**概述**

Python 是一门优美简单、功能强大的动态语言。在刚刚接触这门语言时，我们会被其优美的格式、简洁的语法和无穷无尽的类库所震撼。在真正的将python应用到实际的项目中，你会遇到一些无法避免的问题。最让人困惑不解的问题有二类，一个 编码问题，另一个则是引用问题。

本文主要讨论关于Python中import的机制与实现、以及介绍一些有意思的Python Hooks。

**Python 类库引入机制**

首先，看一个简单的例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | """  目录结构如下：  ├── \_\_init\_\_.py  ├── main.py  └── string.py  """  # main.py 内容如下  import string  print string.a  # string.py 内容如下  a = 2 |

现在，考虑一下：

1. 当我们执行main.py的时候，会发生什么事情？
2. 在main.py文件执行到 import string 的时候，解释器导入的string类库是当前文件夹下的string.py还是系统标准库的string.py呢？
3. 如果明确的指明⾃己要引⼊的类库？

为了搞清楚上面的问题，我们需要了解关于Python类库引入的机制。

**Python的两种引入机制**

Python 提供了二种引入机制：

1. relative import
2. absolute import

**relative import**

relative import 也叫作相对引入，在Python2.5及之前是默认的引入方法。它的使用方法如下：

Python

|  |  |
| --- | --- |
| 1  2  3 | from .string import a  from ..string import a  from ...string import a |

这种引入方式使用一个点号来标识引入类库的精确位置。与linux的相对路径表示相似，一个点表示当前目录，每多一个点号则代表向上一层目录。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | """  ├── \_\_init\_\_.py  ├── foo.py  └── main.py  """  # foo.py  a = 2  # main.py  print \_\_name\_\_  from .foo import a  print a |

相对引入，那么我们需要知道相对什么来引入。相对引入使用被引入文件的 \_\_name\_\_ 属性来决定该文件在整个包结构的位置。那么如果文件的\_\_name\_\_没有包含任何包的信息，例如\_\_name\_\_ 被设置为了\_\_main\_\_，则认为其为‘top level script’，而不管该文件的位置，这个时候相对引入就没有引入的参考物。如上面的程序所示，当我们执行 pythonmain.py 时，Python解释器会抛出 ValueError: Attempted relative import in non-package的异常。

为了解决这个问题，[PEP 0366 — Main module explicit relative imports](https://www.python.org/dev/peps/pep-0366/)提出了一个解决方案。允许用户使用python -m ex2.main的方式,来执行该文件。在这个方案下，引入了一个新的属性\_\_package\_\_。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | liuchang@localhost  ~/Codes/pycon  $ cat ex2/main.py  print \_\_name\_\_  print \_\_package\_\_  from .foo import a  print a  liuchang@localhost  ~/Codes/pycon  $ python -m ex2.main  \_\_main\_\_  ex2  2 |

**absolute import**

absolute import 也叫作完全引入，非常类似于Java的引入进制，在Python2.5被完全实现，但是是需要通过 from \_\_future\_\_ import absolute\_import 来打开该引入进制。在Python2.6之后以及Python3，完全引用成为Python的默认的引入机制。它的使用方法如下：

Python

|  |  |
| --- | --- |
| 1  2 | from pkg import foo  from pkg.moduleA import foo |

要注意的是，需要从包目录最顶层目录依次写下，而不能从中间开始。

在使用该引入方式时，我们碰到比较多的问题就是因为位置原因，Python找不到相应的库文件，抛出ImportError的异常。让我们看一个完全引用的例子:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | """  ex3  ├── \_\_init\_\_.py  ├── foo.py  └── main.py  """  # foo.py  a = 2    # main.py  print \_\_name\_\_  print \_\_package\_\_  from ex2.foo import a  print a |

我们尝试着去运行main.py文件，Python解释器会抛出ImportError。那么我们如何解决这个问题呢？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ python ex3/main.py  \_\_main\_\_  None  Traceback (most recent call last):    File "ex3/main.py", line 3, in <module>      from ex2.foo import a  ImportError: No module named ex2.foo |

