

用 Pyinstaller 模块将 Python 程序打包成 exe 文件（全网最全面最详细，万字详述）

目录

- 一、打包前置知识
 - 1.1 什么是 exe 可执行文件？
 - 1.2 为什么要将 Python 程序打包为 exe 可执行文件？
 - 1.3 为什么 Python 程序不能直接运行呢？
 - 1.4 我们用什么来打包 Python 文件呢？
 - 1.5 打包有哪种分类呢？
- 二、打包的方法
 - 2.1 简单打包
 - 2.1.1 前置
 - 2.1.2 直接打包
 - 2.1.3 稍微“加密”一下源代码
 - 2.2 一般的打包
 - 2.2.1 打开 Anaconda Prompt
 - 2.2.2 下载并安装 Pyinstaller 模块
 - 2.2.3 切换命令行的路径
 - 2.2.4 打包 Python 文件
 - 2.2.5 打包生成文件的位置
 - 2.3 虚拟环境下的打包
 - 2.3.0 先介绍几个 conda 命令
 - 2.3.1 创建虚拟环境
 - 2.3.2 安装需要的第三方包
 - 2.3.3 追求极致的exe大小（非必要步骤）
 - 2.4 多 Python 文件的打包
 - 2.4.1 生成spec文件
 - 2.4.2 编辑spec文件
 - 2.4.3 以spec文件进行打包
 - 2.5 包含资源文件的打包
 - 2.5.0 一点吐槽
 - 2.5.1 编辑spec文件
 - 2.5.2 修改文件打开函数
- 三、打包实战
 - 3.1 打包方式的选择
 - 3.2 打包全过程
 - 3.2.1 第一步：启动 Anaconda Prompt，切换至目标文件夹路径位置
 - 3.2.2 第二步：启动虚拟环境（我的是一个纯净的、第三方包只有 Pyinstaller 的环境）
 - 3.2.3 第三步：生成 spec 文件（我的 Pyinstaller 已经安装好了）
 - 3.2.4 第四步：引入 _py 模块（我的程序用到了大量 open 函数且涉及多文件）
 - 3.2.5 第五步：编辑 spec 文件
 - 3.2.6 第六步：打包项目（注意这里的对象是 spec 文件）
 - 3.2.7 第七步：检验打包效果
- 四、疑难解答与相关建议
 - 4.1 疑难解答
 - 4.1.1 模块找不到的错误 —— ModuleNotFoundError
 - 4.1.2 图形化程序运行没反应，但也没有报错
 - 4.1.3 语法错误 —— SyntaxError
 - 4.1.4 'pyinstaller' 不是内部或外部命令.....
 - 4.1.5 其他情况
 - 4.2 相关建议
 - 4.2.1 关于虚拟环境的建立
 - 4.2.2 关于项目文件的操作

注意：文章已在 2023/11/07 再次修改，用语更严谨，格式更规范，且新增一种打包的方式！👍

一、打包前置知识

1.1 什么是 exe 可执行文件 ？

exe 文件英文全名是 executable file，翻译为可执行文件（但它不等于可执行文件），可执行文件一般来说包含两种，文件扩展名为 .exe 的是其中的一种。正确的 exe 文件一般可以在 Windows 平台上直接双击运行！我们通常用的各种软件都是通过快捷方式打开的，而这个快捷方式的目标地址就是这个软件的一个 exe 文件。还有其他的可执行文件，这里就不详述了。

1.2 为什么要将 Python 程序打包为 exe 可执行文件？

众所周知，Python 程序的运行必须要有 Python 的环境，但是程序编出来是用的，如果是给别人用，而他/她的电脑上又没有 Python 程序运行的环境怎么办呢？总不能让他/她去安装一个吧？这时我们就要将 Python 程序打包为 exe 文件。这样，在 Windows 平台下，就可以直接运行该程序，不论有没有 Python 环境。不过呢，如果对方使用的是 Linux 系统，自带了符合版本要求的 Python 环境，那就不必打包，也更用不到 pyinstaller 了。

1.3 为什么 Python 程序不能直接运行呢？

Python 是解释性语言，它与 C 或者 C++ 等编译型语言不同，C 或者 C++ 都是要编译再运行的，（Windows 平台下编译产生的最终文件一般都是 exe 文件），Python 本质上只是对一段文本进行解释，类似于浏览器解析 html 文件，是不会产生任何可执行文件的。

1.4 我们用来打包 Python 文件呢？

一般我们都用 Python 的 Pyinstaller 模块进行打包，也有其他的打包模块，不过相比之下，Pyinstaller 的使用者最多，用起来也很简单（但效果并不一定是最好的，这里推荐一个效果可能更好的模块 —— Nuitka），因此本文就以 Pyinstaller 模块来打包 Python 程序。

1.5 打包有哪几种分类呢？

根据需要，下面的方法大家可以任选一种进行打包（我一般用第 1 个），不过新手的话建议全部都看一下哦。

- ① **简单打包**：操作最简单，成功可能性最高，不兼容的可能性最低，但**无法保护源代码**，只能保证可以无需 Python 环境即可运行
- ② **一般打包**：步骤最少，操作最简单，但是打包时间久，效果不理想（打包后文件太大，一般 100MB 以上）
- ③ **虚拟环境下的打包**：步骤稍多，操作略微复杂，但是打包快，效果好（打包后文件不大，一般 10MB 以内）
- ④ **多 Python 文件的打包**：步骤更多，操作更复杂，但是可以将多个 Python 文件都打包进去
- ⑤ **包含资源文件的打包**：步骤极为繁琐，操作非常复杂，但是可以把所有的文件都包含进去

二、打包的方法

2.1 简单打包

此方法非常适合于开源用户，如果你没有保护源代码的需求（无所谓源代码会不会被看到），那我**强烈强烈强烈推荐**你使用此方法，这也是我目前一直使用的方法！因为它实在是太简单了，而且几乎不会出现什么问题！

但是一定一定要注意，这种打包方式，源代码虽然无法被保护，但也不是说直接可以给别人看到的，我们可以对它稍微“加密”一下，后面会讲，非常简单的。

另外，此方法为 2023/11/07 添加到这篇文章中的，因此我不会对其像后面几个方法那样进行详述，如有在操作步骤上的问题，可参考后面方法的步骤。

2.1.1 前置

此方法推荐要在学会搭建虚拟环境的情况下操作，效果比较好。关于搭建虚拟环境，往文章下面看。

2.1.2 直接打包

搭建好虚拟环境后，安装所需的包，然后找到启动文件（你程序启动的那个 Python 文件），对其使用打包命令，打包完之后，直接将产生的 dist 文件中的 exe 移动工程目录的根目录下，用它代替启动文件即可，双击起飞！

但是要注意，把文件发给别人使用的时候，必须将整个项目文件夹发给别人，而不是一个 exe 文件。

2.1.3 稍微“加密”一下源代码

打包前运行一下项目，确保执行每个功能后关闭程序，将产生的每个 __pycache__ 文件夹中的文件移动到项目中去，代替对应名称的源代码（记得去除文件名中多余的部分）。然后，源代码文件就可以删除了（**记得这样做之前备份整个项目的源代码**）。

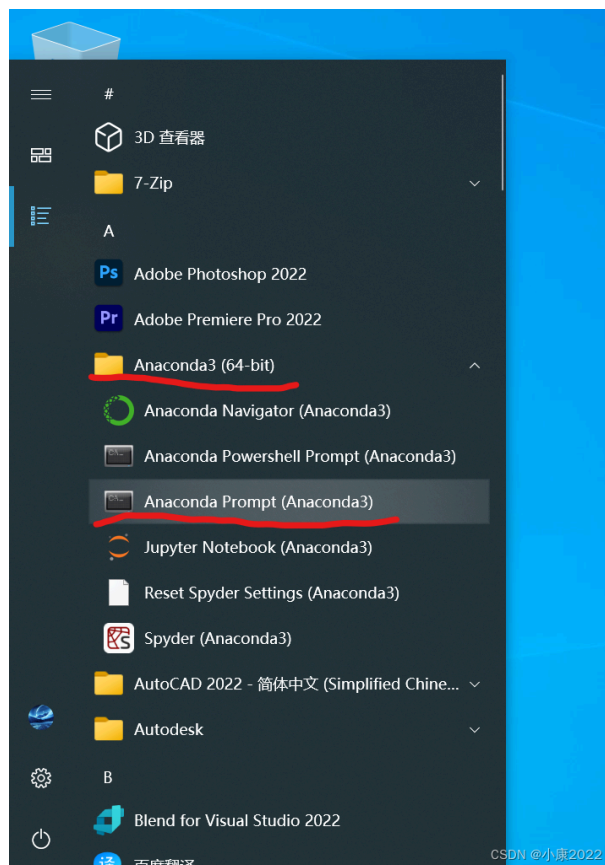
__pycache__ 中都是 pyc 文件，有兴趣的朋友可以了解一下。这不是真正的加密（可以很容易地被解密），pyc 文件都是 Python 解释器产生的字节码文件，但是这样别人也无法直接看到源代码了。

