5. 导入系统

一个 module 内的 Python 代码通过 importing 操作就能够访问另一个模块内的代码。 `import` 语句是唤起导入机制的最常用方式，但不是唯一的方式。 `importlib.import_module()` 以及内置的 `__import__()` 等函数也可以被用来唤起导入机制。

`import` 语句结合了两个操作；它先搜索指定名称的模块，然后将搜索结果绑定到当前作用域中的名称。 `import` 语句的搜索操作被定义为对 `__import__()` 函数的调用并带有适当的参数。 `__import__()` 的返回值会被用于执行 `import` 语句的名称绑定操作。 请参阅 `import` 语句了解名称绑定操作的更多细节。

对 `__import__()` 的直接调用将仅执行模块搜索以及在找到时的模块创建操作。 不过也可能产生某些副作用，例如导入父包和更新各种缓存 (包括 `sys.modules`)，只有 `import` 语句会执行名称绑定操作。

当 `import` 语句被执行时，标准的内置 `__import__()` 函数会被调用。 其他唤起导入系统的机制 (例如 `importlib.import_module()`) 可能会选择绕过 `__import__()` 并使用它们自己的解决方案来实现导入机制。

当一个模块首次被导入时，Python 会搜索该模块，如果找到就创建一个 module 对象 [1] 并初始化它。 如果指定名称的模块未找到，则会引发 `ModuleNotFoundError`。 当唤起导入机制时，Python 会实现多种策略来搜索指定名称的模块。 这些策略可以通过使用使用下文所描述的多种钩子来加以修改和扩展。

> *在 3.3 版本发生变更:* 导入系统已被更新以完全实现 **PEP 302** 中的第二阶段要求。 不会再有任何隐式的导入机制 —— 整个导入系统都通过 `sys.meta_path` 暴露出来。 此外，对原生命名空间包的支持也已被实现 (参见 **PEP 420**)。

5.1. `importlib`

`importlib` 模块提供了一个丰富的 API 用来与导入系统进行交互。 例如 `importlib.import_module()` 提供了相比内置的 `__import__()` 更推荐、更简单的 API 用来唤起导入机制。 更多细节请参看 `importlib` 库文档。

5.2. 包

Python 只有一种模块对象类型，所有模块都属于该类型，无论模块是用 Python、C 还是别的语言实现。 为了帮助组织模块并提供名称层次结构，Python 还引入了 包 的概念。

你可以把包看成是文件系统中的目录，并把模块看成是目录中的文件，但请不要对这个类比做过于字面的理解，因为包和模块不是必须来自于文件系统。 为了方便理解本文档，我们将继续使用这种

目录和文件的类比。 与文件系统一样，包通过层次结构进行组织，在包之内除了一般的模块，还可以有子包。

要注意的一个重点概念是所有包都是模块，但并非所有模块都是包。 或者换句话说，包只是一种特殊的模块。 特别地，任何具有 `__path__` 属性的模块都会被当作是包。

所有模块都有自己的名字。 子包名与其父包名会以点号分隔，与 Python 的标准属性访问语法一致。 因此你可能会有一个名为 `email` 的包，这个包中又有一个名为 `email.mime` 的子包以及该子包中的名为 `email.mime.text` 的子包。

5.2.1. 常规包

Python 定义了两种类型的包，常规包 和 命名空间包。 常规包是传统的包类型，它们在 Python 3.2 及之前就已存在。 常规包通常以一个包含 `__init__.py` 文件的目录形式实现。 当一个常规包被导入时，这个 `__init__.py` 文件会隐式地被执行，它所定义的对象会被绑定到该包命名空间中的名称。 `__init__.py` 文件可以包含与任何其他模块中所包含的 Python 代码相似的代码，Python 将在模块被导入时为其添加额外的属性。

例如，以下文件系统布局定义了一个最高层级的 `parent` 包和三个子包:

```
parent/
    __init__.py
    one/
        __init__.py
    two/
        __init__.py
    three/
        __init__.py
```

导入 `parent.one` 将隐式地执行 `parent/__init__.py` 和 `parent/one/__init__.py`。 后续导入 `parent.two` 或 `parent.three` 则将分别执行 `parent/two/__init__.py` 和 `parent/three/__init__.py`。

