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：

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._baseurl + '/' + 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
```

```
rm lib
>>>
```

最后，建议你花点时间看看 PEP 302 以及importlib的文档。