10.1 构建一个模块的层级包¶

问题¶

你想将你的代码组织成由很多分层模块构成的包。

解决方案¶

封装成包是很简单的。在文件系统上组织你的代码，并确保每个目录都定义了一个__init__.py文件。 例如：

```
graphics/
    __init__.py
    primitive/
        __init__.py
        line.py
        fill.py
        text.py
    formats/
        __init__.py
        png.py
        jpg.py
```

一旦你做到了这一点，你应该能够执行各种import语句，如下：

```
import graphics.primitive.line
from graphics.primitive import line
import graphics.formats.jpg as jpg
```

讨论¶

定义模块的层次结构就像在文件系统上建立目录结构一样容易。 文件__init__.py的目的是要包含不同运行级别的包的可选的初始化代码。 举个例子，如果你执行了语句import graphics， 文件graphics/__init__.py将被导入,建立graphics命名空间的内容。像import graphics.format.jpg这样导入，文件graphics/__init__.py和文件graphics/formats/__init__.py将在文件graphics/formats/jpg.py导入之前导入。

绝大部分时候让__init__.py空着就好。但是有些情况下可能包含代码。 举个例子，__init__.py能够用来自动加载子模块:

```
# graphics/formats/__init__.py
from . import jpg
from . import png
```

像这样一个文件,用户可以仅仅通过import grahpics.formats来代替import graphics.formats.jpg以及import graphics.formats.png。

__init__.py的其他常用用法包括将多个文件合并到一个逻辑命名空间，这将在10.4小节讨论。

敏锐的程序员会发现，即使没有__init__.py文件存在，python仍然会导入包。如果你没有定义__init__.py时，实际上创建了一个所谓的“命名空间包”，这将在10.5小节讨论。万物平等，如果你着手创建一个新的包的话，包含一个__init__.py文件吧。

10.10 通过字符串名导入模块¶

问题¶

你想导入一个模块，但是模块的名字在字符串里。你想对字符串调用导入命令。

解决方案¶

使用importlib.import_module()函数来手动导入名字为字符串给出的一个模块或者包的一部分。举个例子：

```
>>> import importlib
>>> math = importlib.import_module('math')
>>> math.sin(2)
0.9092974268256817
>>> mod = importlib.import_module('urllib.request')
>>> u = mod.urlopen('http://www.python.org')
>>>
```

import_module只是简单地执行和import相同的步骤，但是返回生成的模块对象。你只需要将其存储在一个变量，然后像正常的模块一样使用。

如果你正在使用的包，import_module()也可用于相对导入。但是，你需要给它一个额外的参数。例如：

```
import importlib
# Same as 'from . import b'
b = importlib.import_module('.b', __package__)
```

讨论¶

使用import_module()手动导入模块的问题通常出现在以某种方式编写修改或覆盖模块的代码时候。例如，也许你正在执行某种自定义导入机制，需要通过名称来加载一个模块，通过补丁加载代码。

在旧的代码，有时你会看到用于导入的内建函数__import__()。尽管它能工作，但是importlib.import_module() 通常更容易使用。

自定义导入过程的高级实例见10.11小节

10.11 通过钩子远程加载模块¶

问题¶

你想自定义Python的import语句，使得它能从远程机器上面透明的加载模块。

解决方案¶

首先要提出来的是安全问题。本节讨论的思想如果没有一些额外的安全和认知机制的话会很糟糕。 也就是说，我们的主要目的是深入分析Python的import语句机制。 如果你理解了本节内部原理，你就能够为其他任何目的而自定义import。 有了这些，让我们继续向前走。

本节核心是设计导入语句的扩展功能。有很多种方法可以做这个， 不过为了演示的方便，我们开始先构造下面这个Python代码结构：

```
testcode/
    spam.py
    fib.py
    grok/
        __init__.py
        blah.py
```

这些文件的内容并不重要，不过我们在每个文件中放入了少量的简单语句和函数， 这样你可以测试它们并查看当它们被导入时的输出。例如：

```
# spam.py
print("I'm spam")

def hello(name):
    print('Hello %s' % name)

# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

# grok/__init__.py
print("I'm grok.__init__")

# grok/blah.py
print("I'm grok.blah")
```

这里的目的是允许这些文件作为模块被远程访问。 也许最简单的方式就是将它们发布到一个web服务器上面。在testcode目录中像下面这样运行Python：

```
bash % cd testcode
bash % python3 -m http.server 15000
Serving HTTP on 0.0.0.0 port 15000 ...
```

服务器运行起来后再启动一个单独的Python解释器。 确保你可以使用 `urllib` 访问到远程文件。例如：

```
>>> from urllib.request import urlopen
>>> u = urlopen('http://localhost:15000/fib.py')
>>> data = u.read().decode('utf-8')
>>> print(data)
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
```

```
        return 1
    else:
        return fib(n-1) + fib(n-2)
>>>
```

从这个服务器加载源代码是接下来本节的基础。为了替代手动的通过 `urlopen()` 来收集源文件，我们通过自定义import语句来在后台自动帮我们做到。

加载远程模块的第一种方法是创建一个显式的加载函数来完成它。例如：

```
import imp
import urllib.request
import sys

def load_module(url):
    u = urllib.request.urlopen(url)
    source = u.read().decode('utf-8')
    mod = sys.modules.setdefault(url, imp.new_module(url))
    code = compile(source, url, 'exec')
    mod.__file__ = url
    mod.__package__ = ''
    exec(code, mod.__dict__)
    return mod
```

这个函数会下载源代码，并使用 `compile()` 将其编译到一个代码对象中，然后在一个新创建的模块对象的字典中来执行它。下面是使用这个函数的方式：

```
>>> fib = load_module('http://localhost:15000/fib.py')
I'm fib
>>> fib.fib(10)
89
>>> spam = load_module('http://localhost:15000/spam.py')
I'm spam
>>> spam.hello('Guido')
Hello Guido
>>> fib
<module 'http://localhost:15000/fib.py' from 'http://localhost:15000/fib.py'>
>>> spam
<module 'http://localhost:15000/spam.py' from 'http://localhost:15000/spam.py'>
>>>
```