5.2.2. 命名空间包

命名空间包是由多个 部分 构成的，每个部分为父包增加一个子包。 各个部分可能处于文件系统的不同位置。 部分也可能处于 zip 文件中、网络上，或者 Python 在导入期间可以搜索的其他地方。 命名空间包并不一定会直接对应到文件系统中的对象；它们有可能是无实体表示的虚拟模块。

命名空间包的 `__path__` 属性不使用普通的列表。 而是使用定制的可迭代类型，如果其父包的路径 (或者最高层级包的 `sys.path`) 发生改变，这种对象会在该包内的下一次导入尝试时自动执行新的对包部分的搜索。

命名空间包没有 `parent/__init__.py` 文件。 实际上，在导入搜索期间可能找到多个 `parent` 目录，每个都由不同的部分所提供。 因此 `parent/one` 的物理位置不一定与 `parent/two` 相邻。 在这种情况下，Python 将为顶级的 `parent` 包创建一个命名空间包，无论是它本身还是它的某个子包被导入。

另请参阅 **PEP 420** 了解对命名空间包的规格描述。

5.3. 搜索

为了开始搜索，Python 需要被导入模块（或者包，对于当前讨论来说两者没有差别）的完整 限定名称。 此名称可以来自 `import` 语句所带的各种参数，或者来自传给 `importlib.import_module()` 或 `__import__()` 函数的形参。

此名称会在导入搜索的各个阶段被使用，它也可以是指向一个子模块的带点号路径，例如 `foo.bar.baz`。 在这种情况下，Python 会先尝试导入 `foo`，然后是 `foo.bar`，最后是 `foo.bar.baz`。 如果这些导入中的任何一个失败，都会引发 `ModuleNotFoundError`。

5.3.1. 模块缓存

在导入搜索期间首先会被检查的地方是 `sys.modules`。 这个映射起到缓存之前导入的所有模块的作用（包括其中间路径）。 因此如果之前导入过 `foo.bar.baz`，则 `sys.modules` 将包含 `foo`, `foo.bar` 和 `foo.bar.baz` 条目。 每个键的值就是相应的模块对象。

在导入期间，会在 `sys.modules` 查找模块名称，如存在则其关联的值就是需要导入的模块，导入过程完成。 然而，如果值为 `None`，则会引发 `ModuleNotFoundError`。 如果找不到指定模块名称，Python 将继续搜索该模块。

`sys.modules` 是可写的。删除键可能不会破坏关联的模块（因为其他模块可能会保留对它的引用），但它会使命名模块的缓存条目无效，导致 Python 在下次导入时重新搜索命名模块。键也可以赋值为 `None`，强制下一次导入模块导致 `ModuleNotFoundError`。

但是要小心，因为如果你还保有对某个模块对象的引用，同时停用其在 `sys.modules` 中的缓存条目，然后又再次导入该名称的模块，则前后两个模块对象将 *不是* 同一个。 相反地，`importlib.reload()` 将重用 *同一个* 模块对象，并简单地通过重新运行模块的代码来重新初始化模块内容。

5.3.2. 查找器和加载器

如果指定名称的模块在 `sys.modules` 找不到，则将唤起 Python 的导入协议以查找和加载该模块。此协议由两个概念性模块构成，即 查找器 和 加载器。 查找器的任务是确定是否能使用其所知的策略找到该名称的模块。 同时实现这两种接口的对象称为 导入器 —— 它们在确定能加载所需的模块时会返回其自身。

Python 包含了多个默认查找器和导入器。 第一个知道如何定位内置模块，第二个知道如何定位冻结模块。 第三个默认查找器会在 import path 中搜索模块。 import path 是一个由文件系统路径或 zip 文件组成的位置列表。 它还可以扩展为搜索任意可定位资源，例如由 URL 指定的资源。

导入机制是可扩展的，因此可以加入新的查找器以扩展模块搜索的范围和作用域。

查找器并不真正加载模块。 如果它们能找到指定名称的模块，会返回一个 *模块规格说明*，这是对模块导入相关信息的封装，供后续导入机制用于在加载模块时使用。

以下各节描述了有关查找器和加载器协议的更多细节，包括你应该如何创建并注册新的此类对象来扩展导入机制。

> *在 3.4 版本发生变更:* 在之前的 Python 版本中，查找器会直接返回 加载器，现在它们则返回模块规格说明，其中 *包含* 加载器。 加载器仍然在导入期间被使用，但负担的任务有所减少。

