境下实际上并没有`pip` \=-=/.

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ pip uninsatll numpy
bash: pip: 未找到命令
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ pip install numpy
bash: pip: 未找到命令
```

这说明了，uv 并不像 conda 那样子直接为我们隔绝了系统环境，而是在当前目录下创建了一个.venv，然后所有`uv run xxx`都是在这个.venv 下进行操作。

`uv pip install` vs `uv run pip install`:

先说结论，两者都是对`.venv`下的环境操作。

但是，`uv run pip install`是用 pip 来安装环境，而`uv pip install`是用 uv 来安装环境。

我们简单看一下速度比较：

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ time uv run pip install numpy
Collecting numpy
  Using cached numpy-2.2.3-cp310-cp310-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (62 kB)
Using cached numpy-2.2.3-cp310-cp310-
manylinux_2_17_x86_64.manylinux2014_x86_64.whl (16.4 MB)
Installing collected packages: numpy
Successfully installed numpy-2.2.3

real    0m3.090s
user    0m1.770s
```

```
sys     0m0.163s

xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv pip uninstall numpy
Uninstalled 1 package in 55ms
 - numpy==2.2.3

xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ time uv pip install numpy
Resolved 1 package in 16ms
Installed 1 package in 31ms
 + numpy==2.2.3

real    0m0.064s
user    0m0.007s
sys     0m0.054s
```

好家伙，速度不是一个量级的。

也就是说，我们平时安装环境的时候多用`uv pip install`即可.

稍微总结一下如何用 uv 来取代 pip 的工作。

如果你是用 miniconda ,并且也只是想要用 uv 来加速一下包安装。那么，每次在创建环境后:

```
pip install uv
uv pip install -r requirements.txt/packagename
```

这样应该是最简单的。

如果你和我一样有做整合包的习惯，希望能够把环境打包在项目根目录。那么可以尝试:

```
conda activate 10
uv venv --seed -p 3.10
conda deactivate
uv pip install -r requirements.txt/packagename
```

这个 uv 需要全局安装。参见上面的链接。

以及，当我们 uv 无法安装某个包的时候，我们就需要重新用 pip,这也是我上面花费大量时间来聊`uv pip`和`uv run pip`的原因。

例如:

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run __main__.py
Traceback (most recent call last):
  File "/home/xnne/code/yutto-uiya/src/uiya/__main__.py", line 3, in <module>
    import gradio as gr
  File "/home/xnne/code/yutto-uiya/src/uiya/.venv/lib/python3.10/site-
packages/gradio/__init__.py", line 3, in <module>
    import gradio._simple_templates
  File "/home/xnne/code/yutto-uiya/src/uiya/.venv/lib/python3.10/site-
packages/gradio/_simple_templates/__init__.py", line 1, in <module>
    from .simpledropdown import SimpleDropdown
  File "/home/xnne/code/yutto-uiya/src/uiya/.venv/lib/python3.10/site-
packages/gradio/_simple_templates/simpledropdown.py", line 7, in <module>
    from gradio.components.base import Component, FormComponent
```

```
  File "/home/xnne/code/yutto-uiya/src/uiya/.venv/lib/python3.10/site-
packages/gradio/components/__init__.py", line 1, in <module>
    from gradio.components.annotated_image import AnnotatedImage
  File "/home/xnne/code/yutto-uiya/src/uiya/.venv/lib/python3.10/site-
packages/gradio/components/annotated_image.py", line 14, in <module>
    from gradio import processing_utils, utils
  File "/home/xnne/code/yutto-uiya/src/uiya/.venv/lib/python3.10/site-
packages/gradio/processing_utils.py", line 120, in <module>
    sync_client = httpx.Client(transport=sync_transport)
  File "/home/xnne/code/yutto-uiya/src/uiya/.venv/lib/python3.10/site-
packages/httpx/_client.py", line 697, in __init__
    self._mounts: dict[URLPattern, BaseTransport | None] = {
  File "/home/xnne/code/yutto-uiya/src/uiya/.venv/lib/python3.10/site-
packages/httpx/_client.py", line 700, in <dictcomp>
    else self._init_proxy_transport(
  File "/home/xnne/code/yutto-uiya/src/uiya/.venv/lib/python3.10/site-
packages/httpx/_client.py", line 750, in _init_proxy_transport
    return HTTPTransport(
  File "/home/xnne/code/yutto-uiya/src/uiya/.venv/lib/python3.10/site-
packages/httpx/_transports/default.py", line 191, in __init__
    raise ImportError(
ImportError: Using SOCKS proxy, but the 'socksio' package is not installed. Make
sure to install httpx using `pip install httpx[socks]`.

xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv pip install https[socks]
  × No solution found when resolving dependencies:
  ╰─▶ Because there are no versions of https[socks] and you require https[socks],
we can conclude that
      your requirements are unsatisfiable.
```

我无法使用`uv pip install`来安装`httpx[socks]`，这个时候就得用回`uv run pip install`了。

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run pip install httpx[socks]
Requirement already satisfied: httpx[socks] in ./.venv/lib/python3.10/site-
packages (0.28.1)
Requirement already satisfied: anyio in ./.venv/lib/python3.10/site-packages
(from httpx[socks]) (4.8.0)
Requirement already satisfied: certifi in ./.venv/lib/python3.10/site-packages
(from httpx[socks]) (2025.1.31)
Requirement already satisfied: httpcore==1.* in ./.venv/lib/python3.10/site-
packages (from httpx[socks]) (1.0.7)
Requirement already satisfied: idna in ./.venv/lib/python3.10/site-packages (from
httpx[socks]) (3.10)
Collecting socksio==1.* (from httpx[socks])
  Using cached socksio-1.0.0-py3-none-any.whl.metadata (6.1 kB)
Requirement already satisfied: h11<0.15,>=0.13 in ./.venv/lib/python3.10/site-
packages (from httpcore==1.*->httpx[socks]) (0.14.0)
Requirement already satisfied: exceptiongroup>=1.0.2 in
./.venv/lib/python3.10/site-packages (from anyio->httpx[socks]) (1.2.2)
Requirement already satisfied: sniffio>=1.1 in ./.venv/lib/python3.10/site-
packages (from anyio->httpx[socks]) (1.3.1)
Requirement already satisfied: typing_extensions>=4.5 in
./.venv/lib/python3.10/site-packages (from anyio->httpx[socks]) (4.12.2)
Using cached socksio-1.0.0-py3-none-any.whl (12 kB)
Installing collected packages: socksio
```

```
Successfully installed socksio-1.0.0

xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv run __main__.py
* Running on local URL:  http://127.0.0.1:7860

To create a public link, set `share=True` in `launch()`.
```

solve.

uv lock,锁死环境。

运行这个我们首先需要一个`pyproject.toml`。

这是一个 python+ruff 的简单配置: https://github.com/MrXnneHang/yutto-uiya/blob/gradio-webui/pyproject.toml

之后你运行:

```
xnne@xnne-PC:~/code/yutto-uiya/src/uiya$ uv lock
Using CPython 3.13.0
Resolved 1 package in 4ms
```

你会发现，好像什么包都没锁定，假如，你的`pyproject.toml`中没有提供:

```
[dependency-groups]

# dev = [
#   "pyaml>=25.1.0",
#   "gradio>=5.16.1",
# ]
```

但是如果需要手写 pyproject.toml 似乎太累了，能不能直接从.venv 中提取呢?

