定义扩展模块

针对 CPython 的 C 扩展就是一个共享库（例如，Linux 上的 `.so` 文件，Windows 上的 `.pyd` DLL），它可被加载到 Python 进程中（例如，它可以用兼容的编译器设置进行编译），并且它会导出一个 初始化函数。

要在默认情况下可被导入 (也就是说，通过 `importlib.machinery.ExtensionFileLoader`)，共享库必须在 `sys.path` 中可用，并且必须命名为模块名之后加一个在 `importlib.machinery.EXTENSION_SUFFIXES` 中列出的扩展名。

> **备注：** 构建、打包和分发扩展模块最好使用第三方工具完成，并且超出了本文的范围。一个合适的工具是Setuptools，其文档可以在 https://setuptools.pypa.io/en/latest/setuptools.html 上找到。

通常，初始化函数将返回一个使用 `PyModuleDef_Init()` 来初始化的模块定义。 这允许将创建过程拆分为几个阶段：

- 在任何实质性代码被执行之前，Python 可以确定该模块支持哪些功能，并且可以调整环境或者拒绝加载不兼容的扩展。
- 默认情况下，Python 本身会创建模块对象 -- 也就是说，它所做的与类的 `object.__new__()` 所做的相当。 它还会设置一些初始属性如 `__package__` 和 `__loader__`。
- 在此之后，模块对象将使用扩展专属的代码来初始化 -- 相当于类的 `__init__()`。

这被称为 *多阶段初始化* 以便区别旧式 (但仍受支持的) *单阶段初始化* 方案，旧式方案的初始化函数会返回一个构造完成的模块。 请参阅下面的 单阶段初始化 小节了解详情。

> *在 3.5 版本发生变更:* 增加了对多阶段初始化的支持 (**PEP 489**)。

多个模块实例

在默认情况下，扩展模块都不是单例。 举例来说，如果 `sys.modules` 条目被移除并且模块被重新导入，则会创建一个新的模块对象，并且通常会以新的方法和类型对象充实其内容。 旧模块将成为正常垃圾回收的目标。 这类似于纯 Python 模块的行为。modules.

额外的模块实例可能会在 子解释器 中或者 Python 运行时重新初始化之后 (`Py_Finalize()` 和 `Py_Initialize()`) 被创建。 在这些情况下，模块实例间共享 Python 对象可能导致程序崩溃或未定义的行为。

为避免这种问题，每个扩展模块的实例都应当是 *隔离的*: 对一个实例的修改不应隐式地影响其他的实例，以及模块所拥有的全部状态，包括对 Python 对象的引发，都应当是特定模块实例专属的。 请参阅 隔离扩展模块 了解更多的细节和实用的指南。

一个避免这些问题的简单方式是 针对重复的初始化引发一个错误。

所有模块都应当支持 子解释器，否则就要显式地发出缺乏支持的信号 。 这往往是通过隔离或阻止重复的初始化，如上文所述。 一个模块也可以使用 `Py_mod_multiple_interpreters` 槽位将其限制于主解释器中。

初始化函数

由扩展模块定义的初始化函数具有以下签名：

`PyObject *PyInit_modulename(void)`

其名称应为 `PyInit_<name>`，其中 `<name>` 要替换为模块的名称。

对于名称仅包含 ASCII 字符的模块，函数必须被命名为 `PyInit_<name>`，其中 `<name>` 将用模块的名称为替换。 当使用 多阶段初始化 时，允许包含非 ASCII 字符的模块名称。 在此情况下，初始化函数名称为 `PyInitU_<name>`，其中 `<name>` 将使用 Python 的 *punycode* 编码格式来编码并将连字符替换为下划线。 在 Python 中：

```
def initfunc_name(name):
    try:
        suffix = b'_' + name.encode('ascii')
    except UnicodeEncodeError:
        suffix = b'U_' + name.encode('punycode').replace(b'-', b'_')
    return b'PyInit' + suffix
```