5.3.3. 导入钩子

导入机制被设计为可扩展；其中的基本机制是 *导入钩子*。 导入钩子有两种类型: *元钩子* 和 *导入路径钩子*。

元钩子在导入过程开始时被调用，此时任何其他导入过程尚未发生，但 `sys.modules` 缓存查找除外。 这允许元钩子重载 `sys.path` 过程、冻结模块甚至内置模块。 元钩子的注册是通过向 `sys.meta_path` 添加新的查找器对象，具体如下所述。

导入路径钩子是作为 `sys.path` (或 `package.__path__`) 过程的一部分，在遇到它们所关联的路径项的时候被调用。 导入路径钩子的注册是通过向 `sys.path_hooks` 添加新的可调用对象，具体如下所述。

5.3.4. 元路径

当指定名称的模块在 `sys.modules` 中找不到时，Python 会接着搜索 `sys.meta_path`，其中包含元路径查找器对象列表。 这些查找器将按顺序被查询以确定它们是否知道如何处理该名称的模块。 元路径查找器必须实现名为 `find_spec()` 的方法，它接受三个参数：名称、导入路径和（可选的）目标模块。 元路径查找器可使用任何策略来确定它是否能处理指定名称的模块。

如果元路径查找器知道如何处理指定名称的模块，它将返回一个说明对象。 如果它不能处理该名称的模块，则会返回 `None`。 如果 `sys.meta_path` 处理过程到达列表末尾仍未返回说明对象，则将引发 `ModuleNotFoundError`。 任何其他被引发异常将直接向上传播，并放弃导入过程。

元路径查找器的 `find_spec()` 方法调用将附带两个或三个参数。 第一个是被导入模块的完整限定名称，例如 `foo.bar.baz`。 第二个参数是供模块搜索使用的路径条目。 对于最高层级模块，第二个参数为 `None`，但对于子模块或子包，第二个参数为父包的 `__path__` 属性的值。 如果相应折 `__path__` 属性无法访问，则会引发 `ModuleNotFoundError`。 第三个参数是将被作为稍后加载目标的现有模块对象。 导入系统仅会在重加载期间传入一个目标模块。

对于单个导入请求可以多次遍历元路径。 例如，假设所涉及的模块都尚未被缓存，则导入 `foo.bar.baz` 将首先执行顶级的导入，在每个元路径查找器 (`mpf`) 上调用 `mpf.find_spec("foo", None, None)`。 在导入 `foo` 之后，`foo.bar` 将通过第二次遍历元路径来导入，调用 `mpf.find_spec("foo.bar", foo.__path__, None)`。 一旦 `foo.bar` 完成导入，最后一次遍历将调用 `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)`。

有些元路径查找器只支持顶级导入。 当把 `None` 以外的对象作为第三个参数传入时，这些导入器将总是返回 `None`。

Python 的默认 `sys.meta_path` 具有三种元路径查找器，一种知道如何导入内置模块，一种知道如何导入冻结模块，还有一种知道如何导入来自 import path 的模块 (即 path based finder)。

> *在 3.4 版本发生变更:* 元路径查找器的 `find_spec()` 方法替代了 `find_module()`，后者现已被弃用。 虽然它仍将可以不加修改地继续使用，但导入机制仅会在查找器未实现 `find_spec()` 时尝试使用它。

> *在 3.10 版本发生变更:* 导入系统使用 `find_module()` 现在将引发 `ImportWarning`。

> *在 3.12 版本发生变更:* `find_module()` 已被移除。 请改用 `find_spec()`。

5.4. 加载

当一个模块说明被找到时，导入机制将在加载该模块时使用它（及其所包含的加载器）。 下面是导入的加载部分所发生过程的简要说明:

```
module = None
if spec.loader is not None and hasattr(spec.loader, 'create_module'):
    # 假定 'exec_module' 也将在该加载器上定义。
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)
# 导入相关的模块属性在此设置:
_init_module_attrs(spec, module)

if spec.loader is None:
    # 不受支持
    raise ImportError
if spec.origin is None and spec.submodule_search_locations is not None:
    # 命名空间包
    sys.modules[spec.name] = module
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
else:
    sys.modules[spec.name] = module
    try:
        spec.loader.exec_module(module)
    except BaseException:
        try:
            del sys.modules[spec.name]
        except KeyError:
            pass
        raise
return sys.modules[spec.name]
```

请注意以下细节:

- 如果在 `sys.modules` 中存在指定名称的模块对象，导入操作会已经将其返回。
- 在加载器执行模块代码之前，该模块将存在于 `sys.modules` 中。 这一点很关键，因为该模块代码可能（直接或间接地）导入其自身；预先将其添加到 `sys.modules` 可防止在最坏情况下的无限递归和最好情况下的多次加载。
- 如果加载失败，则该模块 -- 只限加载失败的模块 -- 将从 `sys.modules` 中移除。 任何已存在于 `sys.modules` 缓存的模块，以及任何作为附带影响被成功加载的模块仍会保留在缓存中。 这与重新加载不同，后者会把即使加载失败的模块也保留在 `sys.modules` 中。
- 在模块创建完成但还未执行之前，导入机制会设置导入相关模块属性（在上面的示例伪代码中为“_init_module_attrs”），详情参见 后续部分。

- 模块执行是加载的关键时刻，在此期间将填充模块的命名空间。 执行会完全委托给加载器，由加载器决定要填充的内容和方式。
- 在加载过程中创建并传递给 exec_module() 的模块并不一定就是在导入结束时返回的模块 [2]。

> *在 3.4 版本发生变更:* 导入系统已经接管了加载器建立样板的责任。 这些在以前是由 `importlib.abc.Loader.load_module()` 方法来执行的。

5.4.1. 加载器

模块加载器提供关键的加载功能：模块执行。 导入机制调用 `importlib.abc.Loader.exec_module()` 方法并传入一个参数来执行模块对象。 从 `exec_module()` 返回的任何值都将被忽略。

加载器必须满足下列要求:

- 如果模块是一个 Python 模块（而非内置模块或动态加载的扩展），加载器应该在模块的全局命名空间 (`module.__dict__`) 中执行模块的代码。
- 如果加载器无法执行指定模块，它应该引发 `ImportError`，不过在 `exec_module()` 期间引发的任何其他异常也会被传播。

在许多情况下，查找器和加载器可以是同一对象；在此情况下 `find_spec()` 方法将返回一个规格说明，其中加载器会被设为 `self`。

模块加载器可以选择通过实现 `create_module()` 方法在加载期间创建模块对象。 它接受一个参数，即模块规格说明，并返回新的模块对象供加载期间使用。 `create_module()` 不需要在模块对象上设置任何属性。 如果模块返回 `None`，导入机制将自行创建新模块。

> *Added in version 3.4:* 加载器的 `create_module()` 方法。

> *在 3.4 版本发生变更:* `load_module()` 方法被 `exec_module()` 所替代，导入机制会对加载的所有样板责任作出假定。
>
> 为了与现有的加载器兼容，导入机制会使用导入器的 `load_module()` 方法，如果它存在且导入器也未实现 `exec_module()`。 但是，`load_module()` 现已弃用，加载器应该转而实现 `exec_module()`。
>
> 除了执行模块之外，`load_module()` 方法必须实现上文描述的所有样板加载功能。 所有相同的限制仍然适用，并带有一些附加规定:
>
> - 如果 `sys.modules` 中存在指定名称的模块对象，加载器必须使用已存在的模块。 （否则 `importlib.reload()` 将无法正确工作。） 如果该名称模块不存在于 `sys.modules` 中，加载器必须创建一个新的模块对象并将其加入 `sys.modules`。
> - 在加载器执行模块代码之前，模块 *必须* 存在于 `sys.modules` 之中，以防止无限递归或多次加载。
> - 如果加载失败，加载器必须移除任何它已加入到 `sys.modules` 中的模块，但它必须 **仅限** 移除加载失败的模块，且所移除的模块应为加载器自身显式加载的。

> *在 3.5 版本发生变更:* 当 `exec_module()` 已定义但 `create_module()` 未定义时将引发 `DeprecationWarning`。

> *在 3.6 版本发生变更:* 当 `exec_module()` 已定义但 `create_module()` 未定义时将引发 `ImportError`。

> *在 3.10 版本发生变更:* 使用 `load_module()` 将引发 `ImportWarning`。

5.4.2. 子模块

当使用任意机制 (例如 `importlib` API, `import` 及 `import-from` 语句或者内置的 `__import__()`) 加载一个子模块时，父模块的命名空间中会添加一个对子模块对象的绑定。 例如，如果包 `spam` 有一个子模块 `foo`，则在导入 `spam.foo` 之后， `spam` 将具有一个 绑定到相应子模块的 `foo` 属性。 假如现在有如下的目录结构:

```
spam/
    __init__.py
    foo.py
```