> 滞后导出不是好习惯。就好像 `pip freeze`在最后导出时局限性很大，导出很多依赖库让依赖变得不可读。
> 你应该逐步探索需要的具体第三方库，以及它的版本。至于依赖库，就不用写了。
> 最后你把它写在`pyproject.toml`中，并且 `uv lock ,uv sync(可选),uv run` 进行确认可运行和兼容。

lock env 之后可能出现的问题。 | .venv 目录变更 | 包名称不同

比如我`__main__.py`和 requirements.txt 不在根目录。我一开始创建的`.venv`位于`./src/uiya`(相对于根目录。)

而`uv.lock`需要`pyproject.toml`,`pyproject.toml`需要在根目录下。然后 uv.lock 会自动搜索父文件夹下的`pyproject.toml`并且又重新创建了一个`.venv`。

也就是说一开始我的`.venv`位于`./src/uiya`，现在变成了`./`。

这个最初给我带来了一些困扰，因为我发现我无法通过`uv run`来使用我子文件夹下的`.venv`，并且前面提到，uv 无法为我安装`httpx[socks]`，所以我需要手动安装。

所以，这里的做法最好是，在一开始就创建 pyproject.toml.那么`uv run`,`uv pip install`针对的对象都是根目录（和 pyproject.toml 相同目录）下的`.venv`。

这是一个简单的配置:

```
name = "yutto-uiya"
version = "1.0.2"
description = ""
readme = "README.md"
requires-python = ">=3.10"
dependencies = [
    "gradio==5.16.1",
    "pyaml==25.1.0",
    "socksio==1.0.0",
    "yutto",
]
authors = [{ name = "MrXnneHang", email = "XnneHang@gmail.com" }]
keywords = []
license = { text = "MIT" }
classifiers = [
  "Operating System :: OS Independent",
  "License :: OSI Approved :: MIT License",
  "Typing :: Typed",
  "Programming Language :: Python",
  "Programming Language :: Python :: 3",
  "Programming Language :: Python :: 3.10",
  "Programming Language :: Python :: 3.11",
  "Programming Language :: Python :: 3.12",
  "Programming Language :: Python :: 3.13",
  "Programming Language :: Python :: Implementation :: CPython",
]

[project.urls]
Homepage = "https://github.com/MrXnneHang/yutto-uiya"
Documentation = "https://github.com/MrXnneHang/yutto-uiya"
Repository = "https://github.com/MrXnneHang/yutto-uiya"
Issues = "https://github.com/MrXnneHang/yutto-uiya/issues"

[project.scripts]
uiya = "uiya.__main__:main"

[tool.uv.workspace]
members = ["packages/*"]

[tool.uv.sources]
yutto = { git = "https://github.com/MrXnneHang/yutto.git", rev = "depndency-
adjust" }

[tool.pyright]
include = ["src/uiya", "tests"]
pythonVersion = "3.10"
typeCheckingMode = "strict"

[tool.hatch.build.targets.wheel]
packages = ["src/uiya"]

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

值得注意的是，dependency 和 source 这些并不需要手写。

这些可以通过`uv add gradio==5.16.1`来添加。

也支持 github 仓库比如:

`uv add git+https://github.com/MrXnneHang/yutto.git@depndency-adjust`

所以， uv 实际上是推荐配合 `pyproject.toml` 使用的，如果希望用来管理包版本，因为`uv lock`必须在存在`pyproject.toml`的情况下才可以运行。

另外，虽然 dependency 大部分可以直接从 requirements 里面搬运迁移。但是应该注意一些包命名不同的情况，比如`socksio`在 pip 里面的命名是`httpx[socks]`。

但是一旦你的`pyproject.toml`和`uv.lock`都做好了，那么给别人运行的话只需要`uv run script`即可。

另外，锁死的版本可以让别人不用考虑环境版本兼容问题，友好你我他。

在你需要更新 uv.lock 的时候，你需要先`uv add packagename==version`，然后`uv lock`。

我建议锁死版本不用留余地。此外，uv lock 后，你也不必运行`uv pip install`来手动装包了，你可以直接`uv run script`，它会根据`uv.lock`自动配置包，只要你能这么运行起来，那么别人也可以。当然一些 windows 和 Linux 跨越时用了 win32api 的不算。

为了跨系统友好，应该避免用到系统依赖的底层 api.

其实到了这一步，uv 基本上已经快毕业了，为啥我突然这么自信，因为我经过了一番实战，把我原本一个结构混乱的项目整理并且打包了成了可以直接从 Github 上安装的形式。

https://github.com/MrXnneHang/yutto-uiya

关键在于`pyproject.toml`和`uv.lock`。其中 uv.lock 是根据 pyproject.toml 自动生成的。

实际上动手比光在这看要快得多。而且理解会突然深入，因为你会遇到各种各样的问题。抄作业的过程很痛苦，抄完了却非常舒适。

顺带一提，我抄的是:

https://github.com/yutto-dev/yutto

uv 运行脚本。

可执行文件

前面提到，可以`uv run pip`,`uv run python`等等来运行`bin`下方的内容。

这里不过多赘述了。

Python 脚本。

这里我会分为有`pyproject.toml`和没有`pyproject.toml`两种情况。

有`pyproject.toml`+`uv.lock`的情况

这个运行的话，实际上是根据 `uv.lock` 的内容来运行的，它会在`uv.lock`同级目录自动创建一个`.venv`，然后在这个`.venv`自动配置环境并且运行。

所以，这个的模式一般是：

运行脚本：