正如你所见，对于简单的模块这个是行得通的。不过它并没有嵌入到通常的import语句中，如果要支持更高级的结构比如包就需要更多的工作了。

一个更酷的做法是创建一个自定义导入器。第一种方法是创建一个元路径导入器。如下：

```
# urlimport.py
import sys
import importlib.abc
import imp
from urllib.request import urlopen
from urllib.error import HTTPError, URLError
from html.parser import HTMLParser

# Debugging
import logging
log = logging.getLogger(__name__)

# Get links from a given URL
def _get_links(url):
    class LinkParser(HTMLParser):
        def handle_starttag(self, tag, attrs):
            if tag == 'a':
                attrs = dict(attrs)
                links.add(attrs.get('href').rstrip('/'))
    links = set()
    try:
        log.debug('Getting links from %s' % url)
        u = urlopen(url)
```

```
        parser = LinkParser()
        parser.feed(u.read().decode('utf-8'))
    except Exception as e:
        log.debug('Could not get links. %s', e)
    log.debug('links: %r', links)
    return links

class UrlMetaFinder(importlib.abc.MetaPathFinder):
    def __init__(self, baseurl):
        self._baseurl = baseurl
        self._links = { }
        self._loaders = { baseurl : UrlModuleLoader(baseurl) }

    def find_module(self, fullname, path=None):
        log.debug('find_module: fullname=%r, path=%r', fullname, path)
        if path is None:
            baseurl = self._baseurl
        else:
            if not path[0].startswith(self._baseurl):
                return None
            baseurl = path[0]
        parts = fullname.split('.')
        basename = parts[-1]
        log.debug('find_module: baseurl=%r, basename=%r', baseurl, basename)

        # Check link cache
        if basename not in self._links:
            self._links[baseurl] = _get_links(baseurl)

        # Check if it's a package
        if basename in self._links[baseurl]:
            log.debug('find_module: trying package %r', fullname)
            fullurl = self.base_url + '/' + basename
            # Attempt to load the package (which accesses __init__.py)
            loader = UrlPackageLoader(fullurl)
            try:
                loader.load_module(fullname)
                self._links[fullurl] = _get_links(fullurl)
                self._loaders[fullurl] = UrlModuleLoader(fullurl)
                log.debug('find_module: package %r loaded', fullname)
            except ImportError as e:
                log.debug('find_module: package failed. %s', e)
                loader = None
            return loader
        # A normal module
        filename = basename + '.py'
        if filename in self._links[baseurl]:
            log.debug('find_module: module %r found', fullname)
            return self._loaders[baseurl]
        else:
            log.debug('find_module: module %r not found', fullname)
            return None

    def invalidate_caches(self):
        log.debug('invalidating link cache')
        self._links.clear()

# Module Loader for a URL
class UrlModuleLoader(importlib.abc.SourceLoader):
    def __init__(self, baseurl):
        self._baseurl = baseurl
        self._source_cache = {}

    def module_repr(self, module):
        return '<urlmodule %r from %r>' % (module.__name__, module.__file__)

    # Required method
    def load_module(self, fullname):
        code = self.get_code(fullname)
        mod = sys.modules.setdefault(fullname, imp.new_module(fullname))
        mod.__file__ = self.get_filename(fullname)
```

```
        mod.__loader__ = self
        mod.__package__ = fullname.rpartition('.')[0]
        exec(code, mod.__dict__)
        return mod

    # Optional extensions
    def get_code(self, fullname):
        src = self.get_source(fullname)
        return compile(src, self.get_filename(fullname), 'exec')

    def get_data(self, path):
        pass

    def get_filename(self, fullname):
        return self._baseurl + '/' + fullname.split('.')[-1] + '.py'

    def get_source(self, fullname):
        filename = self.get_filename(fullname)
        log.debug('loader: reading %r', filename)
        if filename in self._source_cache:
            log.debug('loader: cached %r', filename)
            return self._source_cache[filename]
        try:
            u = urlopen(filename)
            source = u.read().decode('utf-8')
            log.debug('loader: %r loaded', filename)
            self._source_cache[filename] = source
            return source
        except (HTTPError, URLError) as e:
            log.debug('loader: %r failed. %s', filename, e)
            raise ImportError("Can't load %s" % filename)

    def is_package(self, fullname):
        return False

# Package loader for a URL
class UrlPackageLoader(UrlModuleLoader):
    def load_module(self, fullname):
        mod = super().load_module(fullname)
        mod.__path__ = [ self._baseurl ]
        mod.__package__ = fullname

    def get_filename(self, fullname):
        return self._baseurl + '/' + '__init__.py'

    def is_package(self, fullname):
        return True

# Utility functions for installing/uninstalling the loader
_installed_meta_cache = { }
def install_meta(address):
    if address not in _installed_meta_cache:
        finder = UrlMetaFinder(address)
        _installed_meta_cache[address] = finder
        sys.meta_path.append(finder)
        log.debug('%r installed on sys.meta_path', finder)

def remove_meta(address):
    if address in _installed_meta_cache:
        finder = _installed_meta_cache.pop(address)
        sys.meta_path.remove(finder)
        log.debug('%r removed from sys.meta_path', finder)
```

下面是一个交互会话，演示了如何使用前面的代码：

```
>>> # importing currently fails
>>> import fib
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> # Load the importer and retry (it works)
```

```
>>> import urlimport
>>> urlimport.install_meta('http://localhost:15000')
>>> import fib
I'm fib
>>> import spam
I'm spam
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>
```

这个特殊的方案会安装一个特别的查找器 `UrlMetaFinder` 实例， 作为 `sys.meta_path` 中最后的实体。 当模块被导入时，会依据 `sys.meta_path` 中的查找器定位模块。 在这个例子中，`UrlMetaFinder` 实例是最后一个查找器方案， 当模块在任何一个普通地方都找不到的时候就触发它。

作为常见的实现方案，`UrlMetaFinder` 类包装在一个用户指定的URL上。 在内部，查找器通过抓取指定URL的内容构建合法的链接集合。 导入的时候，模块名会跟已有的链接作对比。如果找到了一个匹配的， 一个单独的 `UrlModuleLoader` 类被用来从远程机器上加载源代码并创建最终的模块对象。 这里缓存链接的一个原因是避免不必要的HTTP请求重复导入。

自定义导入的第二种方法是编写一个钩子直接嵌入到 `sys.path` 变量中去， 识别某些目录命名模式。 在 `urlimport.py` 中添加如下的类和支持函数：

```
# urlimport.py
# ... include previous code above ...
# Path finder class for a URL
class UrlPathFinder(importlib.abc.PathEntryFinder):
    def __init__(self, baseurl):
        self._links = None
        self._loader = UrlModuleLoader(baseurl)
        self._baseurl = baseurl

    def find_loader(self, fullname):
        log.debug('find_loader: %r', fullname)
        parts = fullname.split('.')
        basename = parts[-1]
        # Check link cache
        if self._links is None:
            self._links = [] # See discussion
            self._links = _get_links(self._baseurl)

        # Check if it's a package
        if basename in self._links:
            log.debug('find_loader: trying package %r', fullname)
            fullurl = self._baseurl + '/' + basename
            # Attempt to load the package (which accesses __init__.py)
            loader = UrlPackageLoader(fullurl)
            try:
                loader.load_module(fullname)
                log.debug('find_loader: package %r loaded', fullname)
            except ImportError as e:
                log.debug('find_loader: %r is a namespace package', fullname)
                loader = None
            return (loader, [fullurl])

        # A normal module
        filename = basename + '.py'
        if filename in self._links:
            log.debug('find_loader: module %r found', fullname)
            return (self._loader, [])
        else:
            log.debug('find_loader: module %r not found', fullname)
            return (None, [])

    def invalidate_caches(self):
        log.debug('invalidating link cache')
        self._links = None
```

```
# Check path to see if it looks like a URL
_url_path_cache = {}
def handle_url(path):
    if path.startswith(('http://', 'https://')):
        log.debug('Handle path? %s. [Yes]', path)
        if path in _url_path_cache:
            finder = _url_path_cache[path]
        else:
            finder = UrlPathFinder(path)
            _url_path_cache[path] = finder
        return finder
    else:
        log.debug('Handle path? %s. [No]', path)

def install_path_hook():
    sys.path_hooks.append(handle_url)
    sys.path_importer_cache.clear()
    log.debug('Installing handle_url')

def remove_path_hook():
    sys.path_hooks.remove(handle_url)
    sys.path_importer_cache.clear()
    log.debug('Removing handle_url')
```