并且 `spam/__init__.py` 中有如下几行内容:

```
from .foo import Foo
```

那么执行如下代码将把 `foo` 和 `Foo` 的名称绑定添加到 `spam` 模块中:

```
>>> import spam
>>> spam.foo
<module 'spam.foo' from '/tmp/imports/spam/foo.py'>
>>> spam.Foo
<class 'spam.foo.Foo'>
```

按照通常的 Python 名称绑定规则，这看起来可能会令人惊讶，但它实际上是导入系统的一个基本特性。 保持不变的一点是如果你有 `sys.modules['spam']` 和 `sys.modules['spam.foo']` (例如在上述导入之后就是如此)，则后者必须显示为前者的 `foo` 属性。

5.4.3. 模块规格说明

导入机制在导入期间会使用有关每个模块的多种信息，特别是加载之前。 大多数信息都是所有模块通用的。 模块规格说明的目的是基于每个模块来封装这些导入相关信息。

在导入期间使用规格说明可允许状态在导入系统各组件之间传递，例如在创建模块规格说明的查找器和执行模块的加载器之间。 最重要的一点是，它允许导入机制执行加载的样板操作，在没有模块规格说明的情况下这是加载器的责任。

模块的规格说明将作为 `module.__spec__` 公开。 正确设置 `__spec__` 将同时应用于 解释器启动期间初始化的模块。 唯一的例外是 `__main__`，其中的 `__spec__` 会 在某些情况下设为 None。

请参阅 `ModuleSpec` 了解有关模块规格的详细内容。

> *Added in version 3.4.*

5.4.4. 模块的 __path__ 属性

__path__ 属性应当是一个（可能为空的）枚举将用于查找包的子模块的位置的字符串 sequence。根据定义，如果一个模块具有 __path__ 属性，它就是一个 package。

包的 __path__ 属性会在导入其子包期间被使用。 在导入机制内部，它的功能与 sys.path 基本相同，即在导入期间提供一个模拟搜索位置列表。 但是，__path__ 相比 sys.path 通常要受到更多约束。

作用于 sys.path 的规则同样适用于包的 __path__。 sys.path_hooks (见下文) 会在遍历包的 __path__ 时被查询。

包的 __init__.py 文件可以设置或更改包的 __path__ 属性，而且这是在 **PEP 420** 之前实现命名空间包的典型方式。 随着 **PEP 420** 的引入，命名空间包不再需要提供仅包含 __path__ 操控代码的 __init__.py 文件；导入机制会自动为命名空间包正确地设置 __path__。

5.4.5. 模块的 repr

默认情况下，全部模块都具有一个可用的 repr，但是你可以依据上述的属性设置，在模块的规格说明中更为显式地控制模块对象的 repr。

如果模块具有 spec (__spec__)，导入机制将尝试用它来生成一个 repr。 如果生成失败或找不到 spec，导入系统将使用模块中的各种可用信息来制作一个默认 repr。 它将尝试使用 module.__name__, module.__file__ 以及 module.__loader__ 作为 repr 的输入，并将任何丢失的信息赋为默认值。

以下是所使用的确切规则:

- 如果模块具有 __spec__ 属性，其中的规格信息会被用来生成 repr。 被查询的属性有 "name", "loader", "origin" 和 "has_location" 等等。
- 如果模块具有 __file__ 属性，这会被用作模块 repr 的一部分。
- 如果模块没有 __file__ 但是有 __loader__ 且取值不为 None，则加载器的 repr 会被用作模块 repr 的一部分。
- 对于其他情况，仅在 repr 中使用模块的 __name__。

> *在 3.12 版本发生变更:* module_repr() 自 Python 3.4 起已被弃用，在 Python 3.12 中已被移除且不会在模块的 repr 计算期间被调用。

5.4.6. 已缓存字节码的失效

在 Python 从 .pyc 文件加载已缓存字节码之前，它会检查缓存是否由最新的 .py 源文件所生成。默认情况下，Python 通过在所写入缓存文件中保存源文件的最新修改时间戳和大小来实现这一点。在运行时，导入系统会通过比对缓存文件中保存的元数据和源文件的元数据确定该缓存的有效性。