首先，我们也可以使用前文所述的module的方式去运行程序，通过-m参数来告诉解释器 \_\_package\_\_ 属性。如下：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5 | liuchang@liuchangdeMacBook-Pro  ~/Codes/pycon  $ python -m ex3.main  \_\_main\_\_  ex3  2 |

另外，我们还有一个办法可以解决该问题，在描述之前，我们介绍一个关于Python的非常有用的小知识：**Python解释器会自动将当前工作目录添加到PYTHONPATH**。如下所示，可以看到我们打印出的 sys.path 已经包含了当前工作目录。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | liuchang@liuchangdeMacBook-Pro  ~/Codes/pycon/ex4  $ cat main.py  import sys  print sys.path  liuchang@liuchangdeMacBook-Pro  ~/Codes/pycon/ex4  $ python main.py  ['/Users/liuchang/Codes/pycon/ex4', '/Library/Python/2.7/site-packages/pip-7.1.0-py2.7.egg', '/Library/Python/2.7/site-packages/mesos-\_PACKAGE\_VERSION\_-py2.7.egg', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python27.zip', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-darwin', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac/lib-scriptpackages', '/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-tk', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-old', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-dynload', '/Users/liuchang/Library/Python/2.7/lib/python/site-packages', '/usr/local/lib/python2.7/site-packages', '/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python/PyObjC', '/Library/Python/2.7/site-packages'] |

了解了Python解释器的这个特性后，我们就可以解决完全引用的找不到类库的问题：执行的时候，让解释器自动的将类库的目录添加到PYTHONPATH中。

我们可以在顶层目录中添加一个run\_ex3.py的文件，文件内容和运行结果如下，可以看到Python解释器正确的执行了ex3.main文件。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | liuchang@liuchangdeMacBook-Pro  ~/Codes/pycon  $ cat run\_ex3.py  from ex3 import main  liuchang@liuchangdeMacBook-Pro  ~/Codes/pycon  $ python run\_ex3.py  ex3.main  None  2 |

**一些实践经验**

**相对引用还是绝对引用？**

上面介绍了Python的两种引用方式，都可以解决引入歧义的问题。那我们应该使用哪一种呢？

先说明一下Python的默认引用方式，在Python2.4及之前，Python只有相对引用这一种方式，在Python2.5中实现了绝对引用，但默认没有打开，需要用户自己指定使用该引用方式。在之后的版本和Python3版本，绝对引用已经成为默认的引用方式。

其次，二种引用方式各有利弊。绝对引用代码更加清晰明了，可以清楚的看到引入的包名和层次，但是，当包名修改的时候，我们需要手动修改所有的引用代码。相对引用则比较精简，不会被包名修改所影响，但是可读性较差，不如完全引用清晰。

最后，对于两种引用的方式选择，还是有争论的。在PEP8中，Python官方推荐的是绝对引用,详细理由可以参考[这儿](https://www.python.org/dev/peps/pep-0008/#imports)。

Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on sys.path ):

Python

|  |  |
| --- | --- |
| 1  2  3 | import mypkg.sibling  from mypkg import sibling  from mypkg.sibling import example |

However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:

Python

|  |  |
| --- | --- |
| 1  2 | from . import sibling  from .sibling import example |

Standard library code should avoid complex package layouts and always use absolute imports. Implicit relative imports should never be used and have been removed in Python 3.