要使用这个路径查找器，你只需要在 `sys.path` 中加入URL链接。例如：

```
>>> # Initial import fails
>>> import fib
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Install the path hook
>>> import urlimport
>>> urlimport.install_path_hook()

>>> # Imports still fail (not on path)
>>> import fib
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Add an entry to sys.path and watch it work
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
I'm fib
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>
```

关键点就是 `handle_url()` 函数，它被添加到了 `sys.path_hooks` 变量中。 当 `sys.path` 的实体被处理时，会调用 `sys.path_hooks` 中的函数。 如果任何一个函数返回了一个查找器对象，那么这个对象就被用来为 `sys.path` 实体加载模块。

远程模块加载跟其他的加载使用方法几乎是一样的。例如：

```
>>> fib
<urlmodule 'fib' from 'http://localhost:15000/fib.py'>
>>> fib.__name__
'fib'
>>> fib.__file__
'http://localhost:15000/fib.py'
>>> import inspect
>>> print(inspect.getsource(fib))
# fib.py
```

```
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
>>>
```

讨论¶

在详细讨论之前，有点要强调的是，Python的模块、包和导入机制是整个语言中最复杂的部分， 即使经验丰富的Python程序员也很少能精通它们。 我在这里推荐一些值的去读的文档和书籍，包括 importlib module 和 PEP 302. 文档内容在这里不会被重复提到，不过我在这里会讨论一些最重要的部分。

首先，如果你想创建一个新的模块对象，使用 `imp.new_module()` 函数：

```
>>> import imp
>>> m = imp.new_module('spam')
>>> m
<module 'spam'>
>>> m.__name__
'spam'
>>>
```

模块对象通常有一些期望属性，包括 `__file__` （运行模块加载语句的文件名） 和 `__package__` (包名)。

其次，模块会被解释器缓存起来。模块缓存可以在字典 `sys.modules` 中被找到。 因为有了这个缓存机制，通常可以将缓存和模块的创建通过一个步骤完成：

```
>>> import sys
>>> import imp
>>> m = sys.modules.setdefault('spam', imp.new_module('spam'))
>>> m
<module 'spam'>
>>>
```

如果给定模块已经存在那么就会直接获得已经被创建过的模块，例如：

```
>>> import math
>>> m = sys.modules.setdefault('math', imp.new_module('math'))
>>> m
<module 'math' from '/usr/local/lib/python3.3/lib-dynload/math.so'>
>>> m.sin(2)
0.9092974268256817
>>> m.cos(2)
-0.4161468365471424
>>>
```

由于创建模块很简单，很容易编写简单函数比如第一部分的 `load_module()` 函数。 这个方案的一个缺点是很难处理复杂情况比如包的导入。 为了处理一个包，你要重新实现普通import语句的底层逻辑（比如检查目录，查找__init__.py文件， 执行那些文件，设置路径等）。这个复杂性就是为什么最好直接扩展import语句而不是自定义函数的一个原因。

扩展import语句很简单，但是会有很多移动操作。 最高层上，导入操作被一个位于sys.meta_path列表中的“元路径”查找器处理。 如果你输出它的值，会看到下面这样：

```
>>> from pprint import pprint
>>> pprint(sys.meta_path)
[<class '_frozen_importlib.BuiltinImporter'>,
<class '_frozen_importlib.FrozenImporter'>,
<class '_frozen_importlib.PathFinder'>]
>>>
```

当执行一个语句比如 `import fib` 时，解释器会遍历sys.mata_path中的查找器对象， 调用它们的 `find_module()` 方法定位正确的模块加载器。 可以通过实验来看看：

```
>>> class Finder:
...     def find_module(self, fullname, path):
...         print('Looking for', fullname, path)
...         return None
...
>>> import sys
>>> sys.meta_path.insert(0, Finder()) # Insert as first entry
>>> import math
Looking for math None
>>> import types
Looking for types None
>>> import threading
Looking for threading None
Looking for time None
Looking for traceback None
Looking for linecache None
Looking for tokenize None
Looking for token None
>>>
```

注意看 `find_module()` 方法是怎样在每一个导入就被触发的。 这个方法中的path参数的作用是处理包。 多个包被导入，就是一个可在包的 `__path__` 属性中找到的路径列表。 要找到包的子组件就要检查这些路径。 比如注意对于 `xml.etree` 和 `xml.etree.ElementTree` 的路径配置：

```
>>> import xml.etree.ElementTree
Looking for xml None
Looking for xml.etree ['/usr/local/lib/python3.3/xml']
Looking for xml.etree.ElementTree ['/usr/local/lib/python3.3/xml/etree']
Looking for warnings None
Looking for contextlib None
Looking for xml.etree.ElementPath ['/usr/local/lib/python3.3/xml/etree']
Looking for _elementtree None
Looking for copy None
Looking for org None
Looking for pyexpat None
Looking for ElementC14N None
>>>
```

在 `sys.meta_path` 上查找器的位置很重要，将它从队头移到队尾，然后再试试导入看：

```
>>> del sys.meta_path[0]
>>> sys.meta_path.append(Finder())
>>> import urllib.request
>>> import datetime
```

现在你看不到任何输出了，因为导入被sys.meta_path中的其他实体处理。 这时候，你只有在导入不存在模块的时候才能看到它被触发：

```
>>> import fib
Looking for fib None
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import xml.superfast
Looking for xml.superfast ['/usr/local/lib/python3.3/xml']
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ImportError: No module named 'xml.superfast'
>>>
```

你之前安装过一个捕获未知模块的查找器，这个是 `UrlMetaFinder` 类的关键。 一个 `UrlMetaFinder` 实例被添加到 `sys.meta_path` 的末尾，作为最后一个查找器方案。 如果被请求的模块名不能定位，就会被这个查找器处理掉。 处理包的时候需要注意，在path参数中指定的值需要被检查，看它是否以查找器中注册的URL开头。 如果不是，该子模块必须归属于其他查找器并被忽略掉。

对于包的其他处理可在 `UrlPackageLoader` 类中被找到。 这个类不会导入包名，而是去加载对应的 `__init__.py` 文件。 它也会设置模块的 `__path__` 属性，这一步很重要， 因为在加载包的子模块时这个值会被传给后面的 `find_module()` 调用。 基于路径的导入钩子是这些思想的一个扩展，但是采用了另外的方法。 我们都知道，`sys.path`