Python 也支持"基于哈希的"缓存文件，即保存源文件内容的哈希值而不是其元数据。 存在两种基于哈希的 .pyc 文件：检查型和非检查型。 对于检查型基于哈希的 .pyc 文件，Python 会通过求哈希源文件并将结果哈希值与缓存文件中的哈希值比对来确定缓存有效性。 如果检查型基于哈希的缓存

文件被确定为失效，Python 会重新生成并写入一个新的检查型基于哈希的缓存文件。 对于非检查型 `.pyc` 文件，只要其存在 Python 就会直接认定缓存文件有效。 确定基于哈希的 `.pyc` 文件有效性的行为可通过 `--check-hash-based-pycs` 旗标来重载。

> *在 3.7 版本发生变更:* 增加了基于哈希的 `.pyc` 文件。在此之前，Python 只支持基于时间戳来确定字节码缓存的有效性。

5.5. 基于路径的查找器

在之前已经提及，Python 带有几种默认的元路径查找器。 其中之一是 path based finder (`PathFinder`)，它会搜索包含一个 路径条目 列表的 import path。 每个路径条目指定一个用于搜索模块的位置。

基于路径的查找器自身并不知道如何进行导入。 它只是遍历单独的路径条目，将它们各自关联到某个知道如何处理特定类型路径的路径条目查找器。

默认的路径条目查找器集合实现了在文件系统中查找模块的所有语义，可处理多种特殊文件类型例如 Python 源码 (`.py` 文件)，Python 字节码 (`.pyc` 文件) 以及共享库 (例如 `.so` 文件)。 在标准库中 `zipimport` 模块的支持下，默认路径条目查找器还能处理所有来自 zip 文件的上述文件类型。

路径条目不必仅限于文件系统位置。 它们可以指向 URL、数据库查询或可以用字符串指定的任何其他位置。

基于路径的查找器还提供了额外的钩子和协议以便能扩展和定制可搜索路径条目的类型。 例如，如果你想要支持网络 URL 形式的路径条目，你可以编写一个实现 HTTP 语义在网络上查找模块的钩子。 这个钩子（可调用对象）应当返回一个支持下述协议的 path entry finder，以被用来获取一个专门针对来自网络的模块的加载器。

预先的警告：本节和上节都使用了 *查找器* 这一术语，并通过 meta path finder 和 path entry finder 两个术语来明确区分它们。 这两种类型的查找器非常相似，支持相似的协议，且在导入过程中以相似的方式运作，但关键的一点是要记住它们是有微妙差异的。 特别地，元路径查找器作用于导入过程的开始，主要是启动 `sys.meta_path` 遍历。

相比之下，路径条目查找器在某种意义上说是基于路径的查找器的实现细节，实际上，如果需要从 `sys.meta_path` 移除基于路径的查找器，并不会有任何路径条目查找器被唤起。

5.5.1. 路径条目查找器

path based finder 会负责查找和加载通过 path entry 字符串来指定位置的 Python 模块和包。 多数路径条目所指定的是文件系统中的位置，但它们并不必受限于此。

作为一种元路径查找器，path based finder 实现了上文描述的 `find_spec()` 协议，但是它还对外公开了一些附加钩子，可被用来定制模块如何从 import path 查找和加载。

有三个变量由 path based finder, `sys.path`, `sys.path_hooks` 和 `sys.path_importer_cache` 所使用。 包对象的 `__path__` 属性也会被使用。 它们提供了可用于定制导入机制的额外方式。

sys.path 包含一个提供模块和包搜索位置的字符串列表。 它初始化自 PYTHONPATH 环境变量以及多种其他特定安装和实现专属的默认位置。 sys.path 中的条目可指定文件系统中的目录、zip 文件及其他可用于搜索模块的潜在"位置"（参见 site 模块），例如 URL 或数据库查询等。 在 sys.path 中应当只有字符串；所有其他数据类型都会被忽略。

path based finder 是一种 meta path finder，因此导入机制会通过调用上文描述的基于路径的查找器的 find_spec() 方法来启动 import path 搜索。 当要向 find_spec() 传入 path 参数时，它将是一个可遍历的字符串列表 —— 通常为用来在其内部进行导入的包的 __path__ 属性。 如果 path 参数为 None，这表示最高层级的导入，将会使用 sys.path。