建议使用一个辅助宏来定义初始化函数：

PyMODINIT_FUNC

声明一个扩展模块初始化函数。 这个宏：

- 指定了 PyObject* 返回类型，
- 添加平台所需的任何特殊链接声明，以及
- 对于 C++，将函数声明为 `extern "C"`。

例如,一个名为 `spam` 模块可以这样定义：

```
static struct PyModuleDef spam_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "spam",
    ...
};

PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModuleDef_Init(&spam_module);
}
```

可以通过定义多个初始化函数从一个共享库导出多个模块。 不过，导入它们需要使用符号链接或自定义导入器，因为默认只有与文件名对应的函数才会被发现。 请参阅 **PEP 489** 中的 Multiple modules in one library 一节了解详情。

初始化函数通常是定义于模块的 C 源代码中唯一的非 `static` 条目。

多阶段初始化

通常，初始化函数 (`PyInit_modulename`) 返回一个带有非 `NULL` 的 `m_slots` 的 `PyModuleDef` 实例。 在它被返回之前，这个 `PyModuleDef` 实例必须先使用以下函数来初始化：instance must be initialized using the following function:

`PyObject` ***PyModuleDef_Init**(`PyModuleDef` *def)*

属于 稳定 ABI 自 3.5 版起.

确保模块定义是一个正确初始化的 Python 对象，并正确报告其类型以及引用计数。

返回强制转换为 `PyObject*` 的 *def*，或者如果出生错误则返回 `NULL`。

调用此函数是 多阶段初始化 所必需的。 它不应在其他上下文中被使用。

请注意 Python 会假定 `PyModuleDef` 结构体是静态分配的。 此函数可以返回一个新引用或借入引用；这个引用不可被释放。

> *Added in version 3.5.*

旧式的单阶段初始化

注意: 单阶段初始化是一种用于初始化扩展模块的旧机制，它具有已知的缺点和设计瑕疵。 建议扩展模块作者改用多阶段初始化。

在单阶段初始化中，初始化函数 (`PyInit_modulename`) 应当完成模块对象的创建、填充和返回操作。通常使用 `PyModule_Create()` 和 `PyModule_AddObjectRef()` 等函数实现该流程。

单阶段初始化与 默认方式 的主要区别如下：

- 单阶段模块本质上是（更准确地说，*包含*）"单例对象"。

  当模块首次初始化时，Python会保存该模块 `__dict__` 中的内容（通常包括模块的函数和类型等）。

  对于后续导入操作，Python不会再次调用初始化函数，而是创建一个带有新 `__dict__` 的模块对象，并将已保存的内容复制到其中。例如，假设有一个单阶段模块 `_testsinglephase` [1] 定义了函数 `sum` 和异常类 `error`：

```
>>> import sys
>>> import _testsinglephase as one
>>> del sys.modules['_testsinglephase']
>>> import _testsinglephase as two
>>> one is two
False
>>> one.__dict__ is two.__dict__
False
>>> one.sum is two.sum
```

```
True
>>> one.error is two.error
True
```

该具体行为应被视为CPython的实现细节。

- 为解决 `PyInit_modulename` 函数不接受 *spec* 参数的限制，导入机制会保存部分状态，并在 `PyInit_modulename` 调用期间将其应用于首个匹配的模块对象。具体表现为：当导入子模块时，该机制会将父包名称自动前置到模块名前。

  单阶段 `PyInit_modulename` 函数应当尽早创建"其所属"模块对象，该操作需在任何其他模块对象创建之前完成。

- 不支持非ASCII模块命名格式（如 `PyInitU_modulename`）。

- 单阶段模块支持模块查找函数如 `PyState_FindModule()`。

[1] `_testsinglephase` 是一个在 CPython 的自我测试套件中使用的内部模块；你的安装版可能包括它也可能不包括它。