是一个Python查找模块的目录列表，例如：

```
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
['',
'/usr/local/lib/python33.zip',
'/usr/local/lib/python3.3',
'/usr/local/lib/python3.3/plat-darwin',
'/usr/local/lib/python3.3/lib-dynload',
'/usr/local/lib/...3.3/site-packages']
>>>
```

在 `sys.path` 中的每一个实体都会被额外的绑定到一个查找器对象上。 你可以通过查看 `sys.path_importer_cache` 去看下这些查找器：

```
>>> pprint(sys.path_importer_cache)
{'.': FileFinder('.'),
'/usr/local/lib/python3.3': FileFinder('/usr/local/lib/python3.3'),
'/usr/local/lib/python3.3/': FileFinder('/usr/local/lib/python3.3/'),
'/usr/local/lib/python3.3/collections': FileFinder('...python3.3/collections'),
'/usr/local/lib/python3.3/encodings': FileFinder('...python3.3/encodings'),
'/usr/local/lib/python3.3/lib-dynload': FileFinder('...python3.3/lib-dynload'),
'/usr/local/lib/python3.3/plat-darwin': FileFinder('...python3.3/plat-darwin'),
'/usr/local/lib/python3.3/site-packages': FileFinder('...python3.3/site-packages'),
'/usr/local/lib/python33.zip': None}
>>>
```

`sys.path_importer_cache` 比 `sys.path` 会更大点， 因为它会为所有被加载代码的目录记录它们的查找器。 这包括包的子目录，这些通常在 `sys.path` 中是不存在的。

要执行 `import fib` ，会顺序检查 `sys.path` 中的目录。 对于每个目录，名称“fib”会被传给相应的 `sys.path_importer_cache` 中的查找器。 这个可以让你创建自己的查找器并在缓存中放入一个实体。试试这个：

```
>>> class Finder:
... def find_loader(self, name):
...     print('Looking for', name)
...     return (None, [])
...
>>> import sys
>>> # Add a "debug" entry to the importer cache
>>> sys.path_importer_cache['debug'] = Finder()
>>> # Add a "debug" directory to sys.path
>>> sys.path.insert(0, 'debug')
>>> import threading
Looking for threading
Looking for time
Looking for traceback
Looking for linecache
Looking for tokenize
Looking for token
>>>
```

在这里，你可以为名字“debug”创建一个新的缓存实体并将它设置成 `sys.path` 上的第一个。 在所有接下来的导入中，你会看到你的查找器被触发了。 不过，由于它返回 (None, [])，那么处理进程会继续处理下一个实体。

`sys.path_importer_cache` 的使用被一个存储在 `sys.path_hooks` 中的函数列表控制。 试试下面的例子，它会清除缓存并给 `sys.path_hooks` 添加一个新的路径检查函数

```
>>> sys.path_importer_cache.clear()
>>> def check_path(path):
...     print('Checking', path)
...     raise ImportError()
...
>>> sys.path_hooks.insert(0, check_path)
>>> import fib
Checked debug
Checking .
```

```
Checking /usr/local/lib/python33.zip
Checking /usr/local/lib/python3.3
Checking /usr/local/lib/python3.3/plat-darwin
Checking /usr/local/lib/python3.3/lib-dynload
Checking /Users/beazley/.local/lib/python3.3/site-packages
Checking /usr/local/lib/python3.3/site-packages
Looking for fib
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>>
```

正如你所见，`check_path()` 函数被每个 `sys.path` 中的实体调用。 不顾，由于抛出了 `ImportError` 异常， 啥都不会发生了（仅仅将检查转移到sys.path_hooks的下一个函数）。

知道了怎样sys.path是怎样被处理的，你就能构建一个自定义路径检查函数来查找文件名，不然URL。例如：

```
>>> def check_url(path):
...     if path.startswith('http://'):
...         return Finder()
...     else:
...         raise ImportError()
...
>>> sys.path.append('http://localhost:15000')
>>> sys.path_hooks[0] = check_url
>>> import fib
Looking for fib # Finder output!
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Notice installation of Finder in sys.path_importer_cache
>>> sys.path_importer_cache['http://localhost:15000']
<__main__.Finder object at 0x10064c850>
>>>
```

这就是本节最后部分的关键点。事实上，一个用来在sys.path中查找URL的自定义路径检查函数已经构建完毕。 当它们被碰到的时候，一个新的 `UrlPathFinder` 实例被创建并被放入 `sys.path_importer_cache`. 之后，所有需要检查 `sys.path` 的导入语句都会使用你的自定义查找器。

基于路径导入的包处理稍微有点复杂，并且跟 `find_loader()` 方法返回值有关。 对于简单模块，`find_loader()` 返回一个元组(loader, None)， 其中的loader是一个用于导入模块的加载器实例。

对于一个普通的包，`find_loader()` 返回一个元组(loader, path)， 其中的loader是一个用于导入包（并执行__init__.py）的加载器实例， path是一个会初始化包的 `__path__` 属性的目录列表。 例如，如果基础URL是 http://localhost:15000 并且一个用户执行 `import grok`, 那么 `find_loader()` 返回的path就会是 ['http://localhost:15000/grok']

`find_loader()` 还要能处理一个命名空间包。 一个命名空间包中有一个合法的包目录名，但是不存在__init__.py文件。这样的话，`find_loader()` 必须返回一个元组(None, path)， path是一个目录列表，由它来构建包的定义有__init__.py文件的__path__属性。 对于这种情况，导入机制会继续前行去检查sys.path中的目录。 如果找到了命名空间包，所有的结果路径被加到一起来构建最终的命名空间包。 关于命名空间包的更多信息请参考10.5小节。

所有的包都包含了一个内部路径设置，可以在__path__属性中看到，例如：

```
>>> import xml.etree.ElementTree
>>> xml.__path__
['/usr/local/lib/python3.3/xml']
>>> xml.etree.__path__
['/usr/local/lib/python3.3/xml/etree']
>>>
```

之前提到，__path__的设置是通过 `find_loader()` 方法返回值控制的。 不过，__path__接下来也被sys.path_hooks中的函数处理。 因此，但包的子组件被加载后，位于__path__中的实体会被 `handle_url()` 函数检查。 这会导致新的 `UrlPathFinder` 实例被创建并且被加入到 `sys.path_importer_cache` 中。

还有个难点就是 `handle_url()` 函数以及它跟内部使用的 `_get_links()` 函数之间的交互。 如果你的查找器实现需要使

用到其他模块（比如urllib.request）， 有可能这些模块会在查找器操作期间进行更多的导入。 它可以导致 `handle_url()` 和其他查找器部分陷入一种递归循环状态。 为了解释这种可能性，实现中有一个被创建的查找器缓存（每一个URL一个）。 它可以避免创建重复查找器的问题。 另外，下面的代码片段可以确保查找器不会在初始化链接集合的时候响应任何导入请求：

```
# Check link cache
if self._links is None:
    self._links = [] # See discussion
    self._links = _get_links(self._baseurl)
```