基于路径的查找器会迭代搜索路径中的每个条目，并且每次都查找与路径条目对应的 path entry finder (PathEntryFinder)。 因为这种操作可能很耗费资源 (例如搜索会有 stat() 调用的开销)，基于路径的查找器会维持一个将路径条目映射到路径条目查找器的缓存。 这个缓存是在 sys.path_importer_cache 中维护的 (尽管如此命名，但这个缓存实际存放的是查找器对象而非仅限于 importer 对象)。 通过这种方式，对特定 path entry 位置的 path entry finder 的高耗费搜索只需进行一次。 用户代码可以自由地从 sys.path_importer_cache 移除缓存条目以强制基于路径的查找器再次执行路径条目搜索。

如果路径条目不存在于缓存中，基于路径的查找器会迭代 sys.path_hooks 中的每个可调用对象。对此列表中的每个 路径条目钩子 的调用会带有一个参数，即要搜索的路径条目。 每个可调用对象或是返回可处理路径条目的 path entry finder，或是引发 ImportError。 基于路径的查找器使用 ImportError 来表示钩子无法找到与 path entry 相对应的 path entry finder。 该异常会被忽略并继续进行 import path 的迭代。 每个钩子应该期待接收一个字符串或字节串对象；字节串对象的编码由钩子决定（例如可以是文件系统使用的编码 UTF-8 或其它编码），如果钩子无法解码参数，它应该引发 ImportError。

如果 sys.path_hooks 迭代结束时没有返回 path entry finder，则基于路径的查找器 find_spec() 方法将在 sys.path_importer_cache 中存入 None (表示此路径条目没有对应的查找器) 并返回 None，表示此 meta path finder 无法找到该模块。

如果 sys.path_hooks 中的某个 path entry hook 可调用对象的返回值 *是* 一个 path entry finder，则以下协议会被用来向查找器请求一个模块的规格说明，并在加载该模块时被使用。

当前工作目录 -- 由一个空字符串表示 -- 的处理方式与 sys.path 中的其他条目略有不同。 首先，如果当前工作目录无法被确定或找到，则 sys.path_importer_cache 中不会存放任何值。 其次，每次模块查找都会对当前工作目录的值进行全新查找。 第三，由 sys.path_importer_cache 使用并由 importlib.machinery.PathFinder.find_spec() 返回的路径将是实际的当前工作目录而非空字符串。

5.5.2. 路径条目查找器协议

为了支持模块和已初始化包的导入，也为了给命名空间包提供组成部分，路径条目查找器必须实现 find_spec() 方法。

find_spec() 接受两个参数，即要导入模块的完整限定名称，以及（可选的）目标模块。 find_spec() 返回模块的完全填充好的规格说明。 这个规格说明总是包含"加载器"集合（但有一个

例外）。

为了向导入机制提示该规格说明代表一个命名空间 portion，路径条目查找器会将 `submodule_search_locations` 设为一个包含该部分的列表。

> *在 3.4 版本发生变更:* `find_spec()` 替代了 `find_loader()` 和 `find_module()`，这两者现在都已被弃用，但会在 `find_spec()` 未定义时被使用。
>
> 较旧的路径条目查找器可能会实现这两个已弃用的方法中的一个而没有实现 `find_spec()`。 为保持向后兼容，这两个方法仍会被接受。 但是，如果在路径条目查找器上实现了 `find_spec()`，这两个遗留方法就会被忽略。
>
> `find_loader()` 接受一个参数，即要导入模块的完整限定名称。 `find_loader()` 返回一个 2 元组，其中第一项是加载器而第二项是命名空间 portion。
>
> 为了向后兼容其他导入协议的实现，许多路径条目查找器也同样支持元路径查找器所支持的传统 `find_module()` 方法。 但是路径条目查找器 `find_module()` 方法的调用绝不会带有 `path` 参数（它们被期望记录来自对路径钩子初始调用的恰当路径信息）。
>
> 路径条目查找器的 `find_module()` 方法已弃用，因为它不允许路径条目查找器为命名空间包提供部分。 如果 `find_loader()` 和 `find_module()` 同时存在于一个路径条目查找器中，导入系统将总是调用 `find_loader()` 而不选择 `find_module()`。

> *在 3.10 版本发生变更:* 导入系统对 `find_module()` 和 `find_loader()` 的调用将引发 `ImportWarning`。

> *在 3.12 版本发生变更:* `find_module()` 和 `find_loader()` 已被移除。

5.6. 替换标准导入系统

替换整个导入系统的最可靠机制是移除 `sys.meta_path` 的默认内容，,将其完全替换为自定义的元路径钩子。

一个可行的方式是仅改变导入语句的行为而不影响访问导入系统的其他 API，那么替换内置的 `__import__()` 函数可能就够了。 这种技巧也可以在模块层级上运用，即只在某个模块内部改变导入语句的行为。