**规范打包发布**

为了别人使用自己代码的方便，应该尽量使用规范的包分发机制。为自己的Python包编写正确的setup.py文件，添加相应的README.md文件。对于提供一些可执行命令的包，则可以使用 console\_entrypoint 的机制来提供。因为打包和分发不是本文重点，不再详细叙述，大家可以查看官方文档。

**使用virtualenv管理包依赖**

在使用Python的时候，尽量使用virtualenv来管理项目，所有的项目从编写到运行都在特定的virtualenv中。并且为自己的项目生成正确的依赖描述文件。

|  |  |
| --- | --- |
| 1 | pip freeze > requirements.txt |

关于virtualenv的用法，可以参考我之前的一篇文章[virtualenv教程](http://lcblog-wordpress.stor.sinaapp.com/uploads/2015/10/virtualenv%E6%95%99%E7%A8%8B.pdf)。

**Python import实现**

Python 提供了 import 语句来实现类库的引用，下面我们详细介绍当执行了 import 语句的时候，内部究竟做了些什么事情。

当我们执行一行  from package import module as mymodule 命令时，Python解释器会查找package这个包的module模块，并将该模块作为mymodule引入到当前的工作空间。所以import语句主要是做了二件事：

1. 查找相应的module
2. 加载module到local namespace

下面我们详细了解python是如何查找模块的。

**查找module的过程**

在import的第一个阶段，主要是完成了查找要引入模块的功能，这个查找的过程如下：

1. 检查 sys.modules (保存了之前import的类库的缓存），如果module被找到，则⾛到第二步。
2. 检查 sys.meta\_path。meta\_path 是一个 list，⾥面保存着一些 finder 对象，如果找到该module的话，就会返回一个finder对象。
3. 检查⼀些隐式的finder对象，不同的python实现有不同的隐式finder，但是都会有 sys.path\_hooks, sys.path\_importer\_cache 以及sys.path。
4. 抛出 ImportError。

**sys.modules**

对于第一步中sys.modules，我们可以打开Python来实际的查看一下其内容：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Python 2.7.10 (default, Aug 22 2015, 20:33:39)  [GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.1)] on darwin  Type "help", "copyright", "credits" or "license" for more information.  >> import sys  >> sys.modules  {'copy\_reg': <module 'copy\_reg' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/copy\_reg.pyc'>, 'sre\_compile': <module 'sre\_compile' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/sre\_compile.pyc'>, '\_sre': <module '\_sre' (built-in)>, 'encodings': <module 'encodings' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/encodings/\_\_init\_\_.pyc'>, 'site': <module 'site' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site.pyc'>, '\_\_builtin\_\_': <module '\_\_builtin\_\_' (built-in)>, 'sysconfig': <module 'sysconfig' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/sysconfig.pyc'>, 'encodings.encodings': None, '\_\_main\_\_': <module '\_\_main\_\_' (built-in)>, 'supervisor': <module 'supervisor' (built-in)>, 'abc': <module 'abc' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/abc.pyc'>, 'posixpath': <module 'posixpath' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/posixpath.pyc'>, '\_weakrefset': <module '\_weakrefset' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/\_weakrefset.pyc'>, 'errno': <module 'errno' (built-in)>, 'encodings.codecs': None, 'sre\_constants': <module 'sre\_constants' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/sre\_constants.pyc'>, 're': <module 're' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/re.pyc'>, '\_abcoll': <module '\_abcoll' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/\_abcoll.pyc'>, 'types': <module 'types' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/types.pyc'>, '\_codecs': <module '\_codecs' (built-in)>, 'encodings.\_\_builtin\_\_': None, '\_warnings': <module '\_warnings' (built-in)>, 'genericpath': <module 'genericpath' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/genericpath.pyc'>, 'stat': <module 'stat' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/stat.pyc'>, 'zipimport': <module 'zipimport' (built-in)>, '\_sysconfigdata': <module '\_sysconfigdata' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/\_sysconfigdata.pyc'>, 'mpl\_toolkits': <module 'mpl\_toolkits' (built-in)>, 'warnings': <module 'warnings' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/warnings.pyc'>, 'UserDict': <module 'UserDict' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/UserDict.pyc'>, 'encodings.utf\_8': <module 'encodings.utf\_8' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/encodings/utf\_8.pyc'>, 'sys': <module 'sys' (built-in)>, '\_osx\_support': <module '\_osx\_support' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/\_osx\_support.pyc'>, 'codecs': <module 'codecs' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/codecs.pyc'>, 'readline': <module 'readline' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-dynload/readline.so'>, 'os.path': <module 'posixpath' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/posixpath.pyc'>, '\_locale': <module '\_locale' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-dynload/\_locale.so'>, 'signal': <module 'signal' (built-in)>, 'traceback': <module 'traceback' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/traceback.pyc'>, 'linecache': <module 'linecache' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/linecache.pyc'>, 'posix': <module 'posix' (built-in)>, 'encodings.aliases': <module 'encodings.aliases' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/encodings/aliases.pyc'>, 'exceptions': <module 'exceptions' (built-in)>, 'sre\_parse': <module 'sre\_parse' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/sre\_parse.pyc'>, 'os': <module 'os' from '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/os.pyc'>, '\_weakref': <module '\_weakref' (built-in)>}  >> sys.modules['zlib'].\_\_file\_\_  '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-dynload/zlib.so' |