最后，查找器的 `invalidate_caches()` 方法是一个工具方法，用来清理内部缓存。 这个方法再用户调用 `importlib.invalidate_caches()` 的时候被触发。 如果你想让URL导入者重新读取链接列表的话可以使用它。

对比下两种方案（修改sys.meta_path或使用一个路径钩子）。 使用sys.meta_path的导入者可以按照自己的需要自由处理模块。 例如，它们可以从数据库中导入或以不同于一般模块/包处理方式导入。 这种自由同样意味着导入者需要自己进行内部的一些管理。 另外，基于路径的钩子只是适用于对sys.path的处理。 通过这种扩展加载的模块跟普通方式加载的特性是一样的。

如果到现在为止你还是不是很明白，那么可以通过增加一些日志打印来测试下本节。像下面这样：

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> import urlimport
>>> urlimport.install_path_hook()
DEBUG:urlimport:Installing handle_url
>>> import fib
DEBUG:urlimport:Handle path? /usr/local/lib/python33.zip. [No]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
DEBUG:urlimport:Handle path? http://localhost:15000. [Yes]
DEBUG:urlimport:Getting links from http://localhost:15000
DEBUG:urlimport:links: {'spam.py', 'fib.py', 'grok'}
DEBUG:urlimport:find_loader: 'fib'
DEBUG:urlimport:find_loader: module 'fib' found
DEBUG:urlimport:loader: reading 'http://localhost:15000/fib.py'
DEBUG:urlimport:loader: 'http://localhost:15000/fib.py' loaded
I'm fib
>>>
```

最后，建议你花点时间看看 PEP 302 以及importlib的文档。

10.12 导入模块的同时修改模块¶

问题¶

你想给某个已存在模块中的函数添加装饰器。 不过，前提是这个模块已经被导入并且被使用过。

解决方案¶

这里问题的本质就是你想在模块被加载时执行某个动作。 可能是你想在一个模块被加载时触发某个回调函数来通知你。

这个问题可以使用10.11小节中同样的导入钩子机制来实现。下面是一个可能的方案：

```
# postimport.py
import importlib
import sys
from collections import defaultdict

_post_import_hooks = defaultdict(list)

class PostImportFinder:
    def __init__(self):
        self._skip = set()

    def find_module(self, fullname, path=None):
        if fullname in self._skip:
            return None
        self._skip.add(fullname)
        return PostImportLoader(self)

class PostImportLoader:
    def __init__(self, finder):
        self._finder = finder

    def load_module(self, fullname):
        importlib.import_module(fullname)
        module = sys.modules[fullname]
        for func in _post_import_hooks[fullname]:
            func(module)
        self._finder._skip.remove(fullname)
        return module

def when_imported(fullname):
    def decorate(func):
        if fullname in sys.modules:
            func(sys.modules[fullname])
        else:
            _post_import_hooks[fullname].append(func)
        return func
    return decorate

sys.meta_path.insert(0, PostImportFinder())
```

这样，你就可以使用 `when_imported()` 装饰器了，例如：

```
>>> from postimport import when_imported
>>> @when_imported('threading')
... def warn_threads(mod):
...     print('Threads? Are you crazy?')
...
>>>
>>> import threading
Threads? Are you crazy?
>>>
```

作为一个更实际的例子，你可能想在已存在的定义上面添加装饰器，如下所示：

```
from functools import wraps
from postimport import when_imported

def logged(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__, args, kwargs)
        return func(*args, **kwargs)
    return wrapper

# Example
@when_imported('math')
def add_logging(mod):
    mod.cos = logged(mod.cos)
    mod.sin = logged(mod.sin)
```

讨论¶

本节技术依赖于10.11小节中讲述过的导入钩子，并稍作修改。

`@when_imported` 装饰器的作用是注册在导入时被激活的处理器函数。 该装饰器检查sys.modules来查看模块是否真的已经被加载了。 如果是的话，该处理器被立即调用。不然，处理器被添加到 `_post_import_hooks` 字典中的一个列表中去。 `_post_import_hooks` 的作用就是收集所有的为每个模块注册的处理器对象。 一个模块可以注册多个处理器。

要让模块导入后触发添加的动作，`PostImportFinder` 类被设置为sys.meta_path第一个元素。 它会捕获所有模块导入操作。

本节中的 `PostImportFinder` 的作用并不是加载模块，而是自带导入完成后触发相应的动作。 实际的导入被委派给位于sys.meta_path中的其他查找器。 `PostImportLoader` 类中的 `imp.import_module()` 函数被递归的调用。 为了避免陷入无线循环，`PostImportFinder` 保持了一个所有被加载过的模块集合。 如果一个模块名存在就会直接被忽略掉。

当一个模块被 `imp.import_module()` 加载后， 所有在_post_import_hooks被注册的处理器被调用，使用新加载模块作为一个参数。

有一点需要注意的是本机不适用于那些通过 `imp.reload()` 被显式加载的模块。 也就是说，如果你加载一个之前已被加载过的模块，那么导入处理器将不会再被触发。 另外，要是你从sys.modules中删除模块然后再重新导入，处理器又会再一次触发。

更多关于导入后钩子信息请参考 PEP 369.

10.13 安装私有的包¶

问题¶

你想要安装一个第三方包，但是没有权限将它安装到系统Python库中去。 或者，你可能想要安装一个供自己使用的包，而不是系统上面所有用户。

解决方案¶

Python有一个用户安装目录，通常类似”~/.local/lib/python3.3/site-packages”。 要强制在这个目录中安装包，可使用安装选项“–user”。例如：

```
python3 setup.py install --user
```

或者

```
pip install --user packagename
```

在sys.path中用户的“site-packages”目录位于系统的“site-packages”目录之前。 因此，你安装在里面的包就比系统已安装的包优先级高 （尽管并不总是这样，要取决于第三方包管理器，比如distribute或pip）。

讨论¶

通常包会被安装到系统的site-packages目录中去，路径类似“/usr/local/lib/python3.3/site-packages”。 不过，这样做需要有管理员权限并且使用sudo命令。 就算你有这样的权限去执行命令，使用sudo去安装一个新的，可能没有被验证过的包有时候也不安全。

安装包到用户目录中通常是一个有效的方案，它允许你创建一个自定义安装。

另外，你还可以创建一个虚拟环境，这个我们在下一节会讲到。

10.14 创建新的Python环境¶

问题¶

你想创建一个新的Python环境，用来安装模块和包。 不过，你不想安装一个新的Python克隆，也不想对系统Python环境产生影响。

解决方案¶

你可以使用 `pyvenv` 命令创建一个新的“虚拟”环境。 这个命令被安装在Python解释器同一目录，或Windows上面的Scripts目录中。下面是一个例子：