2.2 一般的打包

一般的打包方式，最简单，但是打包的成品有些许臃肿，不是特别推荐。

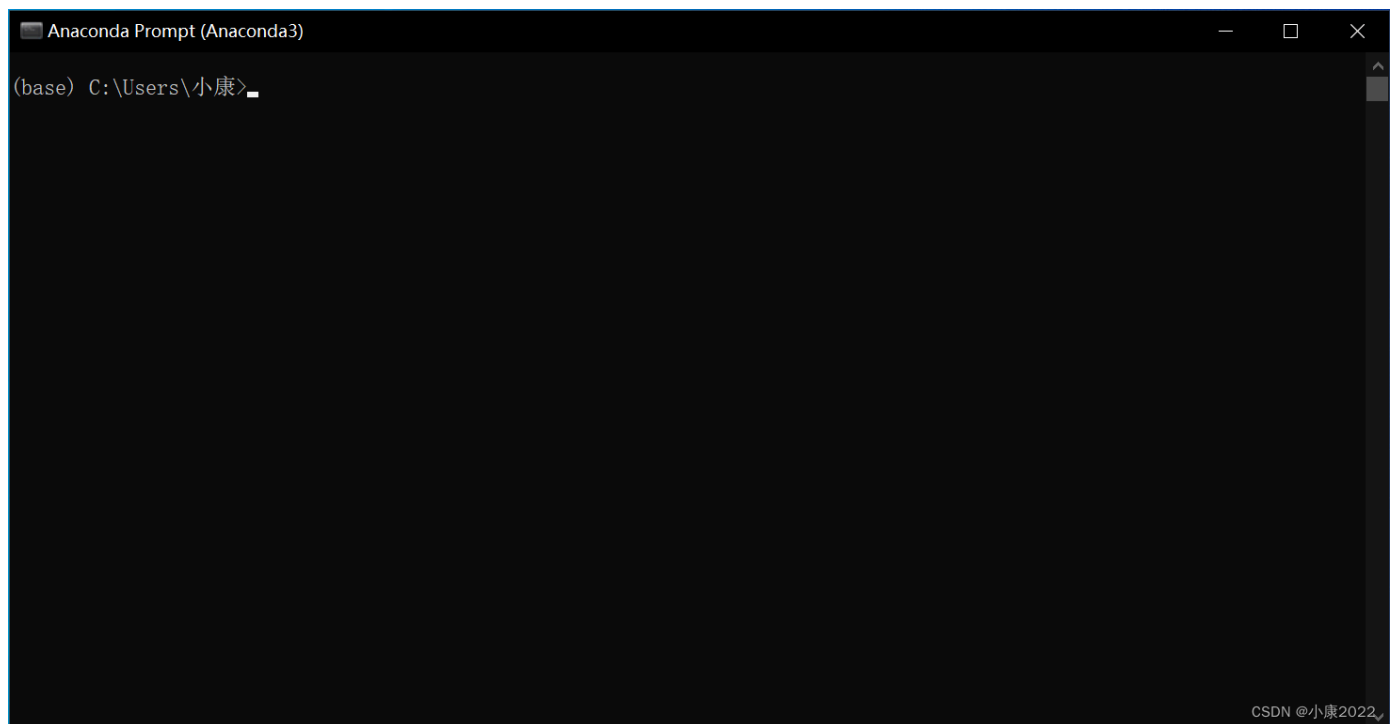
2.2.1 打开 Anaconda Prompt

如果你安装了 Anaconda 的 Python 集成环境的话，在菜单页面的所有应用里面可以看到 Anaconda 以及 Anaconda Prompt。



Anaconda

点进去就可以看见如下的界面:

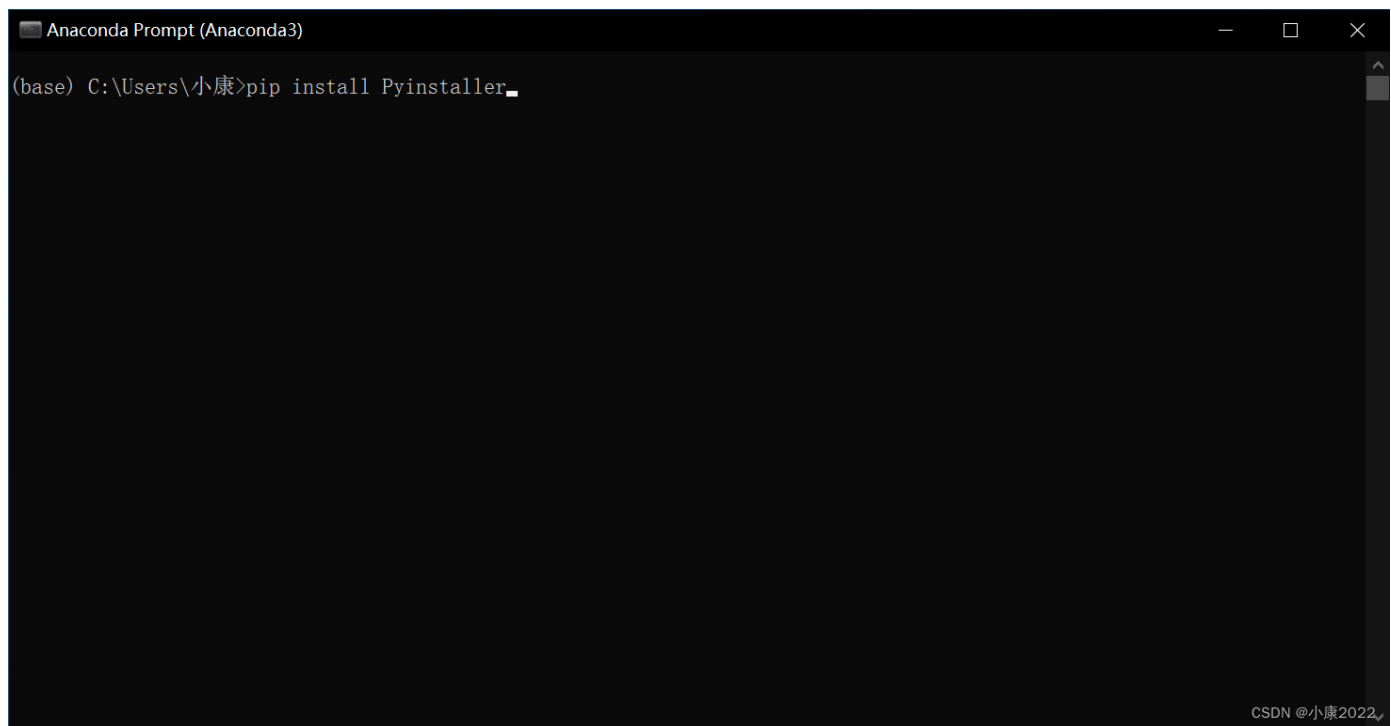


Anaconda Prompt

2.2.2 下载并安装 Pyinstaller 模块

这个用 pip 模块直接下载就行，直接就下载在本次需要打包的 Python 环境下（base 环境）

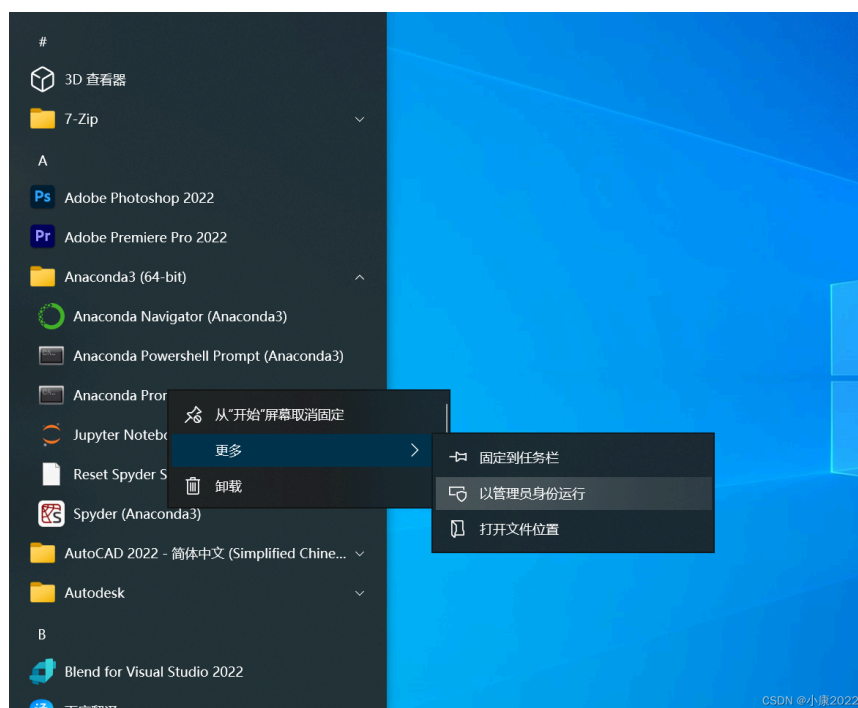
```
pip install Pyinstaller
```



pip insatl Pyinstaller

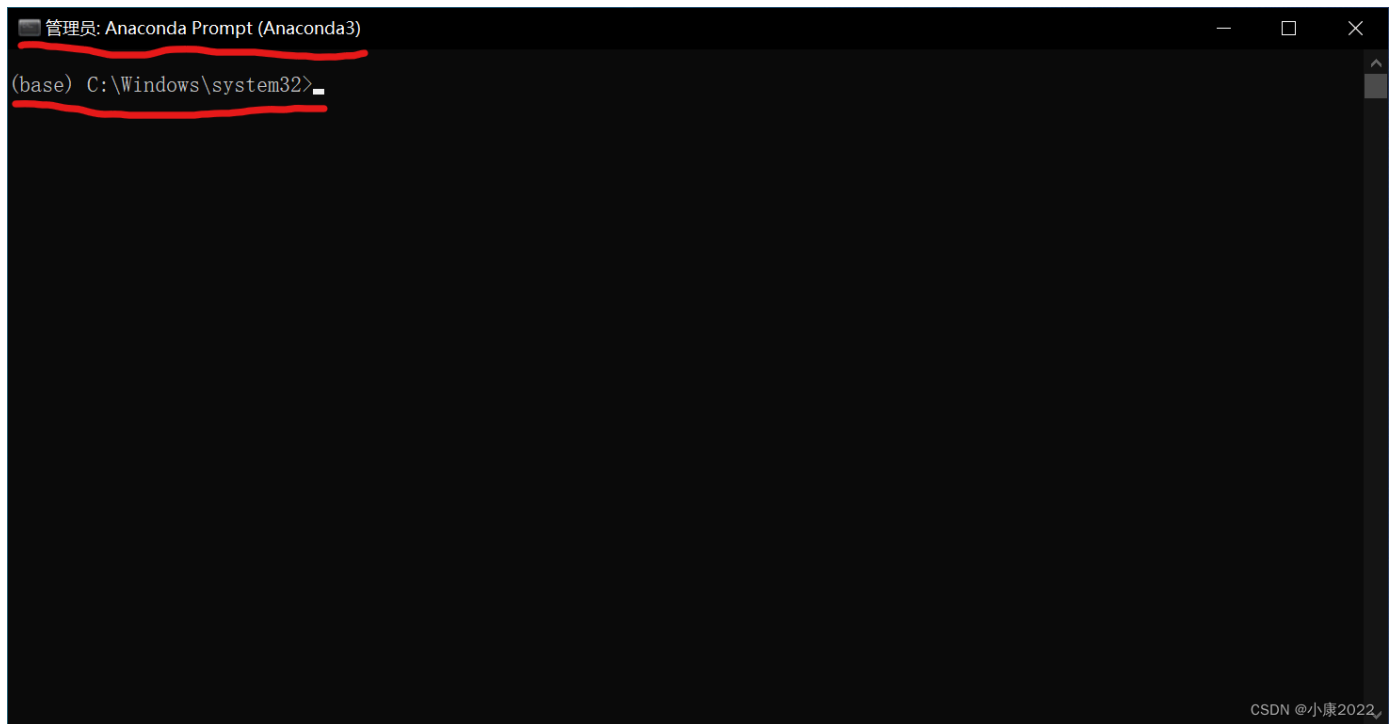
当然了，已经安装过 Pyinstaller 模块的可以跳过这一步。

如果出现什么疑难杂症，大概率是权限导致的问题，按照下面的方法重新打开 Anaconda Prompt 就好了。



管理员身份打开

此时，Anaconda Prompt 的显示文字会变成如下这样：



管理员身份运行

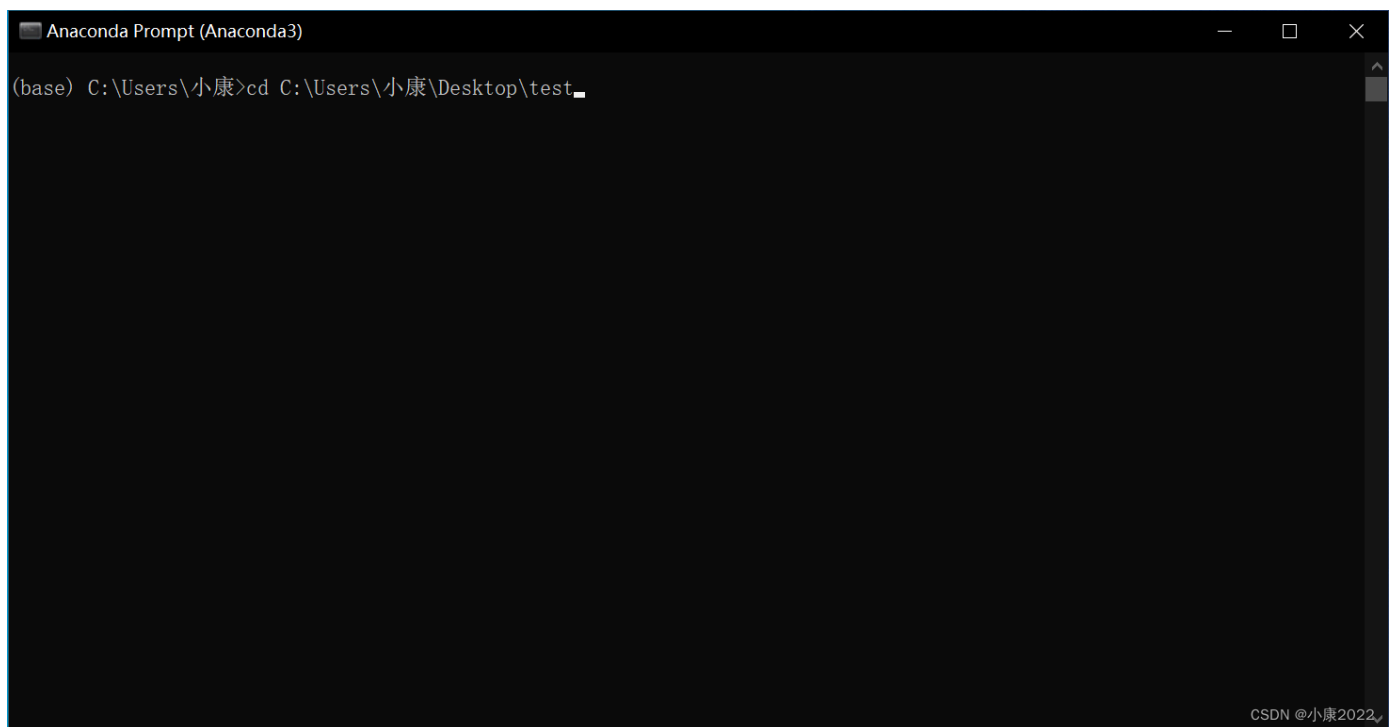
然后再 pip 就行，这样应该就没有什么问题了。

2.2.3 切换命令行的路径

因为你要打包的文件在对应的文件夹里面，而 Pyinstaller 一开始是不知道要打包的文件在哪里的，所以要直接切换命令行的路径到目标文件夹路径，使得后面的步骤中，Pyinstaller 可以找到对应的文件。