可以看到sys.modules已经保存了一些包的信息，由这些信息，我们就可以直接知道要查找的包的位置等信息。

**finder、loader和importer**

在上文中，我们提到了sys.meta\_path中保证了一些finder对象。在python中，不仅定义了finder的概念，还定义了loader和importor的概念。

* finder的任务是决定自己是否根据名字找到相应的模块，在py2中，finder对象必须实现find\_module()方法，在py3中必须要实现find\_module()或者find\_loader（)方法。如果finder可以查找到模块，则会返回一个loader对象(在py3.4中，修改为返回一个module specs)。
* loader则是负责加载模块，它必须实现一个load\_module()的方法。
* importer 则指一个对象，实现了finder和loader的方法。因为Python是duck type，只要实现了方法，就可以认为是该类。

**sys.meta\_path**

在Python查找的时候，如果在sys.modules没有查找到，就会依次调用sys.meta\_path中的finder对象。默认的情况下，sys.meta\_path是一个空列表，并没有任何finder对象。

Python

|  |  |
| --- | --- |
| 1  2 | In [6]: sys.meta\_path  Out[6]: [] |

我们可以向sys.meta\_path中添加一些定义的finder，来实现对Python加载模块的修改。比如下例，我们实现了一个会将每次加载包的信息打印出来的finder。

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | from \_\_future\_\_ import print\_function  import sys    class Watcher(object):      @classmethod      def find\_module(cls, name, path, target=None):          print("Importing", name, path, target)          return None    sys.meta\_path.insert(0, Watcher)    import socket |

当我们执行的时候，就可以看到系统加载socket包时所发生的事情。

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | liuchang@localhost  ~/Codes/pycon/ex5\_meta\_path  $ python finder1.py  Importing socket None None  Importing \_socket None None  Importing functools None None  Importing \_functools None None  Importing \_ssl None None  Importing cStringIO None None |

**sys.path hook**

Python import的hook分为二类，一类是上一章节已经描述的meta hook，另一类是 path hook。

当处理sys.path（或者package.**path**)时，就会调用对应的一部分的 Pack hook。Path Hook是通过向sys.path\_hooks 中添加一个importer生成器来注册的。

sys.path\_hooks 是由可被调用的对象组成，它会顺序的检查以决定他们是否可以处理给定的sys.path的一项。每个对象会使用sys.path项的路径来作为参数被调用。如果它不能处理该路径，就必须抛出ImportError，如果可以，则会返回一个importer对象。之后，不会再尝试其它的sys.path\_hooks对象，即使前一个importer出错了。

详细可以参考[registering-hooks](https://www.python.org/dev/peps/pep-0302/#specification-part-2-registering-hooks)。

**python import hooks**

在介绍完Python的引用机制与一些实现方法后，接下来我们介绍一些关于如何根据自己的需求来扩展Python的引用机制。

在开始详细介绍前，给大家展示一个实用性不高，但是很有意思的例子：**让Python在执行代码的时候自动安装缺失的类库**。我们会实现一个autoinstall的模块，只要import了该模块，就可以打开该功能。如下所示，我们尝试引入tornado库的时候，iPython会提示我们没有安装。然后，我们引入了autoinstall，再尝试引入tornado，iPython就会自动的安装tornado库。