```
git clone XXX
uv run XXX
```

什么都不用管，要是跑不起来绝对不是我的问题，绝对是维护者的问题。

如果维护者的话，一般就是这样：

```
uv add package==version
uv lock
uv run XXX # 如果成功，就上传，否则就考虑调整package版本
```

最后调试成功，如果你是个好人，你会选择把 uv.lock 一起上传，并且跟你的使用者说，你可以直接`uv run XXX`运行。

因为，大部分的人真的不清楚 uv 这东西咋用。他们会找半天 requirements.txt,没找到，然后尝试了半天`uv pip install uv.lock`...

然后怎么也搞不清`uv run pip`和`uv pip`。我以前就是这样的，甚至都有心理阴影了。

没有`pyproject.toml`+`uv.lock`的情况

这个比较类似于直接把 pip 换成`uv pip`进行加速。

其实目前我更加推荐用`pyproject.toml` + `uv.lock` 进行管理。

但是你也依然可以使用`requirements.txt`.

这种情况，没必要考虑`.venv`,直接在 conda 里面装 uv 和环境就好了。

原理：**如果你激活了 conda 环境，那么`uv pip install` 等等也默认会装在 conda 环境里。**

当然你实在需要一个独立的`.venv`，那么也可以考虑这样的做法：

```
conda activate 11 # python 3.11 的基础环境
uv venv --seed -p 3.11
conda deactivate # 前面提到，必须退出环境否则会装在conda里。
uv pip install -r requirements.txt
```

这样的话，你的`.venv`就是独立的了。但是它有个局限性，就是，你似乎只能创建在当前目录下并且在此访问它？如果运行 uv run XXX 的时候。

它并不像有`pyproject.toml`+`uv.lock`的情况，即使`.venv`在根目录，你也可以在项目目录下的任何地方访问它。

所以我个人只推荐`pyproject.toml`+`uv.lock`的模式和`conda + uv`取代`conda + pip`的模式。

当我打包好项目后，如何反复编译?

最开始我运行的方式是:

```
uv run __main__.py
```

这个适用于开发阶段，反复调试。

后来进入编译调试阶段。即我希望可以直接运行我的可执行文件。这里是`uiya`。

我的做法是很蠢的,上传 github,然后用:

```
conda create -n test-uiya python=3.10
conda activate test-uiya
uv pip install git+https://github.com/MrXnneHang/yutto-uiya.git@gradio-webui
```

然后每次更新代码后我都重新`uv pip install`一遍,好在它会检测最新分支且不会被 cache 影响不需要 uninstall.

它确实成功了，在我第一次安装成功并且运行起来的时候我非常兴奋。

但是后来这个方式让我非常疲惫。

因为每次都需要先把代码上传到 github，然后再重新安装一遍。这个过程非常繁琐。

我希望可以一行代码就自动更新。

这里就得提到本地编译和更新了。上面那个方法我就称他为远程编译吧=-=。

本地编译生成可执行程序。

emmmmm,我应该咋说呢?

好像不需要任何指令哈。

万能的`uv run`。

你只需要`uv run uiya`就可以了。

它会根据你最新的代码运行，因为这个时候实际上应该处于一种未编译的状态，而你在`pyproject.toml`中配置了`uiya = "uiya.__main__:main"`，所以它会自动运行你的`__main__.py`。
`

而特意编译好像也没啥必要哈，就像 vue 可以实时预览一样，这样的效率总会比编译一下再看一下要高。

我从最初就疑惑的 uv sync

我以为它会在我们的包管理中起到很重要的作用，但是实际上好像不用也可以?

`sync`的翻译是同步，同步的是环境。

可能是在`uv lock`后同步一下？但是我记得好像`uv run`每次都会检查`uv.lock`的内容。

我们这里来看一下一个用例:

```
如果你想要本地调试，最佳实践是从 GitHub 上下载最新的源码来运行

git clone git@github.com:yutto-dev/yutto.git
cd yutto/
uv sync
uv run yutto -v
```

https://github.com/yutto-dev/yutto/blob/main/CONTRIBUTING.md

可以推测出来这个 sync 是用来安装环境的，但是现在似乎可以直接`uv run yutto`了。

也就是说，`uv sync`目前应该已经集成到了`uv run`的步骤中了，所以我们不需要手动运行`uv sync`。

当然偶尔碰到奇怪的环境问题可以`uv sync`来瞅瞅咋回事。

按照我们之前的实验，`uv run`生成的`.venv`和`pyproject.toml`以及`uv.lock`同一目录。

我们再测一下`uv sync`：

```
xnne@xnne-PC:~$ git clone git@github.com:MrXnneHang/yutto-uiya.git
正克隆到 'yutto-uiya'...
remote: Enumerating objects: 898, done.
remote: Counting objects: 100% (57/57), done.
remote: Compressing objects: 100% (39/39), done.
remote: Total 898 (delta 23), reused 36 (delta 12), pack-reused 841 (from 1)
接收对象中: 100% (898/898), 389.53 KiB | 429.00 KiB/s, 完成.
处理 delta 中: 100% (512/512), 完成.

xnne@xnne-PC:~$ cd yutto-uiya/

xnne@xnne-PC:~/yutto-uiya$ uv sync
Using CPython 3.13.0
Creating virtual environment at: .venv
Resolved 64 packages in 2ms
      Built yutto-uiya @ file:///home/xnne/yutto-uiya
Prepared 1 package in 1.49s

xnne@xnne-PC:~/yutto-uiya$ ls .venv/
bin/          CACHEDIR.TAG  .gitignore  lib/          lib64/        pyvenv.cfg
```

这里看到，它确实是在装环境，并且装在了根目录的`.venv`下。

推测正确。

不过这里有个神奇的一点，是 python 版本，它会默认找 source 里面排最前面的，如果要控制 python 版本的话，可以参考前面`conda`激活再退出的做法。

```
(10) xnne@xnne-PC:~/yutto-uiya$ uv sync -p 3.10
Using CPython 3.10.16
```

当然，似乎也没必要用到 conda,因为 uv 有自己的全局解释器，如果指定的版本没有会自动下载补上。就是位置比较诡异。

如果环境有问题，可以考虑删掉`.venv`然后重新`uv sync`,要是还不行，那就去找维护者=-=。

恭喜， uv 毕业。

这个似乎长的离谱，但是我就写了两天。大部分都是废话哈。

有时间可能会整理一个用法版本，但是我一直以来都只重探索不重整理。

所以..看缘分。