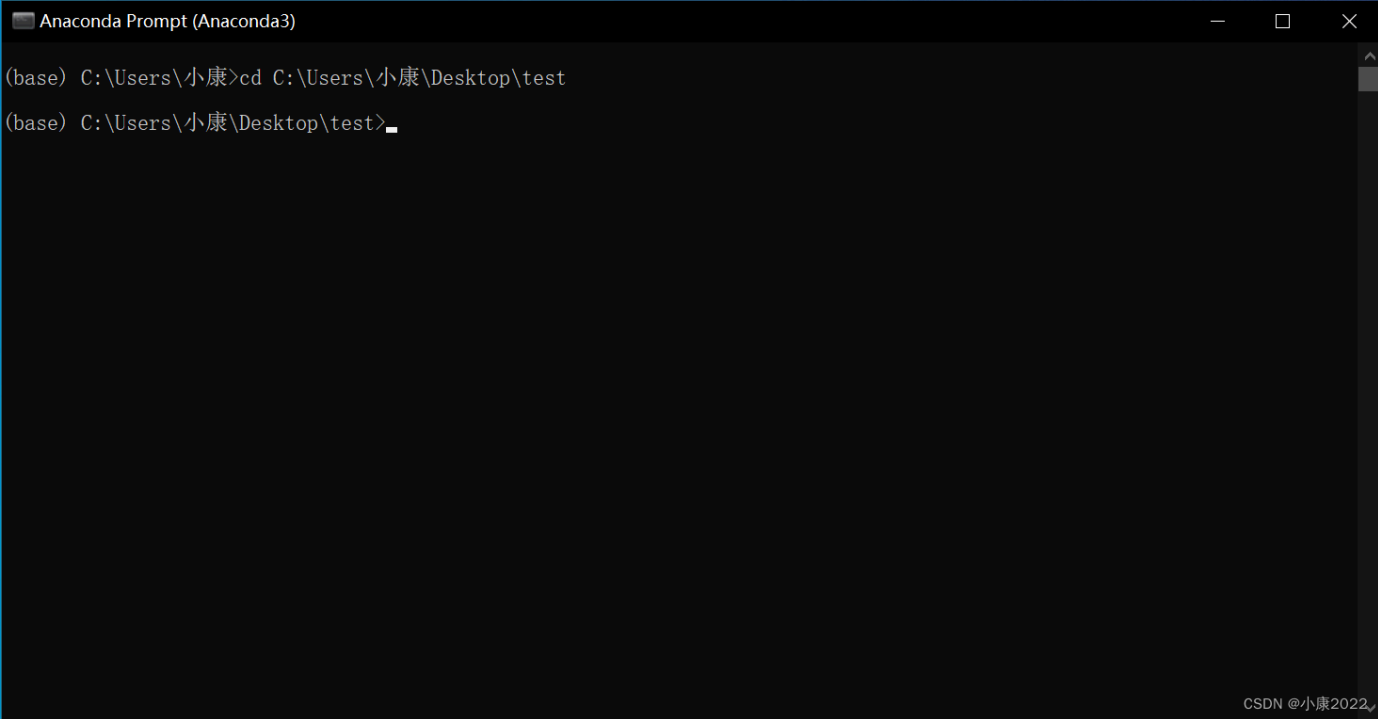
cd 文件夹路径

这里我的打包文件夹放在了桌面上，文件夹名为 test，要打包的 Python 文件在 test 文件夹内，名为 Python.py。于是我的文件夹路径为 C:\Users\小康\Desktop\test（一定要是绝对路径）。



切换路径

然后回车就可以看到下面这样的就说明成功了。



切换路径

2.2.4 打包 Python 文件

输入如下格式的命令即可

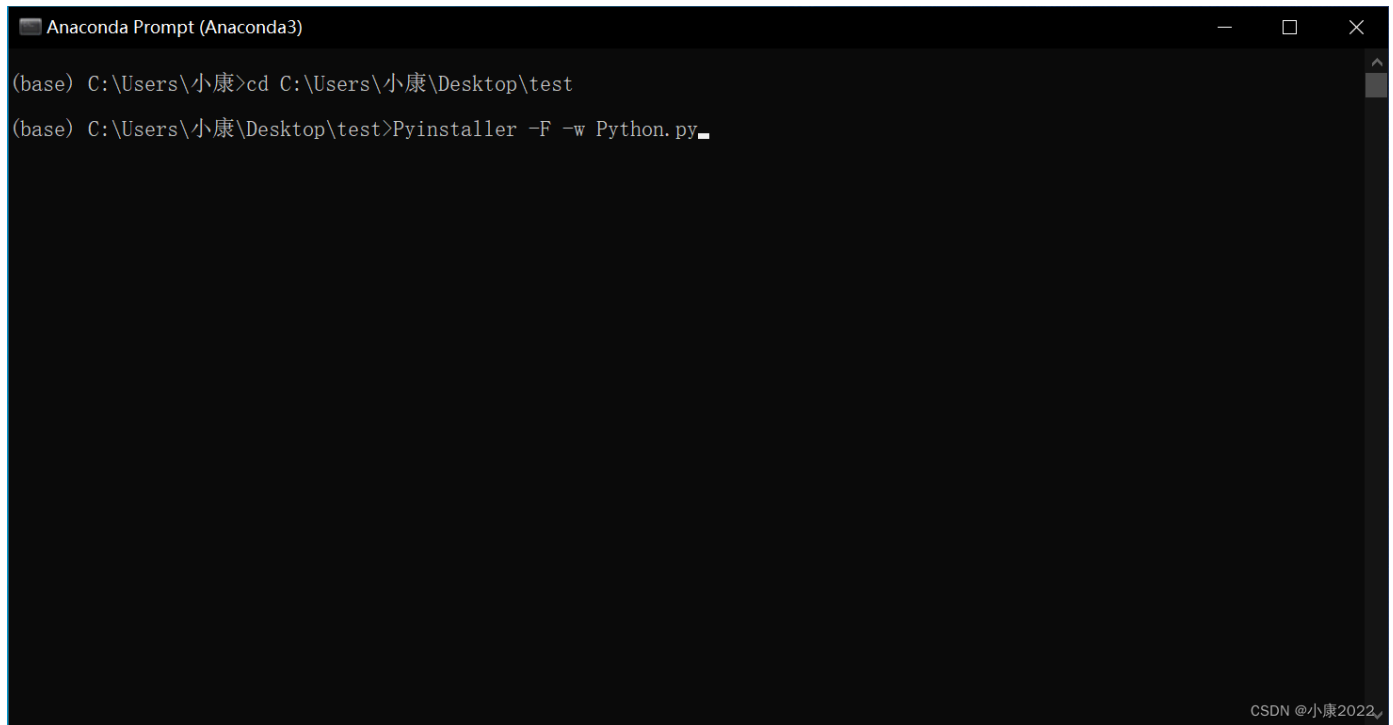
```
Pyinstaller -option1 -option2 ... 要打包的文件
```

参数选项比较多，这里我列一个表：

参数选项	描述
-F, -onefile	只生成一个单个文件（只有一个 exe 文件）
-D, -onedir	打包多个文件，在dist中生成很多依赖文件，适合以框架形式编写工具代码，这样代码易于维护
-K, -tk	在部署时包含 TCL/TK
-a, -ascii	不包含编码 在支持 Unicode 的 Python 版本上默认包含所有的编码
-d, -debug	产生 debug 版本的可执行文件
-w, -windowed, -noconsole	使用 Windows 子系统执行 当程序启动的时候不会打开命令行(只对 Windows 有效)
-c, -nowindowed, -console	使用控制台子系统执行(默认)(只对 Windows 有效) pyinstaller -c xxxx.py pyinstaller xxxx.py --console
-s, -strip	可执行文件和共享库将 run through strip 注意 Cygwin 的 strip 往往使普通的 win32 DLL 无法使用
-X, -upx	如果有 UPX 安装(执行 Configure.py 时检测)，会压缩执行文件(Windows 系统中的 DLL 也会)
-o DIR, -out=DIR	指定 spec 文件的生成目录，如果没有指定，而且当前目录是 PyInstaller 的根目录，会自动创建一个用于输出(spec 和生成的可执行文件)的目录 如果没有指定，而当前目录不是 Pyinstaller 的根目录，则会输出到当前的目录下
-p DIR, -path=DIR	设置导入路径(和使用 PYTHONPATH 效果相似) 可以用路径分割符(Windows 使用分号，Linux 使用冒号)分割，指定多个目录 也可以使用多个 -p 参数来设置多个导入路径，让 pyinstaller 自己去找程序需要的资源
-i -icon=<FILE.ICO>	将 file.ico 添加为可执行文件的资源(只对 Windows 系统有效)，改变程序的图标
-i -icon=<FILE.EXE,N>	将 file.exe 的第 n 个图标添加为可执行文件的资源(只对 Windows 系统有效)
-v FILE, -version=FILE	将 verfile 作为可执行文件的版本资源(只对 Windows 系统有效)
-n NAME, -name=NAME	可选的项目(产生的 spec 的)名字 如果省略，第一个脚本的主文件名将作为 spec 的名字

这里简单地举几个例子，让大家明白这个参数怎么写。

```
1 # 这一般是用来打包界面化的程序的，如用tkinter、Pyqt5等制作的程序。
2 # -w 的意思就是exe运行的时候不弹出那个命令行（黑窗口）
3 Pyinstaller -F -w somefile.py
4
5 # 这一般用来添加exe的图标
6 Pyinstaller -F -i someicon.ico somefile.py
```



```

Anaconda Prompt (Anaconda3)

(base) C:\Users\小康>cd C:\Users\小康\Desktop\test
(base) C:\Users\小康\Desktop\test>Pyinstaller -F -w Python.py

```

CSDN @小康2022

打包文件

然后回车它就会自动打包了。说明一下，一般我们都只会选择其中的几个参数选项，如 -F 和 -w，根据需要，我们还会选择其他的一些参数。当出现如下的文字（主要是最后一行文字）时就代表打包成功了！

```

9169 INFO: Writing RT_GROUP_ICON 0 resource with 104 bytes
9169 INFO: Writing RT_ICON 1 resource with 3752 bytes
9170 INFO: Writing RT_ICON 2 resource with 2216 bytes
9170 INFO: Writing RT_ICON 3 resource with 1384 bytes
9170 INFO: Writing RT_ICON 4 resource with 38188 bytes
9170 INFO: Writing RT_ICON 5 resource with 9640 bytes
9171 INFO: Writing RT_ICON 6 resource with 4264 bytes
9172 INFO: Writing RT_ICON 7 resource with 1128 bytes
9174 INFO: Copying 0 resources to EXE
9174 INFO: Embedding manifest in EXE
9175 INFO: Updating manifest in C:\Users\小康\Desktop\test\dist\Python.exe.notanexecutable
9176 INFO: Updating resource type 24 name 1 language 0
9178 INFO: Appending PKG archive to EXE
9183 INFO: Fixing EXE headers
9267 INFO: Building EXE from EXE-00.toc completed successfully.

```

CSDN @小康2022

打包完毕

2.2.5 打包生成文件的位置

让我们回到最初切换的文件夹里，我们可以看到，多了下面三个文件（build 文件夹、dist 文件夹和 spec 文件）：



文件夹

我们想要的 exe 文件就在新生成的 dist 文件夹里面。此时的 exe 文件有可能还运行不了，因为它可能涉及到一些资源文件或者其他的 Python 文件。将它们放到同一文件夹下即可正确运行。

这里说明一下，打包完之后，spec 文件和 build 文件夹就没用了，可以删除了。

这里一般的打包方式产生的 exe 文件都比较大，这是因为 Pyinstaller 打包的时候会把环境中的库和模块全部打包进去，这就会使一些你根本用不着的库和模块也被打包进去了！而且这些库被打包之后不仅会使 exe 文件变大，还会使其运行变卡变慢、变得十分臃肿。因此，不建议这样的打包方式。十分地建议大家用第二种方式进行打包——虚拟环境下的打包。

2.3 虚拟环境下的打包

所谓的虚拟环境，就是我们自己创建一个小型的 Python 环境，也可以这样理解，自己创一个新的、纯净的、没有奇奇怪怪的第三方库和模块的 Python 环境。这个环境你也是可以用来编写 Python 程序的，但这里我们是要来打包 exe 的，这就要求它里面的库和模块尽可能的少。

2.3.0 先介绍几个 conda 命令

① 导出虚拟环境的列表

```
conda env list
```

② 导出当前环境的包

```
conda list
```

③ 启动/切换至名为name的Python环境

```
conda activate name
```

默认为base环境（名为base）

④ 退出虚拟环境

```
conda deactivate
```


⑤ 创建新的、名为name的、Python 版本为3.x的虚拟环境

2.3.1 创建虚拟环境

```
conda create -n env 1 python==3.10.8
```



2.3.2 安装需要的第三方包

下载库很慢的，可以在 pip 时加上镜像源的地址：

```
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple 包的名字
```

这里有一点很关键！不能忘记！Pyinstaller 也是第三方的包，所以新的环境里面一定一定要 `pip install Pyinstaller`！

其他的打包步骤和一般的打包方式一模一样，请看上面的步骤。

2.3.3 追求极致的exe大小（非必要步骤）

如果你想让你的 Python 程序打包后的 exe 大小，小到不能再小的话，那么就要尽可能地删去虚拟环境里面的一些用不到的包（用 `pip uninstall` 来删）。

我这里有一个环境的包，它已经把一般程序用不到的包删干净了（没有第三方包）。你可以参考一下（通过输入 `conda list` 命令来查看）。

```
(e1) C:\Users\小康>conda list
# packages in environment at D:\Anaconda3\envs\e1:
#
# Name                      Version           Build    Channel
bzip2                        1.0.8             he774522_0 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
ca-certificates             2022.4.26         haa95532_0 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
libffi                       3.4.2             hd77b12b_4 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
openssl                      1.1.1q            h2bbff1b_0 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
pip                          22.1.2            py310haa95532_0 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
pyinstaller                  5.2               pypi_0    pypi
pyinstaller-hooks-contrib   2022.8             pypi_0    pypi
python                      3.10.0            h96c0403_3 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
setuptools                   61.2.0            py310haa95532_0 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
sqlite                       3.38.5            h2bbff1b_0 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
tk                            8.6.12            h2bbff1b_0 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
tzdata                       2022a             hda174b7_0 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
vc                           14.2              h21ff451_1 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
vs2015_runtime              14.27.29016       h5e58377_2 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
wheel                        0.37.1            pyhd3eb1b0_0 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
wincertstore                 0.2               py310haa95532_2 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
xz                            5.2.5             h8cc25b3_1 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
zlib                         1.2.12            h8cc25b3_2 https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main
```

追求极致

2.4 多 Python 文件的打包

或许对于单个文件而言，你已经清楚该怎么做了，但是对于多个 Python 文件同时打包而言，你未必清楚。

我相信我们大多数人在编写大项目的时候，都会将一个程序拆解成多个 Python 文件以便于维护。但是前面的打包方式又只能打包一个 Python 文件，其他的 Python 文件就只能作为资源文件放在外面。但是这样别人使用这个程序的时候，不就能看到那些在外面的 Python 文件的源代码了吗？谁愿意把源代码给别人免费看呢？

所以，我们就要将多个 Python 文件同时打包！这里要说一点，这里的多个 Python 文件同时打包时，还是要使用 -F 参数，生成一个文件，

这里可以创建虚拟环境来打包，也可以不用。

2.4.1 生成spec文件

同之前的步骤一样，先打开 Anaconda Prompt，然后输入如下命令以生成 Python 源文件 name.py 的 spec 文件，这里的 name.py 一般选取多个 Python 文件的主文件。

```
pyi-makespec -option1 -option2 -... name.py
```

option 参数和之前步骤里的一样，输入你需要的参数即可。回到我们源代码的文件夹中，可以发现已经多了一个 name.spec 的文件了。

2.4.2 编辑spec文件

spec 文件实际上就是一个文本格式的文件，可以用任意文本编辑器打开，也可以用你的 IDE 直接打开，细心的人会发现，里面的内容实际上就是 Python 格式的代码，只不过文件扩展名改成了 spec 而已。

spec 文件实际上就包含了打包的所有参数，我们可以对其进行修改，以达到自定义打包的效果。

找到下面的这句：

```
*Python.spec - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
# -*- mode: python ; coding: utf-8 -*-

block_cipher = None

a = Analysis(
    ['name.py'],
    pathex=[],
    binaries=[],
    datas=[],
    hiddenimports=[],
    hookspath=[],
    hooksconfig={},
    runtime_hooks=[],
    excludes=[],
    win_no_prefer_redirects=False,
    win_private_assemblies=False,
    cipher=block_cipher,
    noarchive=False,
)
pyz = PYZ(a.pure, a.zipped_data, cipher=block_cipher)
```

spec 文件

它实际上就是个列表！将你需要的 Python 文件的路径（允许相对路径）都以字符串的形式写进这个列表里面，如果是与 name.py 不在同一目录（同一文件夹）下的 Python 文件，那就要写它的绝对路径。编辑完之后，记得保存文件。

2.4.3 以spec文件进行打包

回到之前的命令行（Anaconda Prompt），输入以下命令进行打包。

```
Pyinstaller name.spec
```

慢慢等待，打包完之后，就是我们想要的 exe 文件了，它把所有的 Python 文件都加了进去！但是很可惜，资源文件还是要放在同一目录下才可以正确运行 exe 程序。不过我还是极力推荐这种方式！我每次打包就是用的这一种方式，毕竟资源文件也不是必须打包进 exe 才好的，有些时候，我们的 Python 代码一般都不会超过 1MB 吧（想必大部分人都没有），而资源文件却有大几十甚至几百 MB，打包进去之后，会使得 exe 程序运行变慢，这不好。