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | In [1]: import tornado  ---------------------------------------------------------------------------  ImportError                               Traceback (most recent call last)  <ipython-input-1-3eac10687b7e> in <module>()  ----> 1 import tornado    ImportError: No module named tornado    In [2]: import autoinstall    In [3]: import tornado  Installing tornado    Collecting tornado    Downloading tornado-4.2.1.tar.gz (434kB)  Collecting backports.ssl-match-hostname (from tornado)    Downloading http://182.92.2.186:7002/packages/backports.ssl\_match\_hostname-3.4.0.2-py2-none-any.whl  Collecting certifi (from tornado)    Downloading certifi-2015.9.6.2-py2.py3-none-any.whl (371kB)  Installing collected packages: backports.ssl-match-hostname, certifi, tornado    Running setup.py install for tornado  Successfully installed backports.ssl-match-hostname-3.4.0.2 certifi-2015.9.6.2 tornado-4.2.1 |

这个功能的实现其实很简单，利用了sys.meta\_path。autoinstall的全部代码如下：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | from \_\_future\_\_ import print\_function  import sys  import subprocess    class AutoInstall(object):      \_loaded = set()        @classmethod      def find\_module(cls, name, path, target=None):          if path is None and name not in cls.\_loaded:              cls.\_loaded.add(name)              print("Installing", name)              try:                  out = subprocess.check\_output(['sudo', sys.executable, '-m', 'pip', 'install', name])                  print(out)              except Exception as e:                  print("Failed" + e.message)          return None    sys.meta\_path.append(AutoInstall) |

**import hook的重要性**

我们为什么需要Python import的hook呢？使用import的hook可以让我们做到很多事情，比如说当我们的Python包存储在一个非标准的文件中，或者Python程序存储在网络数据库中，或者像py2exe一样将Python程序打包成了一个文件，我们需要一种方法来正确的解析它们。

其次，我们希望在Python加载类库的时候，可以额外的做一些事情，比如上传审计信息，比如延迟加载，比如自动解决上例的依赖未安装的问题。

所以，import系统的Hook技术是值的花时间学习的。

**如何实现import hooks**

Python提供了一些方法，让我们可以在代码中动态的调用import。主要有如下几种：

1. \_\_import\_\_ : Python的内置函数
2. imputil : Python的import工具库，在py2.6被声明废弃，py3中彻底移除。
3. imp : Python2 的一个import库，py3中移除
4. importlib : Python3 中最新添加，backport到py2.7，但只有很小的子集（只有一个函数）。

Python2 所有关于import的库的列表参见[Importing Modules](https://docs.python.org/2/library/modules.html)。Python3 的可以参考[Importing Modules](https://docs.python.org/3/library/modules.html) [PEP 0302 — New Import Hooks](https://www.python.org/dev/peps/pep-0302) 提案详细的描述了importlib的目的、用法。

**一些Hook示例**

**Lazy化库引入**

使用Import Hook，我们可以达到Lazy Import的效果，当我们执行import的时候，实际上并没引入该库，只有真正的使用这个库的时候，才会将其引入到当前工作空间。 具体的代码可以参考[github](https://github.com/noahmorrison/limp)。 实现的效果如下：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #!/usr/bin/python    import limp  # Lazy imports begin now    import json  import sys    print('json' in sys.modules)  # False  print(', '.join(json.loads('["Hello", "World!"]')))  print('json' in sys.modules)  # True |