```
bash % pyvenv Spam
bash %
```

传给 `pyvenv` 命令的名字是将要被创建的目录名。当被创建后，Span目录像下面这样：

```
bash % cd Spam
bash % ls
bin include lib pyvenv.cfg
bash %
```

在bin目录中，你会找到一个可以使用的Python解释器：

```
bash % Spam/bin/python3
Python 3.3.0 (default, Oct 6 2012, 15:45:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
['',
'/usr/local/lib/python33.zip',
'/usr/local/lib/python3.3',
'/usr/local/lib/python3.3/plat-darwin',
'/usr/local/lib/python3.3/lib-dynload',
'/Users/beazley/Spam/lib/python3.3/site-packages']
>>>
```

这个解释器的特点就是他的site-packages目录被设置为新创建的环境。 如果你要安装第三方包，它们会被安装在那里，而不是通常系统的site-packages目录。

讨论¶

创建虚拟环境通常是为了安装和管理第三方包。 正如你在例子中看到的那样，`sys.path` 变量包含来自于系统Python的目录， 而 site-packages目录已经被重定位到一个新的目录。

有了一个新的虚拟环境，下一步就是安装一个包管理器，比如distribute或pip。 但安装这样的工具和包的时候，你需要确保你使用的是虚拟环境的解释器。 它会将包安装到新创建的site-packages目录中去。

尽管一个虚拟环境看上去是Python安装的一个复制， 不过它实际上只包含了少量几个文件和一些符号链接。 所有标准库函文件和可执行解释器都来自原来的Python安装。 因此，创建这样的环境是很容易的，并且几乎不会消耗机器资源。

默认情况下，虚拟环境是空的，不包含任何额外的第三方库。如果你想将一个已经安装的包作为虚拟环境的一部分，可以使用“–system-site-packages”选项来创建虚拟环境，例如：

```
bash % pyvenv --system-site-packages Spam
bash %
```

跟多关于 `pyvenv` 和虚拟环境的信息可以参考 PEP 405.

10.15 分发包¶

问题¶

你已经编写了一个有用的库，想将它分享给其他人。

解决方案¶

如果你想分发你的代码，第一件事就是给它一个唯一的名字，并且清理它的目录结构。 例如，一个典型的函数库包会类似下面这样：

```
projectname/
    README.txt
    Doc/
        documentation.txt
    projectname/
        __init__.py
        foo.py
        bar.py
        utils/
            __init__.py
            spam.py
            grok.py
    examples/
        helloworld.py
        ...
```

要让你的包可以发布出去，首先你要编写一个 `setup.py`，类似下面这样：

```
# setup.py
from distutils.core import setup

setup(name='projectname',
    version='1.0',
    author='Your Name',
    author_email='you@youraddress.com',
    url='http://www.you.com/projectname',
    packages=['projectname', 'projectname.utils'],
)
```

下一步，就是创建一个 `MANIFEST.in` 文件，列出所有在你的包中需要包含进来的非源码文件：

```
# MANIFEST.in
include *.txt
recursive-include examples *
recursive-include Doc *
```

确保 `setup.py` 和 `MANIFEST.in` 文件放在你的包的最顶级目录中。 一旦你已经做了这些，你就可以像下面这样执行命令来创建一个源码分发包了：

```
% bash python3 setup.py sdist
```

它会创建一个文件比如"projectname-1.0.zip" 或 "projectname-1.0.tar.gz", 具体依赖于你的系统平台。如果一切正常， 这个文件就可以发送给别人使用或者上传至 Python Package Index.

讨论¶

对于纯Python代码，编写一个普通的 `setup.py` 文件通常很简单。 一个可能的问题是你必须手动列出所有构成包源码的子目录。 一个常见错误就是仅仅只列出一个包的最顶级目录，忘记了包含包的子组件。 这也是为什么在 `setup.py` 中对于包的说明包含了列表 `packages=['projectname', 'projectname.utils']`

大部分Python程序员都知道，有很多第三方包管理器供选择，包括setuptools、distribute等等。 有些是为了替代标准库中的distutils。注意如果你依赖这些包， 用户可能不能安装你的软件，除非他们已经事先安装过所需要的包管理器。 正因

如此，你更应该时刻记住越简单越好的道理。 最好让你的代码使用标准的Python 3安装。 如果其他包也需要的话，可以通过一个可选项来支持。

对于涉及到C扩展的代码打包与分发就更复杂点了。 第15章对关于C扩展的这方面知识有一些详细讲解，特别是在15.2小节中。

10.2 控制模块被全部导入的内容¶

问题¶

当使用’from module import *’ 语句时，希望对从模块或包导出的符号进行精确控制。

解决方案¶

在你的模块中定义一个变量 __all__ 来明确地列出需要导出的内容。

举个例子:

```
# somemodule.py
def spam():
    pass

def grok():
    pass

blah = 42
# Only export 'spam' and 'grok'
__all__ = ['spam', 'grok']
```

讨论¶

尽管强烈反对使用 ‘from module import *’, 但是在定义了大量变量名的模块中频繁使用。 如果你不做任何事, 这样的导入将会导入所有不以下划线开头的。 另一方面,如果定义了 __all__ , 那么只有被列举出的东西会被导出。

如果你将 __all__ 定义成一个空列表, 没有东西将被导入。 如果 __all__ 包含未定义的名字, 在导入时引起AttributeError。

10.3 使用相对路径名导入包中子模块¶

问题¶

将代码组织成包,想用import语句从另一个包名没有硬编码过的包中导入子模块。

解决方案¶

使用包的相对导入，使一个模块导入同一个包的另一个模块 举个例子，假设在你的文件系统上有mypackage包，组织如下：

```
mypackage/
    __init__.py
    A/
        __init__.py
        spam.py
        grok.py
    B/
        __init__.py
        bar.py
```

如果模块mypackage.A.spam要导入同目录下的模块grok，它应该包括的import语句如下：

```
# mypackage/A/spam.py
from . import grok
```

如果模块mypackage.A.spam要导入不同目录下的模块B.bar，它应该使用的import语句如下：

```
# mypackage/A/spam.py
from ..B import bar
```

两个import语句都没包含顶层包名，而是使用了spam.py的相对路径。

讨论¶

在包内，既可以使用相对路径也可以使用绝对路径来导入。 举个例子：

```
# mypackage/A/spam.py
from mypackage.A import grok # OK
from . import grok # OK
import grok # Error (not found)
```

像mypackage.A这样使用绝对路径名的不利之处是这将顶层包名硬编码到你的源码中。如果你想重新组织它，你的代码将更脆，很难工作。 举个例子，如果你改变了包名，你就必须检查所有文件来修正源码。 同样，硬编码的名称会使移动代码变得困难。举个例子，也许有人想安装两个不同版本的软件包，只通过名称区分它们。 如果使用相对导入，那一切都ok，然而使用绝对路径名很可能会出问题。

import语句的 . 和 .. 看起来很滑稽, 但它指定目录名.为当前目录，..B为目录../B。这种语法只适用于import。 举个例子：