而且，你想啊，现在大部分的软件，资源文件啊什么的不都是放在 exe 外面的么？

2.5 包含资源文件的打包

这个打包方式，就是对多个 Python 文件的打包方式的补充。

2.5.0 一点吐槽

说到这个，我不得不吐槽一句，网上大部分的打包资源文件的方法都是一模一样的，繁琐且复杂，而且好多根本都实现不了，搞得我当时初学的时候一脸懵……，什么引用 os、sys 库搞些什么路径操作啊什么的，辣么麻烦，还有什么把图片文件用 base64 硬编码的啊什么的，也不解释原理，只能说离谱……

Python 的简约风格都被他们忘得一干二净了！对于打包资源文件的路径问题，虽然它打包后认不得相对路径，但是绝对路径总是认得的嘛，没有程序不认识绝对路径！

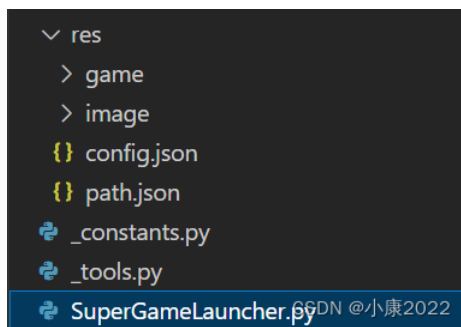
有人说绝对路径改不了啊，到别人的电脑上怎么运行呢？我只能说……你见识短浅了！一个简单的装饰器知识就可以解决的问题！

好了，不吐槽了，继续说正事。

多个 Python 文件的打包还是和之前讲到的一样，这里只说资源文件的方法。也是一样的，编辑 spec 文件。

2.5.1 编辑spec文件

我的建议是，把资源文件（或者文件夹）都统一放在一个与 Python 文件同一目录下的 res 文件夹里，方便打包，就如下图。



资源文件夹

然后，将下图中标识的这一行改成这样：



spec 文件

改完记得保存！

2.5.2 修改文件打开函数

这里有三种方法，前两种是我的方法，最后一种是网上别人的方法。

① 引入特定的模块

这个模块的代码很简单，放在下面，**一定要将模块命名为 `_py`，并在引用其他第三方模块之前就引用它，但又一定要在下划线开头的模块之后引用**（否则会有 BUG），**它在主 Python 文件里引用一次即可（其他的文件不用引用）**！它可以将 `open` 函数改成我们想要的，而且原来的代码还完全不用修改！

```
1 import builtins
2
3
4 def wrapper(function):
5     def _open(*args, **kw):
6         """ 修改路径 """
7         _args = list(args)
8         _args[0] = __file__[:-4] + args[0]
9         if kw.get('file'):
10             kw['file'] = __file__[:-4] + kw['file']
11         return function(*_args, **kw)
12     return _open
13
14
15 setattr(builtins, 'open', wrapper(open))
```

这里的 `__file__` 是 Python 文件的属性，是一个字符串，为该文件的绝对路径，不管该文件在哪里，`__file__` 都是对应的绝对路径。然后我们用写一个 `wrapper` 函数充当装饰器，将内置的 `open` 函数包装一下。再引入 `builtins` 模块（内置函数和类的模块），给它添加一个名为 `open` 新属性以覆盖原来的 `open` 函数，并对该项目整体生效即可！

这种方法的好处在于，它只需要在主文件里引用一次即可，其他的什么都不用改！（④ 特别注意 里的除外）

② 自己手动修改 open 函数修改路径

这个修改是在源代码中修改的（每一个用到了 open 函数的 Python 文件都要改一次），目的就是为了让相对路径变成会根据主 Python 文件的路径而变化的绝对路径。修改的装饰器如下：

```
1 # 编写装饰器
2 def wrapper(function):
3     def _open(*args, **kw):
4         """ 修改路径 """
5         args_list = list(args)
6         key = '/'.join(__file__.split('\\')[:-1]) + '/'
7         args_list[0] = key + args[0]
8         if kw.get('file'):
9             kw['file'] = key + kw['file']
10        return function(*args_list, **kw)
11    return _open
12
13
14 # 装饰内置函数open
15 open = wrapper(open)
```

把这段代码写在文件的开头即可（或者说在使用open函数之前）。

③ 网友的其他方法

他们就是写了这样一个函数来代替 open，也是手动修改的 open 函数，不得不说，看起来有点麻烦（每个用了 open 函数的 Python 文件都要引入 os 和 sys 模块）。

```
1 import os
2 import sys
3
4
5 def get_resource_path(relative_path):
6     if hasattr(sys, '_MEIPASS'):
7         return os.path.join(sys._MEIPASS, relative_path)
8     return os.path.join(os.path.abspath("."), relative_path)
```

其他的都是一样的。

④ 特别注意

这里还要提一下，无论是前面的哪一种方法，只要你使用了参数为路径的其他函数时，也要改一下，其实就是在相对路径前面加上方法②中的 key 即可。

其实直接方法③来代替也可以，但是功能上容易出错，而且如果 Python 文件较多，那么每个 Python 文件都这样引用两个模块（sys 和 os），看起来比较麻烦。

当我们使用了 tkinter 模块的时候，PhotoImage 类就是要这样写的一个例子（其中的 __init__ 方法用到了路径）：

```
1 class PhotoImage(tkinter.PhotoImage):
2
3     def __init__(self, *args, **kw):
4         if kw.get('file'):
5             key = '/'.join(__file__.split('\\')[:-1]) + '/'
6             kw['file'] = key + kw['file']
7         tkinter.PhotoImage.__init__(self, *args, **kw)
```

这个代码就要写在使用 PhotoImage 的开头，后续调用时就用这个 PhotoImage，使用其他模块时，遇到参数为路径的函数或类，都要这样修改。

最后一步，和之前的方法一样，打包你的程序即可！

三、打包实战

我这里以一个我的半成品为例，进行打包。项目是一个图形化界面的程序。我们要将其打包成只含有一个 exe 的文件。

3.1 打包方式的选择

我的项目里面包含多个 Python 文件，要用**多 Python 文件打包**方式；

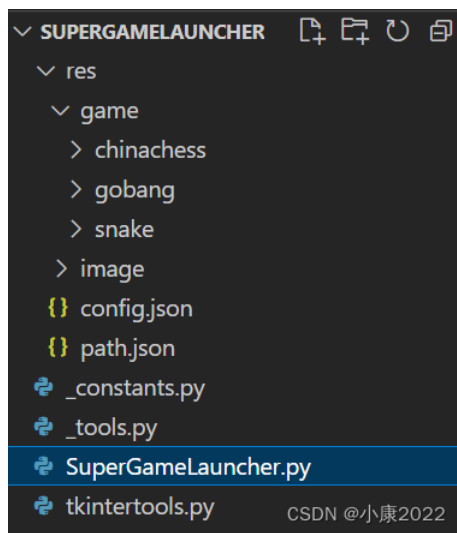
项目比较大，为节省打包时间，并追求极致的 exe 大小，采用**虚拟环境打包**方式；

项目含有资源文件夹，采用**包含资源文件的打包**方式。

打包方式选择好了，开始打包！

3.2 打包全过程

项目全部文件（蓝色背景的是主文件）

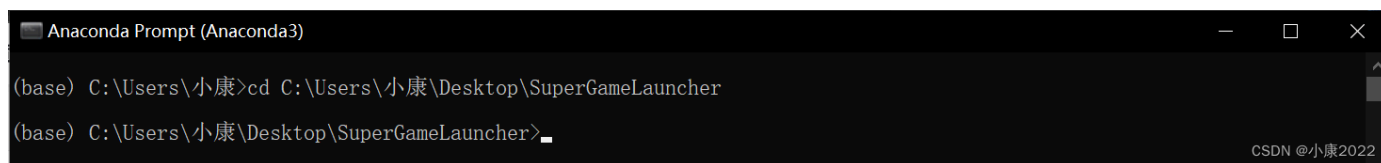


项目文件

包含四个 Python 文件、一个资源文件夹（res），资源文件夹里面又包含了一些子文件夹和 json 文件。

3.2.1 第一步：启动 Anaconda Prompt，切换至目标文件夹路径位置

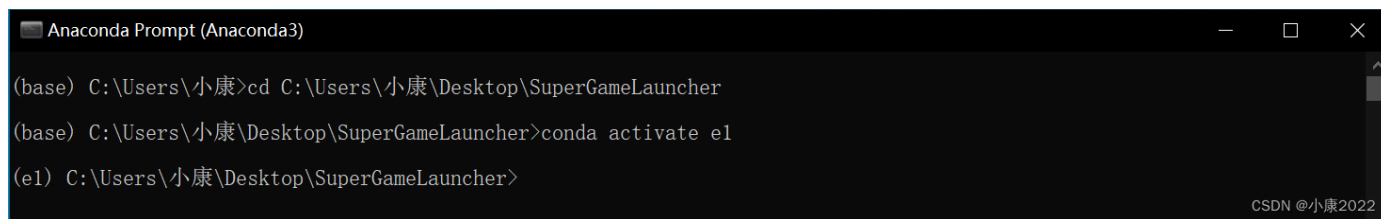
```
cd C:\Users\小康\Desktop\SuperGameLauncher
```



切换至目标文件夹路径位置

3.2.2 第二步：启动虚拟环境（我的是一个纯净的、第三方包只有 Pyinstaller 的环境）

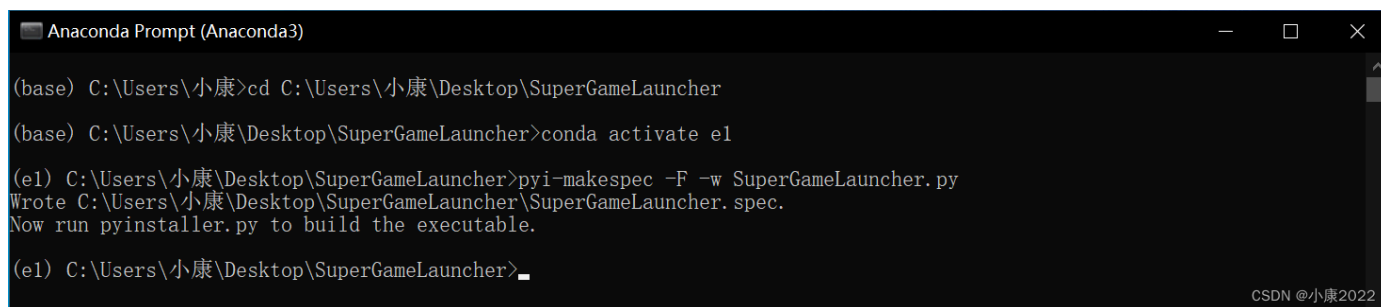
```
conda activate e1
```



启动虚拟环境

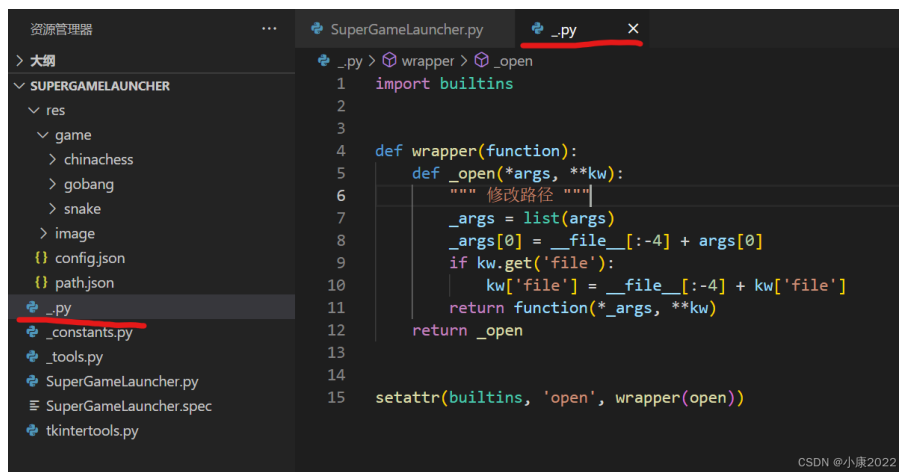
3.2.3 第三步：生成 spec 文件（我的 Pyinstaller 已经安装好了）

```
pyi-makespec -F -w SuperGameLauncher.py
```

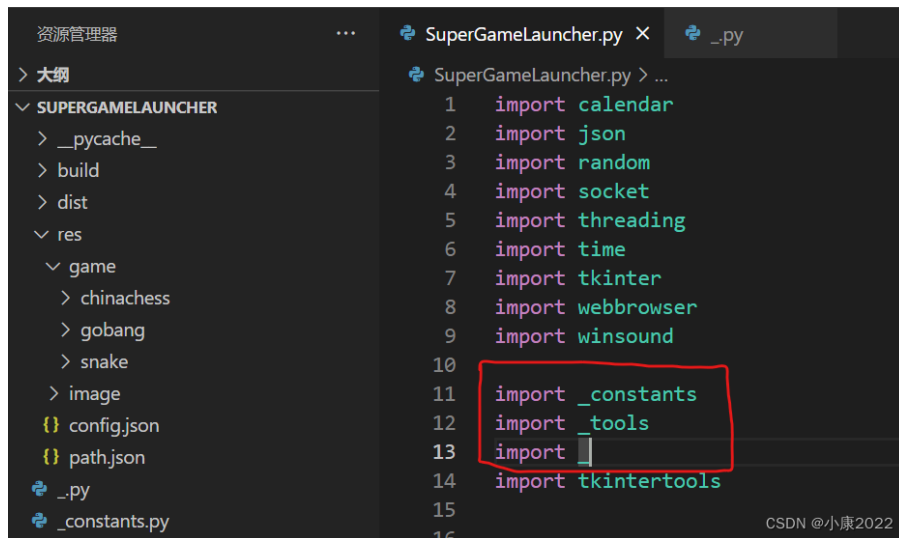


生成 spec 文件

3.2.4 第四步：引入 _py 模块（我的程序用到了大量 open 函数且涉及多文件）

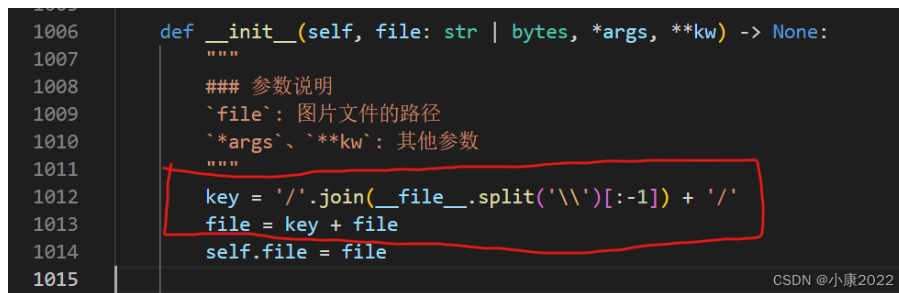


_py 自定义模块



引入 _ 模块

处理特殊的情况（tkintertools 模块里面有参数为路径的类）：



处理特殊情况

3.2.5 第五步：编辑 spec 文件

修改图中标识的两处地方（_py 不要忘了）。

```
SuperGameLauncher.spec - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
# -*- mode: python ; coding: utf-8 -*-

block_cipher = None

a = Analysis(
    ['SuperGameLauncher.py', 'tkintertools.py', '_tools.py', '_constants.py', '_py'],
    pathex=[],
    binaries=[],
    datas=[('res', 'res')],
    hiddenimports=[],
    hookspath=[],
    hooksconfig={},
    runtime_hooks=[],
    excludes=[],
    win_no_prefer_redirects=False,
    win_private_assemblies=False,
    cipher=block_cipher,
    noarchive=False,
)
pyz = PYZ(a.pure, a.zipped_data, cipher=block_cipher)
```

第 1 行, 第 1 列 100% Windows (CRLF) UTF-8 编码

编辑 spec 文件

3.2.6 第六步：打包项目（注意这里的对象是 spec 文件）

Pyinstaller SuperGameLauncher.spec

```
Anaconda Prompt (Anaconda3)
(base) C:\Users\小康>cd C:\Users\小康\Desktop\SuperGameLauncher
(base) C:\Users\小康\Desktop\SuperGameLauncher>conda activate e1
(e1) C:\Users\小康\Desktop\SuperGameLauncher>pyi-makespec -F -w SuperGameLauncher.py
Wrote C:\Users\小康\Desktop\SuperGameLauncher\SuperGameLauncher.spec.
Now run pyinstaller.py to build the executable.
(e1) C:\Users\小康\Desktop\SuperGameLauncher>Pyinstaller SuperGameLauncher.spec_
```

CSDN @小康2022

准备打包

打包成功!