它的实现也很简单：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | import sys  import imp    \_lazy\_modules = {}    class LazyModule():      def \_\_init\_\_(self, name):          self.name = name        def \_\_getattr\_\_(self, attr):          path = \_lazy\_modules[self.name]          f, pathname, desc = imp.find\_module(self.name, path)            lf = sys.meta\_path.pop()          imp.load\_module(self.name, f, pathname, desc)          sys.meta\_path.append(lf)            self.\_\_dict\_\_ = sys.modules[self.name].\_\_dict\_\_          return self.\_\_dict\_\_[attr]    class LazyFinder(object):        def find\_module(self, name, path):          \_lazy\_modules[name] = path          return self        def load\_module(self, name):          return LazyModule(name)    sys.meta\_path.append(LazyFinder()) |

**Flask 插件库统一入口**

使用过Flask的同学都知道，Flask的对于插件提供了统一的入口。比如说我们安装了Flask\_API这个库，然后我们可以直接 import flask\_api 来使用这个库，同时Flask还允许我们采用 import flask.ext.api 的方式来引用该库。

这里Flask就是使用了import 的hook，当引入flask.ext的包时，就自动的引用相应的库。Flask实现了一个叫ExtensionImporter的类，这个类实现了find\_module和load\_module代码实现如下[github](https://github.com/mitsuhiko/flask/blob/master/flask/exthook.py#L27)：

Python

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | class ExtensionImporter(object):      """This importer redirects imports from this submodule to other locations.      This makes it possible to transition from the old flaskext.name to the      newer flask\_name without people having a hard time.      """        def \_\_init\_\_(self, module\_choices, wrapper\_module):          self.module\_choices = module\_choices          self.wrapper\_module = wrapper\_module          self.prefix = wrapper\_module + '.'          self.prefix\_cutoff = wrapper\_module.count('.') + 1        def \_\_eq\_\_(self, other):          return self.\_\_class\_\_.\_\_module\_\_ == other.\_\_class\_\_.\_\_module\_\_ and \                 self.\_\_class\_\_.\_\_name\_\_ == other.\_\_class\_\_.\_\_name\_\_ and \                 self.wrapper\_module == other.wrapper\_module and \                 self.module\_choices == other.module\_choices        def \_\_ne\_\_(self, other):          return not self.\_\_eq\_\_(other)        def install(self):          sys.meta\_path[:] = [x for x in sys.meta\_path if self != x] + [self]        def find\_module(self, fullname, path=None):          if fullname.startswith(self.prefix):              return self        def load\_module(self, fullname):          if fullname in sys.modules:              return sys.modules[fullname]          modname = fullname.split('.', self.prefix\_cutoff)[self.prefix\_cutoff]          for path in self.module\_choices:              realname = path % modname              try:                  \_\_import\_\_(realname)              except ImportError:                  exc\_type, exc\_value, tb = sys.exc\_info()                  # since we only establish the entry in sys.modules at the                  # very this seems to be redundant, but if recursive imports                  # happen we will call into the move import a second time.                  # On the second invocation we still don't have an entry for                  # fullname in sys.modules, but we will end up with the same                  # fake module name and that import will succeed since this                  # one already has a temporary entry in the modules dict.                  # Since this one "succeeded" temporarily that second                  # invocation now will have created a fullname entry in                  # sys.modules which we have to kill.                  sys.modules.pop(fullname, None)                    # If it's an important traceback we reraise it, otherwise                  # we swallow it and try the next choice.  The skipped frame                  # is the one from \_\_import\_\_ above which we don't care about                  if self.is\_important\_traceback(realname, tb):                      reraise(exc\_type, exc\_value, tb.tb\_next)                  continue              module = sys.modules[fullname] = sys.modules[realname]              if '.' not in modname:                  setattr(sys.modules[self.wrapper\_module], modname, module)              return module          raise ImportError('No module named %s' % fullname) |

然后在Flask的ext目录下的\_\_init\_\_.py文件中，初始化了该Importer。

Python

|  |  |
| --- | --- |
| 1  2  3  4 | def setup():      from ..exthook import ExtensionImporter      importer = ExtensionImporter(['flask\_%s', 'flaskext.%s'], \_\_name\_\_)      importer.install() |