想要选择性地预先防止在元路径上从一个钩子导入某些模块（而不是完全禁用标准导入系统），只需直接从 `find_spec()` 引发 `ModuleNotFoundError` 而非返回 `None` 就足够了。 返回后者表示元路径搜索应当继续，而引发异常则会立即终止搜索。

5.7. 包相对导入

相对导入使用前缀点号。 一个前缀点号表示相对导入从当前包开始。 两个或更多前缀点号表示对当前包的上级包的相对导入，第一个点号之后的每个点号代表一级。 例如，给定以下的包布局结构:

```
package/
    __init__.py
```

```
    subpackage1/
        __init__.py
        moduleX.py
        moduleY.py
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

不论是在 `subpackage1/moduleX.py` 还是 `subpackage1/__init__.py` 中，以下导入都是有效的:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
```

绝对导入可以使用 `import <>` 或 `from <> import <>` 语法，但相对导入只能使用第二种形式；其中的原因在于:

```
import XXX.YYY.ZZZ
```

应当提供 `XXX.YYY.ZZZ` 作为可用表达式，但 .moduleY 不是一个有效的表达式。

5.8. 有关 __main__ 的特殊事项

对于 Python 的导入系统来说 `__main__` 模块是一个特殊情况。 正如在 另一节 中所述，`__main__` 模块是在解释器启动时直接初始化的，与 `sys` 和 `builtins` 很类似。 但是，与那两者不同，它并不被严格归类为内置模块。 这是因为 `__main__` 被初始化的方式依赖于唤起解释器所附带的旗标和其他选项。

5.8.1. __main__.__spec__

根据 `__main__` 被初始化的方式，`__main__.__spec__` 会被设置相应值或是 `None`。

当 Python 附加 `-m` 选项启动时，`__spec__` 会被设为相应模块或包的模块规格说明。 `__spec__` 也会在 `__main__` 模块作为执行某个目录，zip 文件或其它 `sys.path` 条目的一部分加载时被填充。

在 其余的情况 下 `__main__.__spec__` 会被设为 `None`，因为用于填充 `__main__` 的代码不直接与可导入的模块相对应:

- 交互型提示
- `-c` 选项
- 从 stdin 运行
- 直接从源码或字节码文件运行

请注意在最后一种情况中 `__main__.__spec__` 总是为 `None`，*即使* 文件从技术上说可以作为一个模块被导入。 如果想要让 `__main__` 中的元数据生效，请使用 `-m` 开关。

还要注意即使是在 `__main__` 对应于一个可导入模块且 `__main__.__spec__` 被相应地设定时，它们仍会被视为 *不同的* 模块。 这是由于以下事实：使用 `if __name__ == "__main__":` 检测来保护的代码块仅会在模块被用来填充 `__main__` 命名空间时而非普通的导入时被执行。

5.9. 参考文献

导入机制自 Python 诞生之初至今已发生了很大的变化。 原始的 包规格说明 仍然可以查阅，但在撰写该文档之后许多相关细节已被修改。

原始的 `sys.meta_path` 规格说明见 **PEP 302**，后续的扩展说明见 **PEP 420**。

PEP 420 为 Python 3.3 引入了 命名空间包。 **PEP 420** 还引入了 `find_loader()` 协议作为 `find_module()` 的替代。

PEP 366 描述了新增的 `__package__` 属性，用于在模块中的显式相对导入。

PEP 328 引入了绝对和显式相对导入，并初次提出了 `__name__` 语义，最终由 **PEP 366** 为 `__package__` 加入规范描述。

PEP 338 定义了将模块作为脚本执行。

PEP 451 在 spec 对象中增加了对每个模块导入状态的封装。 它还将加载器的大部分样板责任移交回导入机制中。 这些改变允许弃用导入系统中的一些 API 并为查找器和加载器增加一些新的方法。

备注

[1] 参见 `types.ModuleType`。

[2] importlib 实现避免直接使用返回值。 而是通过在 `sys.modules` 中查找模块名称来获取模块对象。 这种方式的间接影响是被导入的模块可能在 `sys.modules` 中替换其自身。 这属于具体实现的特定行为，不保证能在其他 Python 实现中起作用。