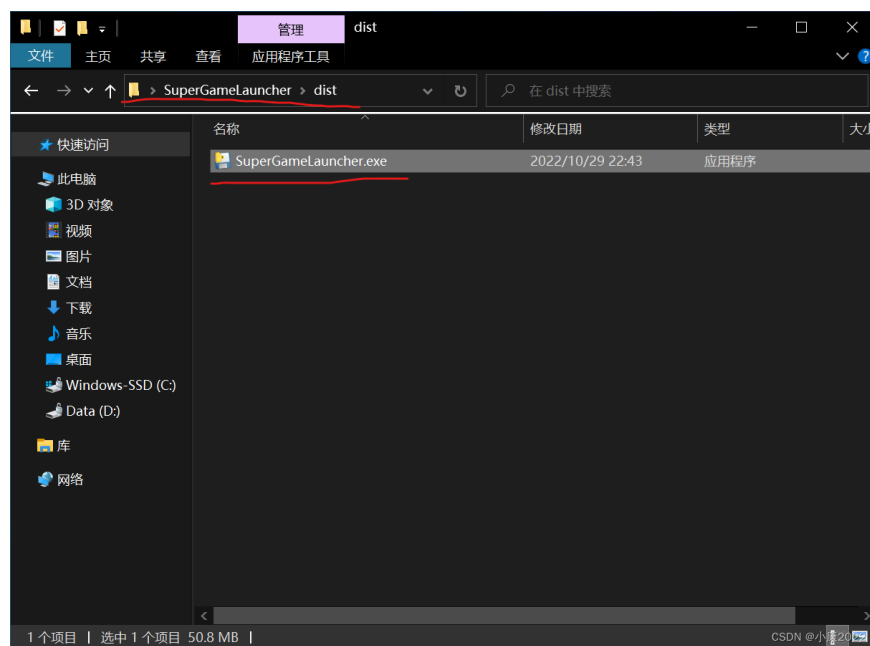

```
Anaconda Prompt (Anaconda3)
7869 INFO: checking PKG
7869 INFO: Building PKG because PKG-00.toc is non existent
7869 INFO: Building PKG (CArchive) SuperGameLauncher.pkg
10907 INFO: Building PKG (CArchive) SuperGameLauncher.pkg completed successfully.
10927 INFO: Bootloader D:\Anaconda3\envs\el\lib\site-packages\PyInstaller\bootloader\Windows-64bit\runw.exe
10927 INFO: checking EXE
10927 INFO: Building EXE because EXE-00.toc is non existent
10927 INFO: Building EXE from EXE-00.toc
10927 INFO: Copying bootloader EXE to C:\Users\小康\Desktop\SuperGameLauncher\dist\SuperGameLauncher.exe.notanexecutable
10939 INFO: Copying icon to EXE
10939 INFO: Copying icons from ['D:\Anaconda3\envs\el\lib\site-packages\PyInstaller\bootloader\images\icon-windowed.ico']
10939 INFO: Writing RT_GROUP_ICON 0 resource with 104 bytes
10939 INFO: Writing RT_ICON 1 resource with 3752 bytes
10939 INFO: Writing RT_ICON 2 resource with 2216 bytes
10939 INFO: Writing RT_ICON 3 resource with 1384 bytes
10939 INFO: Writing RT_ICON 4 resource with 38188 bytes
10939 INFO: Writing RT_ICON 5 resource with 9640 bytes
10939 INFO: Writing RT_ICON 6 resource with 4264 bytes
10939 INFO: Writing RT_ICON 7 resource with 1128 bytes
10939 INFO: Copying 0 resources to EXE
10939 INFO: Embedding manifest in EXE
10939 INFO: Updating manifest in C:\Users\小康\Desktop\SuperGameLauncher\dist\SuperGameLauncher.exe.notanexecutable
10939 INFO: Updating resource type 24 name 1 language 0
10949 INFO: Appending PKG archive to EXE
10979 INFO: Fixing EXE headers
11209 INFO: Building EXE from EXE-00.toc completed successfully.

(e1) C:\Users\小康\Desktop\SuperGameLauncher>
```

打包成功

3.2.7 第七步：检验打包效果

找到项目中的 dist 文件夹，打开后会会有一个 exe 文件。对于我这个项目而言，这已经是非常小的尺寸了（50.8MB），毕竟资源文件就有 42.2MB，也就是说，除去资源文件，源代码占用的尺寸为 8.6MB！怎么样，是不是很不错呢？



dist 文件夹

双击运行！



四、疑难解答与相关建议

4.1 疑难解答

因为经常有朋友问我各种各样关于打包失败的问题，我在此处给出一些问题的可能解决办法（不保证解决，仅供参考）。

4.1.1 模块找不到的错误 —— ModuleNotFoundError

这与你的打包方式有关，有些打包方式会出现这种情况，了解 Pyinstaller 的机制会有一定帮助。

Pyinstaller 打包后程序运行机制和直接用 Python 解释器相比，在源代码文件查找方面有一点不同，Python 解释器是在工程目录下查找源代码文件和资源文件，但是 Pyinstaller 如果是将所有文件打包成一个可执行文件，而没有其他的文件，那么它会运行前将源代码文件和资源文件重新生成出来，放在用户临时文件夹中，然后修改工程目录的根目录地址为那个临时文件夹路径，这样 Python 就能找到对应的文件。如果你没有打包好，导致文件没有被打包进 exe，就无法被重新生成，也就导致了 ModuleNotFoundError。

当然，如果你本就不打算将源代码文件和资源文件放进可执行文件里面，而是像一般的软件一样，分别放在各个文件夹中，那就没啥事了，不用考虑上面的情况。此时你出现这种报错，纯粹就是真的缺少对应的模块，如果是用虚拟环境打包的，可能是虚拟环境缺少对应的模块，如果不是，那么可能是你不小心把什么东西给删除了。

4.1.2 图形化程序运行没反应，但也没有报错

此时应该看看你打包时有没有加上 -w 命令。-w 的作用是将运行时出现的终端隐藏，但是所有的报错信息一般都是出现那里面的，你把它隐藏了肯定就看不到了，因此你应该将这个命令删去后打包，然后运行，查看报错信息，彻底解决问题后再重新进行打包（此时再把 -w 命令加回来）

4.1.3 语法错误 —— SyntaxError

这种情况，先看看源代码有没有问题（别搞成下面的其他情况中的第一种了），如果没有问题，那可能是你在打包时使用的 Python 版本与源代码运行时版本不一致导致的。简单说，出现了比较高级的语法，低版本 Python 无法解释。比如 Python3.12 用于简化泛型的类型形参语法，Python3.11 的类型匹配语法 match 和 Python3.8 的海象运算符等。

但也并不是说高版本就一定解决问题，或许这个语法错误是解决了，但可能出现一些不兼容的问题，一般是第三方模块导致的，比如第三方包 scipy 的 API 曾经就大幅度修改过。使用对应版本的 Python 解释器非常重要！

4.1.4 'pyinstaller' 不是内部或外部命令.....

这种情况一般是你电脑上安装的 Python 非全局环境下的（你可以理解为环境变量有问题）。网上有很多方法解决这个问题，但大多都是要你去改环境变量，这样太麻烦了！既然操作系统无法直接调用它，那让 Python 解释器调用它不就行了？

很简单，Python 的环境变量一般不会有问题，这样写，用全局环境下的 Python 去调用其第三方模块 pyinstaller 即可：

```
python -m pyinstaller ...(后面内容一样)
```

省略号表示正常情况下的其他命令。注意，有些操作系统可能要将上面的 'python' 改为 'python3'。

4.1.5 其他情况

如果是很一般的报错，并非上述提到的某一种，此时你应该先检查源代码是否可以正常运行！这一点很重要！曾经有个哥们儿，拿着别人的程序源代码，想打包后再使用，给我看了一系列报错和问题，我半天找不到问题所在，折腾了半小时都搞不定。我最后问他，源代码可以正常运行吗？过了一会儿，他居然跟我说，源代码运行就已经报错了！！真搞不懂怎么会有人出现这种情况，源代码都无法运行，打包肯定也运行不了啊！

如果没有上述的任何一种，但就是一些莫名其妙的问题，可能与 Pyinstaller 模块本身或者平台有关，这我暂时也没有找到比较一般的解决办法。

4.2 相关建议

4.2.1 关于虚拟环境的建立

那啥，大伙儿搭建虚拟环境不要再使用 Anaconda 了，那玩意儿真的大（几个 GB），而且运行起来非常慢，附带了一堆不必要的东西（我只是搭个虚拟环境而已，用不到那么多啊）。建议使用 miniconda，Anaconda 的官方精简版本，只包含了 conda 核心（搭建虚拟环境要用的玩意儿）和一些小型包（自带的一个 base 环境）。miniconda 只有 100 多 MB，和 Anaconda 完全不是一个量级的，非常小巧，渣电脑也能带的动。

不过，要注意的是，miniconda 是没有图形界面的，新手可能不会操作……😂我的评价是，自己去学吧。

4.2.2 关于项目文件的操作

建议打包前备份一份完整的项目文件，避免打包失败产生影响整个项目的源代码（打包会产生一些文件和文件夹）。

后续可能还会对此文章进行持续更新，增加更多疑难问题的解答和建议，但，也可能不更新（缺少更新动力 —— 你的点赞和收藏😂）。

看了这么多，我要你一个赞和一个收藏不过分吧？不收藏也行，至少点赞……算了😂