```
from . import grok # OK
import .grok # ERROR
```

尽管使用相对导入看起来像是浏览文件系统，但是不能到定义包的目录之外。也就是说，使用点的这种模式从不是包的目录中导入将会引发错误。

最后，相对导入只适用于在合适的包中的模块。尤其是在顶层的脚本的简单模块中，它们将不起作用。如果包的部分被作为脚本直接执行，那它们将不起作用 例如：

```
% python3 mypackage/A/spam.py # Relative imports fail
```

另一方面，如果你使用Python的-m选项来执行先前的脚本，相对导入将会正确运行。 例如：

```
% python3 -m mypackage.A.spam # Relative imports work
```

更多的包的相对导入的背景知识,请看 PEP 328 .

10.4 将模块分割成多个文件¶

问题¶

你想将一个模块分割成多个文件。但是你不想将分离的文件统一成一个逻辑模块时使已有的代码遭到破坏。

解决方案¶

程序模块可以通过变成包来分割成多个独立的文件。考虑下下面简单的模块：

```
# mymodule.py
class A:
    def spam(self):
        print('A.spam')

class B(A):
    def bar(self):
        print('B.bar')
```

假设你想mymodule.py分为两个文件，每个定义的一个类。要做到这一点，首先用mymodule目录来替换文件mymodule.py。 这这个目录下，创建以下文件：

```
mymodule/
    __init__.py
    a.py
    b.py
```

在a.py文件中插入以下代码：

```
# a.py
class A:
    def spam(self):
        print('A.spam')
```

在b.py文件中插入以下代码：

```
# b.py
from .a import A
class B(A):
    def bar(self):
        print('B.bar')
```

最后，在 __init__.py 中，将2个文件粘合在一起：

```
# __init__.py
from .a import A
from .b import B
```

如果按照这些步骤，所产生的包MyModule将作为一个单一的逻辑模块：

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>> b = mymodule.B()
>>> b.bar()
B.bar
>>>
```

讨论¶

在这个章节中的主要问题是一个设计问题，不管你是否希望用户使用很多小模块或只是一个模块。举个例子，在一个大型的代码库中，你可以将这一切都分割成独立的文件，让用户使用大量的import语句，就像这样：

```
from mymodule.a import A
from mymodule.b import B
...
```

这样能工作，但这让用户承受更多的负担，用户要知道不同的部分位于何处。通常情况下，将这些统一起来，使用一条import将更加容易，就像这样：

```
from mymodule import A, B
```

对后者而言，让mymodule成为一个大的源文件是最常见的。但是，这一章节展示了如何合并多个文件合并成一个单一的逻辑命名空间。 这样做的关键是创建一个包目录，使用 __init__.py 文件来将每部分粘合在一起。

当一个模块被分割，你需要特别注意交叉引用的文件名。举个例子，在这一章节中，B类需要访问A类作为基类。用包的相对导入 from .a import A 来获取。

整个章节都使用包的相对导入来避免将顶层模块名硬编码到源代码中。这使得重命名模块或者将它移动到别的位置更容易。（见10.3小节）

作为这一章节的延伸，将介绍延迟导入。如图所示，__init__.py文件一次导入所有必需的组件的。但是对于一个很大的模块，可能你只想组件在需要时被加载。 要做到这一点，__init__.py有细微的变化：

```
# __init__.py
def A():
    from .a import A
    return A()

def B():
    from .b import B
    return B()
```

在这个版本中，类A和类B被替换为在第一次访问时加载所需的类的函数。对于用户，这看起来不会有太大的不同。例如：

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>>
```

延迟加载的主要缺点是继承和类型检查可能会中断。你可能会稍微改变你的代码，例如:

```
if isinstance(x, mymodule.A): # Error
...

if isinstance(x, mymodule.a.A): # Ok
...
```

延迟加载的真实例子, 见标准库 multiprocessing/__init__.py 的源码.

10.5 利用命名空间导入目录分散的代码¶

问题¶

你可能有大量的代码，由不同的人来分散地维护。每个部分被组织为文件目录，如一个包。然而，你希望能用共同的包前缀将所有组件连接起来，不是将每一个部分作为独立的包来安装。

解决方案¶

从本质上讲，你要定义一个顶级Python包，作为一个大集合分开维护子包的命名空间。这个问题经常出现在大的应用框架中，框架开发者希望鼓励用户发布插件或附加包。

在统一不同的目录里统一相同的命名空间，但是要删去用来将组件联合起来的__init__.py文件。假设你有Python代码的两个不同的目录如下：

```
foo-package/
    spam/
        blah.py

bar-package/
    spam/
        grok.py
```

在这2个目录里，都有着共同的命名空间spam。在任何一个目录里都没有__init__.py文件。

让我们看看，如果将foo-package和bar-package都加到python模块路径并尝试导入会发生什么

```
>>> import sys
>>> sys.path.extend(['foo-package', 'bar-package'])
>>> import spam.blah
>>> import spam.grok
>>>
```

两个不同的包目录被合并到一起，你可以导入spam.blah和spam.grok，并且它们能够工作。

讨论¶

在这里工作的机制被称为“包命名空间”的一个特征。从本质上讲，包命名空间是一种特殊的封装设计，为合并不同的目录的代码到一个共同的命名空间。对于大的框架，这可能是有用的，因为它允许一个框架的部分被单独地安装下载。它也使人们能够轻松地为这样的框架编写第三方附加组件和其他扩展。

包命名空间的关键是确保顶级目录中没有__init__.py文件来作为共同的命名空间。缺失__init__.py文件使得在导入包的时候会发生有趣的事情：这并没有产生错误，解释器创建了一个由所有包含匹配包名的目录组成的列表。特殊的包命名空间模块被创建，只读的目录列表副本被存储在其__path__变量中。 举个例子：

```
>>> import spam
>>> spam.__path__
_NamespacePath(['foo-package/spam', 'bar-package/spam'])
>>>
```

在定位包的子组件时，目录__path__将被用到(例如, 当导入spam.grok或者spam.blah的时候).

包命名空间的一个重要特点是任何人都可以用自己的代码来扩展命名空间。举个例子，假设你自己的代码目录像这样：

```
my-package/
    spam/
        custom.py
```

如果你将你的代码目录和其他包一起添加到sys.path，这将无缝地合并到别的spam包目录中：

```
>>> import spam.custom
>>> import spam.grok
>>> import spam.blah
>>>
```

一个包是否被作为一个包命名空间的主要方法是检查其__file__属性。如果没有，那包是个命名空间。这也可以由其字符表现形式中的“namespace”这个词体现出来。

```
>>> spam.__file__
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute '__file__'
>>> spam
<module 'spam' (namespace)>
>>>
```

更多的包命名空间信息可以查看 PEP 420.

10.6 重新加载模块¶

问题¶

你想重新加载已经加载的模块，因为你对其源码进行了修改。

解决方案¶

使用imp.reload()来重新加载先前加载的模块。举个例子：

