SHIT HAPPENS

在之前的章节讲到了很多关于PyInstaller的一般用法。但是由于Python版本碎片化和无穷尽产生的第三方库的不可预测性等原因，当你使用PyInstaller对你心爱的应用打包时，常常会看到Python trackback这种字样（又tm报错了！）。那么当你在寻求专家帮忙之前，不妨按照下面的顺序“操作”一下，看能不能依靠自己的力量把问题解决了。

对特殊问题的解决方法(Recipes and Examples for Specific Problems)

在PyInstaller的FAQ页面中可以找到一些解决常见问题的方法。在PyInstaller Recipes页面你还能找到一些高级使用或处理问题的代码例子。比如说：

一种很高深莫测添加文件的方法（程序员是需要bigger的）

打包一个典型的Django应用

设置PyQt4 API级别的 run-time hook（运行时钩）

解决Windows环境下多线程制约的方法

除此之外，还有许许多多由用户贡献的解决方案。请感激这些无私的贡献者为社区做出的贡献。

确定纰漏之处(Finding out What Went Wrong)

Build-time Messages

当PyInstaller执行分析待打包内容的步骤时，会产生一些错误以及警告信息。如果你执行操作前在命令行输入—log-level选项以及打印的日志级别，这些信息就会（部分）展示出来。并且可以通过work-path=参数来让PyInstaller确定把这些日志存放在哪个目录下。

如果发现有模组在代码中被导入，但在实际环境中并没有找到的话，PyInstaller也会记录这样的信息。当然这里的导入方式不仅局限于import somepackage,from … import something的方式，使用在\_\_init\_\_.py文件声明的方式也是会被打包程序所侦测的。因为分析程序在寻找存在主次关系的导入包、模组时，确实会出现找不到的情况。

“module not found”这样的信息是不会被划分为ERROR级别的信息，原因是这种信息一般数量很多。举个例子，很多标准模组会针对系统环境的不同而导入不同的模组。那么如果“一旦找不到”就报ERROR级别的错误，显示是不妥当的。

所有“module not found”的信息都会写入build/name/wanname.txt文件中，但并不会在标准输出中进行打印。你可以看看警告文件，你会发现很多这种模组找不到的信息，但实际上它们并不会最终打包的可执行程序产生什么负面影响。

如果你运行一个已经打包的可执行文件，而它报出Python的ImportError，那你应该查看一下警告文件。然后看看Helping PyInstaller Find Modules章节是怎样处理这个问题的。

依赖图像Build-Time Dependency Graph

当你在命令行中指定—log-level=DEBUG后，PyInstaller会在build目录中写两个关于数据依赖内容的文件。

如果你在工作目录中看到build/name/xref-name.html文件，它的工作是列举所有导入的图像的相关模组等内容。你可以用任何浏览器打开它（因为它是HTML文件啊！！）然后在这个文件中，你可以查询到这些图像内容为什么被导入。（就是需求该数据的最高级模组是哪些）

工作目录中的build/name/graph-name.dot文件是GraphViz输入文件，你可以用GraphViz命令来运行这个文件然后用可视化的方式向你展示所有导入模组的依赖关系。

上面所涉及的文件体积可能比较庞大，因为即便是运行最简单的“你好世界”程序，也需要大量标准模组参与工作。所以，从实用性的角度，产生图像文件可能并没太大的作用。（这意思是，这功能其实就单纯的炫技，你别想多…）

创建错误(Build-Time Python Errors)

PyInstaller有时会被抛出的Python异常所终结。大多数情况下，异常消息都会清晰地告知你原因，比如‘你的系统不支持’，‘PyInstaller需要Python2.7以上版本的支持’等等，这里也提一下，为了Python更好的发展，请转用一些较新的python版本。

在这些抛出的异常中，有一个非常具有迷惑性，就是IOError。PyInstaller需要打包Python一些比较基础的东西，比如说Python解释器的核心、动态加载库的连接等。在不同的系统上，这些内容的存档位置并不相同。在一些情况下可能就没有安装Python动态库（或静态链接存在但不可用）。然后你就需要自己安装一个发开包。

PyInstaller会在不同的地方搜索Python库文件，对于大多数系统而言，/lib /usr/lib是最先被检查的路径。如果你不能把所需要的内容存在放这些路径，则可以尝试通过LD\_LIBARARY\_PATH或者DYLD\_LIBRARY\_PATH来设置正确的环境路径。

获取DEBUG信息(Getting Debug Messages)

当执行打包程序时候添加—debug参数，可以打印一些参数。在开发一个层级逻辑比较复杂的应用时，可以考虑使用这个参数。这样可以让你的程序处于“看起来正常运行”的状态之中。

一般来说，DEBUG过程信息是会打印在标准输出之中的。不过如果在windows上打包应用时添加了—windowed选项，那这些信息就会以MessageBoxes的方式展示。Mac OS X同样。

记得在打包正式版本的时候不要添加—debug选项，不然程序的用户会抓狂的。

获得Python冗长的导入(Getting Python’s Verbose Imports)

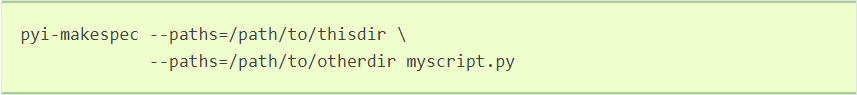
你可以通过传递-v标记(pass a -v flag)来内嵌Python解释器（查看之前的Giving Run-time Python Options）。这种方法非常有效，即便应用当前已经在运行。这样就可以确保打包内容已经取得所有导入内容且没有泄露(leaking out)出本地安装的Python。

如果在命令行添加了—windowed选项，那么警告信息会一直输出到标准输出而非可见的。

帮助PyInstaller寻找模组(Helping PyInstaller Find Modules)

写全路径(Extending the Path)

如果分析程序已经识别了一个代码所需要得而模组，而就是找不到这个模组。很常见的一个原因就是代码在修改sys.path。最有用的方式就是使用—path选项来列出所有代码会去寻找导入内容的位置。



之后这些路径都会被记录在spec文件中，并在分析过程中添加到系统路径中，也就是sys.path。

列出隐式引用（Listing Hidden Imports）

如果分析程序已经认为它找到了所有导入内容，而运行打包程序仍然会提示导入错误的话，那么问题很可能源于隐式导入(hidden import)也就是说在分析阶段导入内容是不可见的。

这种情况会发生在代码使用\_\_import\_\_、exec和eval的情况下。或一个使用Python/C API来导入的额外模组中。这样分析程序就什么都找不到。甚至连警告都没有，就在运行程序时候给你一个生硬的ImportError异常。

为了解决问题，可以在创建应用时添加-v标。

To find these hidden imports, build the app with the -v flag (Getting Python’s Verbose Imports above) and run it.

或者你很了解代码所用的模组会出现这种情况，然后在打包的时候指定—hidden-import参数、写入spec文件或使用钩子文件(可以查看后面的Understanding PyInstaller Hooks章节)

拓展\_\_path\_\_属性(Extending a Package’s \_\_path\_\_)

Python语言有一个机制，就是允许代码通过\_\_path\_\_来添加一些路径搜索所需的导入内容。通常来说，

Python allows a script to extend the search path used for imports through the \_\_path\_\_ mechanism. Normally, the \_\_path\_\_ of an imported module has only one entry, the directory in which the \_\_init\_\_.py was found. But \_\_init\_\_.py is free to extend its \_\_path\_\_ to include other directories. For example, the win32com.shell.shell module actually resolves to win32com/win32comext/shell/shell.pyd. This is because win32com/\_\_init\_\_.py appends ../win32comext to its \_\_path\_\_.

Because the \_\_init\_\_.py of an imported module is not actually executed during analysis, changes it makes to \_\_path\_\_ are not seen by PyInstaller. We fix the problem with the same hook mechanism we use for hidden imports, with some additional logic; see Understanding PyInstaller Hooks below.

Note that manipulations of \_\_path\_\_ hooked in this way apply only to the Analysis. At runtime all imports are intercepted and satisfied from within the bundle. win32com.shell is resolved the same way as win32com.anythingelse, and win32com.\_\_path\_\_ knows nothing of ../win32comext.

Once in a while, that’s not enough.

改变缓存行为(Changing Runtime Behavior)

More bizarre situations can be accomodated with runtime hooks. These are small scripts that manipulate the environment before your main script runs, effectively providing additional top-level code to your script.

There are two ways of providing runtime hooks. You can name them with the option --runtime-hook=path-to-script.

Second, some runtime hooks are provided. At the end of an analysis, the names in the module list produced by the Analysis phase are looked up in loader/rthooks.dat in the PyInstaller install folder. This text file is the string representation of a Python dictionary. The key is the module name, and the value is a list of hook-script pathnames. If there is a match, those scripts are included in the bundled app and will be called before your main script starts.

Hooks you name with the option are executed in the order given, and before any installed runtime hooks. If you specify --runtime-hook=file1.py --runtime-hook=file2.py then the execution order at runtime will be:

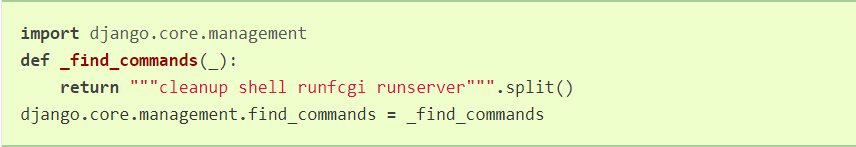
Code of file1.py.

Code of file2.py.

Any hook specified for an included module that is found in rthooks/rthooks.dat.

Your main script.

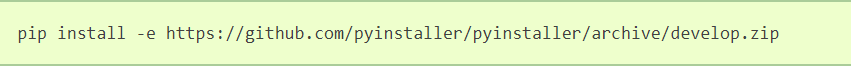
Hooks called in this way, while they need to be careful of what they import, are free to do almost anything. One reason to write a run-time hook is to override some functions or variables from some modules. A good example of this is the Django runtime hook (see loader/rthooks/pyi\_rth\_django.py in the PyInstaller folder). Django imports some modules dynamically and it is looking for some .py files. However .py files are not available in the one-file bundle. We need to override the function django.core.management.find\_commands in a way that will just return a list of values. The runtime hook does this as follows:



装最新的版本(Getting the Latest Version)

If you have some reason to think you have found a bug in PyInstaller you can try downloading the latest development version. This version might have fixes or features that are not yet at PyPI. You can download the latest stable version and the latest development version from the PyInstaller Downloads page.

You can also install the latest version of PyInstaller directly using pip:



主动请求帮助(Asking for Help)

如果上面的内容仍解决不了你的问题，你可以发邮件给我们。还有如果你发现了BUG，可千万别吝啬告知我们，可以在How to Report Bugs页面找到告知BUG的方式。