使用自定义文件(USING SPEC FILE)

当你执行pyinstaller options.. myscript.py进行文件打包时，PyInstaller做的第一件工作并不是创建spec文件。这个文件会创建于—specpath=参数所指定位置，默认是在当前目录。

spec文件很重要，因为它会告诉PyInstaller怎样去处理你的代码。这个文件会将代码名称大多数你在命令行输入的参数进行编码。实际上spec文件就是一段可执行的Python代码。PyInstaller创建应用也就是靠执行这段代码。

对于PyInstaller的大多数使用情况，你并不需要测试或修改spec文件，因为它可以通过你在命令行输入的参数来告诉PyInstaller所有必要的信息。

但当你处于下面四种情况，就有必要自己修改spec文件：

1. 你想把代码所用到的数据文件一并进行打包。
2. 当你希望包含一些PyInstaller并不了解的run-time库，比如.dll .so这样的文件。
3. 当你想添加Python run-time选项到可执行文件中。
4. 当你想创建一个拥有常见合并模组（merged common modules）的多程序(multiprogram)包。

后文会提及针对这几个情况的相关处理方法。

首先你可以通过这个命令来创建一个spec文件：

pyi-makespec options name.py [other script..]

上面命令中的options和之前选项参数章节的是一致的，如果你忘记了可以指定哪些参数，现在不妨回去查看一下。这个命令不会创建可执行文件。

当你使用上面的命令创建了spec文件并按照自己的需求修改了spec文件，你就可以输入以下命令来通过这个spec文件创建可执行文件：

pyinstaller options name.spec

当你创建了spec文件，大多数命令选项都会被编码。然后当你依此spec文件创建可执行文件时，这些指定的选项或者参数就不能被修改了。即便你仍然添加在上面命令的options中，他们也会被spec文件中指定的相同类型的参数所顶替掉。这是你需要留意的。

换句话说，如果你是用spec文件进行构建，只能在命令行中输入以下参数（也就是上面options中）:

-upx-dir=

-distpath=

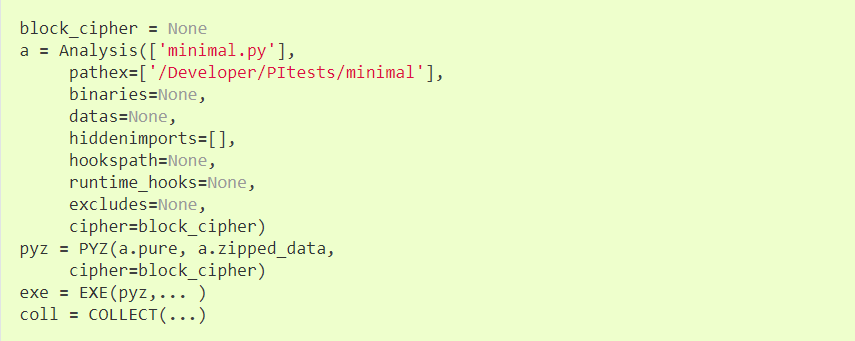
-workpath=

-noconfirm

-ascii

SPEC文件相关的操作(Spec File Operation)

当使用PyInstaller创建了spec文件，或者打开了一个已经给定的spec文件，PyInstaller命令都会像执行代码一样执行spec文件。下面是一个很简短的spec文件，其作用是用来指引PyInstaller打包一个最小化、单文件夹的应用：



spec文件的内容中包含这么几类：Analysis,PYZ,EXE以及COLLECT。那么分别来简单讲一下每一类的作用：

1. Analysis是代码文件相关配置的清单。它用来分析所有模组导入和其他依赖。分配给a的结果对象包含了一张各个被命名的类成员所包含依赖的列表：

scripts:在命令行中被命名的python代码文件

pure:原始代码所需要的纯python模组

binaries:原始代码所需要的非python模组，包括—add-binary选项提供的参数

datas:要被应用包含的非二进制文件，包括—add-data选项所提供的参数

1. PYZ类的实例是一个.pyz格式的压缩包（在后面的Inspecting Archives部分会讲到），用来压缩所有来自a.pure的Python模组。
2. EXE类的实例由分析代码和PYZ压缩包所创建，这个对象用于创建可执行文件。
3. COLLECT类的实例用来创建所有其他部分的输出目录。

上面说的实例就是你写在spec文件中的a,pyz,exe与coll，spec文件是python代码而不是一般的配置文件。

向包中添加文件(Adding Files to the Bundle)

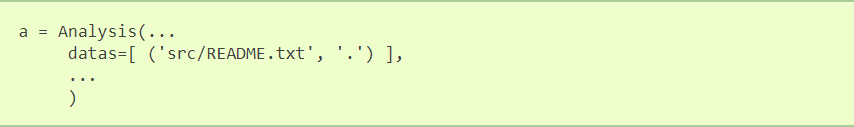
为了添加文件，你已经创建了一张文件列表并把它提供给Analysis函数。当你要打包成单目录，这些要添加的文件就会被拷贝到与可执行文件伴随的目录中。当你要打包成单一可执行文件，这些文件就会被压缩进可执行文件，然后当你启动应用时，解压到那一长串名称的缓存目录中（如果意外终止缓存目录可能不会被删除）。不管在什么情况下，如果你想在run-time中找到这些数据文件，请查看Run-time Information章节。

添加数据文件(Adding Data Files)

你既可以通过命令行中的—add-data也可以通过修改spec文件来添加数据文件。如果通过后者，你则需要提供一份列表来提供这些数据文件的信息。这份列表需要是元组的形式。每个元组必须有两个值，且类型必须是字符串：

1. 第一个值必须是目前存在于系统的文件
2. 第二个值指定包含这个文件的目录名称

我明白，给你个例子会更直观一些：



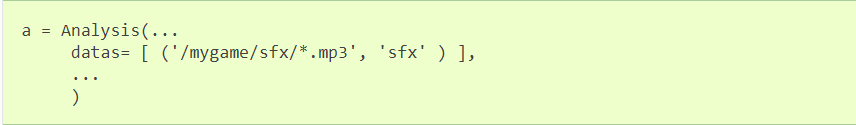
上面的内容就相当于你在命令行输入：



不过，可别忘了，如果你是用spec文件来引导PyInstaller进行打包，它就不会再读命令行中一些已经在spec文件中给出的参数了。

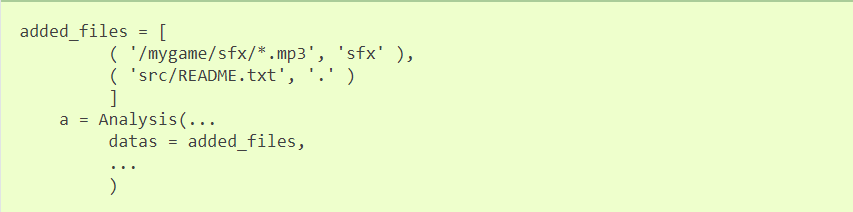
通过上面例子的这种写法，PyInstaller就会把README.txt这个文件拷贝进最高级（原文是the top level）的打包应用中。对了，这里的路径是相对于spec文件而言的。

还有/ \都可以在这里作为路径的分隔符号，你还可以使用一些匹配符号，比如说

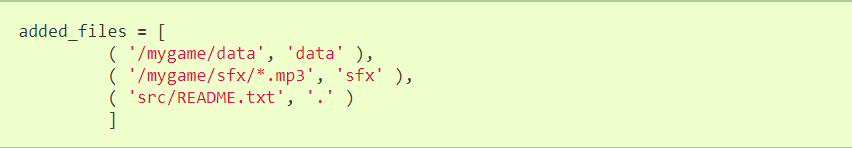


这样sfx目录下的所有mp3为后缀的文件都会被拷贝进打包应用下名为sfx的文件夹中。

当然如果你看过关于一些Pythonic的文章，里面一定提到了一些让代码更可读的方法。这也是Python这门语言在设计之初就定下来的基础思想。比如你可以这样编写：



当然不仅可以指定文件，文件夹也是可以指定的，比如下面就会将整个data目录复刻到包中：



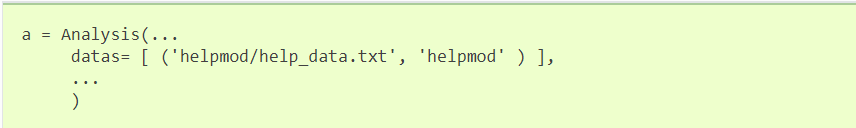
从一个模组中使用数据文件(Using Data Files from a Module)

如果你要添加的数据文件是包含在一个Python模组中的，那么你可以通过pkgutil.get\_data()的方法来重新取回它们。

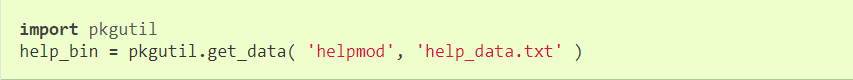
比如说你的应用有个模组叫helmod，然后你的代码和代码的spec文件都在这个目录下：



然后你的代码中肯定包含import helpmod，这样PyInstaller就会在打包应用中创建目录的准备内容（arrangement？），不过只会包括.py文件，而help\_data.txt这个文件就会被华丽地无视。而代码又是需要这个数据文件的。那么你必须把它添加进spec的Analysis中：



这样当你的代码在被执行的时候，才能顺利地找到这个文件。当然，因为这个数据文件是模组的一部分，所以你也可以使用标准库方法pkgutil.get\_data()来获取这个数据文件，具体用法是：



如果是Python3，那么help\_data.txt文件返回的内容是二进制字符串，所以你可能要进行解码

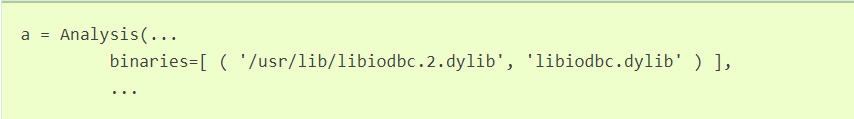


添加二进制文件(Adding Binary Files)

你可以在打包时通过在命令行中添加—add-binary选项来添加二进制文件，当然也可以在spec文件中为它们创建一个待添加的列表。如果采用后者，你需要创建一个元组来记录这些文件，一定要是元组(tuples)，可别写成列表了。两者最大的区别就是python规定元组定义好之后就只是可读的，而不能随意修改这些数据！然后把这个元组赋给spec文件的Analysis中的binaries=参数。

一般来说，PyInstaller会通过分析导入的模组来找出其中涉及到的.so以及.dll库文件。但也有“二般”情况，就是PyInstaller搞不清楚有些模块是否真正地被代码导入与使用。你也看到有—hidden-import=这么一个参数，它就是为这种特殊情况准备的。不过即便如此，PyInstaller仍可能找不到所有代码依赖的内容。

假设，有这么一个C语言编写、且使用了Python C-API的叫special\_ops.so的模组。然后你在代码中用Import的方式导入了special\_ops，PyiNstaller也里当然分析到了这个导入关系。但可怜的程序并不知道，special\_ops.so还和libiodbc.2.dylib还有一层依赖关系，所以并没有分析到还需要将后者也打进压缩包中。遇到这种情况，就需要你再做一点微小的工作了：



或者在命令行这么敲一行：



另外，因为spec文件本身就是python代码，所以在编写的时候，你可以按照自己的习惯或者需求改变代码“陈述”的方式。

高级的添加文件方式(Advanced Methods of Adding Files)

PyInstaller还支持一种非常高端上档次的添加文件的方法。在一些很特殊的情况下很奏效，也就是下面的The TOC and Tree Classes部分。

给予运行时Python选项(Giving Run-time Python Options)

python解释器可以获取很多命令行选项，但对于已经打包的应用而言，只有下面几项是被支持的：

v 每次当模组初始化时，用参数v向标准输出传送一条信息

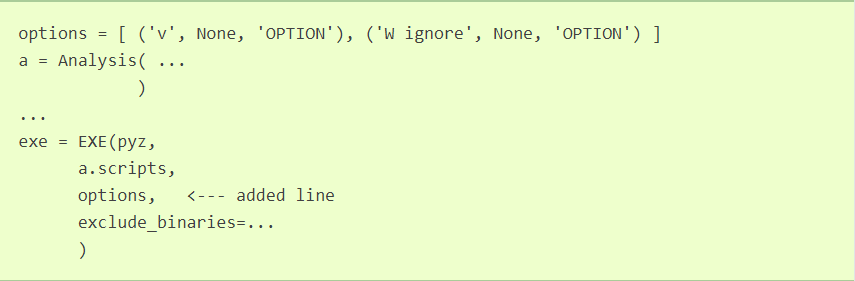
u 对于未缓冲的标准输入输出(unbuffered stdio)

w 用来修改警告行为，比如W ignore W once W error

如果想传递这些选项，也得像上面的在spec文件中添加二进制文件一样，创建一个元组来填写这些内容。每个元组包含三项内容:

1. 用来表示选项的字符串，比如说v或W ignore
2. None(既然是空为啥要列出来啊？难道是为了保持格式统一？)
3. 字符串 OPTION

来个例子便于大家理解：



对于在Mac OS X下打包的Spec文件选项(Spec File Option for a Mac OS X Bundle)

这块内容是讲在Mac OS X 环境下给打包程序添加图标与Info.plist的内容，看篇幅，Info.plist是在Mac OS X系统下很重要的内容，但是目前主要考虑到本文是给Linux环境下要给python代码进行快速打包的运维或开发人员以及自己的翻译计划（其实翻译主要是为了逼自己看几篇常用python模块的英文文档），暂时先不翻译这部分。在之后有空余时间且积累一定翻译经验之后再补全这些未翻译的部分。

多程序包打包(Multipackage Bundle)

注意！这个特性在PyInstaller3.0的版本中是损坏不可用的，请在问题修复之前不要尝试此功能。如果你很需求这个特性，请到github PyInstaller Issue #1527留言。

有一些产品是由不同的应用组成，而这些组件应用可能都依赖同一套第三方库，或者共享某些代码。那么如果在打包时，把这些应用单独对待，分别拷贝多份相同的代码与库文件到打包内容中，那多么令人遗憾啊！（这种追求完美的态度值得我们学习，但是在打造完美的过程中，却需要精湛的编程技艺与全方位的考虑）

所以我们想到这个Multipackage的特性，让你可以把一系列共享相同代码和依赖库的应用进行打包。并且和一般打包一样，既可以打包成单独文件夹的形式，也可以最小化到单独一个可执行文件。所有这些应用都会共享一个外部指向，告诉它们依赖的DLL内容在什么地方。

这样就会大大节省磁盘空间。每件事物都有其代价，应用启动的时间会相应增加。（这里需要屏幕前的你进行评估，做出适合自己的选择）。

对于包含指向输出目录的硬编码路径的二进制文件之间的外部引用而言，是不能被重新排列的。（The external references between binaries include hard-coded paths to the output directory, and cannot be rearranged. ）如果你使用的是单一目录模式，你必须把所有应用相关的目录都存放在相同的父级目录下。如果你使用的是单一文件模式，在你安装应用时候，也得把所有相关的应用都放置在相同的目录下。

为了创建这样的一组应用，你应该在打包之前自定义spec文件，使其包含MERGE函数的调用。这个函数需求一张分析代码的列表，然后从中找到共同的依赖关系，然后让分析过程尽可能减少对存储的需求。（就是尽量减少打包内容重复的依赖内容）

对于在参数列表中待分析对象的顺序不同会产生不同的影响。

The order of the analysis objects in the argument list matters. The MERGE function packages each dependency into the first script from left to right that needs that dependency. A script that comes later in the list and needs the same file will have an external reference to the prior script in the list. You might sequence the scripts to place the most-used scripts first in the list.

所以你应该把用得最多的代码排序在列表的首位。

A custom spec file for a multipackage bundle contains one call to the MERGE function:

对于多包（mulitipackage）捆绑包的自定义spec文件也包含对MERGE函数的调用。

MERGE(\*args)

MERGE is used after the analysis phase and before EXE and COLLECT. Its variable-length list of arguments consists of a list of tuples, each tuple having three elements:

The first element is an Analysis object, an instance of class Analysis, as applied to one of the apps.

The second element is the script name of the analyzed app (without the .py extension).

The third element is the name for the executable (usually the same as the script).

MERGE examines the Analysis objects to learn the dependencies of each script. It modifies these objects to avoid duplication of libraries and modules. As a result the packages generated will be connected.

举例说明MERGE spec文件(Example MERGE spec file)

对Spec文件可用的全局参数(Globals Available to the Spec File)