```
>>> import spam
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>>
```

讨论¶

重新加载模块在开发和调试过程中常常很有用。但在生产环境中的代码使用会不安全，因为它并不总是像您期望的那样工作。

reload()擦除了模块底层字典的内容，并通过重新执行模块的源代码来刷新它。模块对象本身的身份保持不变。因此，该操作在程序中所有已经被导入了的地方更新了模块。

尽管如此，reload()没有更新像”from module import name”这样使用import语句导入的定义。举个例子：

```
# spam.py
def bar():
    print('bar')

def grok():
    print('grok')
```

现在启动交互式会话：

```
>>> import spam
>>> from spam import grok
>>> spam.bar()
bar
>>> grok()
grok
>>>
```

不退出Python修改spam.py的源码，将grok()函数改成这样：

```
def grok():
    print('New grok')
```

现在回到交互式会话，重新加载模块，尝试下这个实验：

```
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>> spam.bar()
bar
>>> grok() # Notice old output
grok
>>> spam.grok() # Notice new output
New grok
>>>
```

在这个例子中，你看到有2个版本的grok()函数被加载。通常来说，这不是你想要的，而是令人头疼的事。

因此，在生产环境中可能需要避免重新加载模块。在交互环境下调试，解释程序并试图弄懂它。

10.7 运行目录或压缩文件¶

问题¶

您有一个已成长为包含多个文件的应用，它已远不再是一个简单的脚本，你想向用户提供一些简单的方法运行这个程序。

解决方案¶

如果你的应用程序已经有多个文件，你可以把你的应用程序放进它自己的目录并添加一个__main__.py文件。 举个例子，你可以像这样创建目录：

```
myapplication/
    spam.py
    bar.py
    grok.py
    __main__.py
```

如果__main__.py存在，你可以简单地在顶级目录运行Python解释器：

```
bash % python3 myapplication
```

解释器将执行__main__.py文件作为主程序。

如果你将你的代码打包成zip文件，这种技术同样也适用，举个例子：

```
bash % ls
spam.py bar.py grok.py __main__.py
bash % zip -r myapp.zip *.py
bash % python3 myapp.zip
... output from __main__.py ...
```

讨论¶

创建一个目录或zip文件并添加__main__.py文件来将一个更大的Python应用打包是可行的。这和作为标准库被安装到Python库的代码包是有一点区别的。相反，这只是让别人执行的代码包。

由于目录和zip文件与正常文件有一点不同，你可能还需要增加一个shell脚本，使执行更加容易。例如，如果代码文件名为myapp.zip，你可以创建这样一个顶级脚本：

```
#!/usr/bin/env python3 /usr/local/bin/myapp.zip
```

10.8 读取位于包中的数据文件¶

问题¶

你的包中包含代码需要去读取的数据文件。你需要尽可能地用最便捷的方式来做这件事。

解决方案¶

假设你的包中的文件组织成如下：

```
mypackage/
    __init__.py
    somedata.dat
    spam.py
```

现在假设spam.py文件需要读取somedata.dat文件中的内容。你可以用以下代码来完成：

```
# spam.py
import pkgutil
data = pkgutil.get_data(__package__, 'somedata.dat')
```

由此产生的变量是包含该文件的原始内容的字节字符串。

讨论¶

要读取数据文件，你可能会倾向于编写使用内置的I/O功能的代码，如open()。但是这种方法也有一些问题。

首先，一个包对解释器的当前工作目录几乎没有控制权。因此，编程时任何I/O操作都必须使用绝对文件名。由于每个模块包含有完整路径的__file__变量，这弄清楚它的路径不是不可能，但它很凌乱。

第二，包通常安装作为.zip或.egg文件，这些文件并不像在文件系统上的一个普通目录里那样被保存。因此，你试图用open()对一个包含数据文件的归档文件进行操作，它根本不会工作。

pkgutil.get_data()函数是一个读取数据文件的高级工具，不用管包是如何安装以及安装在哪。它只是工作并将文件内容以字节字符串返回给你

get_data()的第一个参数是包含包名的字符串。你可以直接使用包名，也可以使用特殊的变量，比如__package__。第二个参数是包内文件的相对名称。如果有必要，可以使用标准的Unix命名规范到不同的目录，只要最后的目录仍然位于包中。

10.9 将文件夹加入到sys.path¶

问题¶

你无法导入你的Python代码因为它所在的目录不在sys.path里。你想将添加新目录到Python路径，但是不想硬链接到你的代码。

解决方案¶

有两种常用的方式将新目录添加到sys.path。第一种，你可以使用PYTHONPATH环境变量来添加。例如：

```
bash % env PYTHONPATH=/some/dir:/other/dir python3
Python 3.3.0 (default, Oct 4 2012, 10:17:33)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/some/dir', '/other/dir', ...]
>>>
```

在自定义应用程序中，这样的环境变量可在程序启动时设置或通过shell脚本。

第二种方法是创建一个.pth文件，将目录列举出来，像这样：

```
# myapplication.pth
/some/dir
/other/dir
```

这个.pth文件需要放在某个Python的site-packages目录，通常位于/usr/local/lib/python3.3/site-packages 或者 ~/.local/lib/python3.3/sitepackages。当解释器启动时，.pth文件里列举出来的存在于文件系统的目录将被添加到sys.path。安装一个.pth文件可能需要管理员权限，如果它被添加到系统级的Python解释器。

讨论¶

比起费力地找文件，你可能会倾向于写一个代码手动调节sys.path的值。例如:

```
import sys
sys.path.insert(0, '/some/dir')
sys.path.insert(0, '/other/dir')
```

虽然这能“工作”，但是在实践中极为脆弱，应尽量避免使用。这种方法的问题是，它将目录名硬编码到了你的源代码。如果你的代码被移到一个新的位置，这会导致维护问题。更好的做法是在不修改源代码的情况下，将path配置到其他地方。如果您使用模块级的变量来精心构造一个适当的绝对路径，有时你可以解决硬编码目录的问题，比如__file__。举个例子：

```
import sys
from os.path import abspath, join, dirname
sys.path.insert(0, join(abspath(dirname(__file__)), 'src'))
```

这将src目录添加到path里，和执行插入步骤的代码在同一个目录里。

site-packages目录是第三方包和模块安装的目录。如果你手动安装你的代码，它将被安装到site-packages目录。虽然用于配置path的.pth文件必须放置在site-packages里，但它配置的路径可以是系统上任何你希望的目录。因此，你可以把你的代码放在一系列不同的目录，只要那些目录包含在.pth文件里。

第十章：模块与包¶

模块与包是任何大型程序的核心，就连Python安装程序本身也是一个包。本章重点涉及有关模块和包的常用编程技术，例如如何组织包、把大型模块分割成多个文件、创建命名空间包。同时，也给出了让你自定义导入语句的秘籍。

Contents: