初始化，最终化和线程

请参阅 Python 初始化配置 了解如何在初始化之前配置解释器的详情。

在Python初始化之前

在一个植入了 Python 的应用程序中，`Py_Initialize()` 函数必须在任何其他 Python/C API 函数之前被调用；例外的只有个别函数和 全局配置变量。

在初始化Python之前，可以安全地调用以下函数：

- 初始化解释器的函数：
  - `Py_Initialize()`
  - `Py_InitializeEx()`
  - `Py_InitializeFromConfig()`
  - `Py_BytesMain()`
  - `Py_Main()`
  - 运行时预初始化相关函数在 Python 初始化配置 中介绍
- 配置函数：
  - `PyImport_AppendInittab()`
  - `PyImport_ExtendInittab()`
  - `PyInitFrozenExtensions()`
  - `PyMem_SetAllocator()`
  - `PyMem_SetupDebugHooks()`
  - `PyObject_SetArenaAllocator()`
  - `Py_SetProgramName()`
  - `Py_SetPythonHome()`
  - `PySys_ResetWarnOptions()`
  - 配置相关函数在 Python 初始化配置 中介绍
- 信息函数：
  - `Py_IsInitialized()`
  - `PyMem_GetAllocator()`
  - `PyObject_GetArenaAllocator()`
  - `Py_GetBuildInfo()`
  - `Py_GetCompiler()`
  - `Py_GetCopyright()`
  - `Py_GetPlatform()`
  - `Py_GetVersion()`
  - `Py_IsInitialized()`
- 工具
  - `Py_DecodeLocale()`

  - 状态报告和工具相关函数在 Python 初始化配置 中介绍
- 内存分配器：
  - PyMem_RawMalloc()
  - PyMem_RawRealloc()
  - PyMem_RawCalloc()
  - PyMem_RawFree()
- 同步：
  - PyMutex_Lock()
  - PyMutex_Unlock()

备注： 虽然它们看起来与上面列出的某些函数类似，但以下函数 **不应** 在解释器被初始化之前调用: Py_EncodeLocale(), Py_GetPath(), Py_GetPrefix(), Py_GetExecPrefix(), Py_GetProgramFullPath(), Py_GetPythonHome(), Py_GetProgramName(), PyEval_InitThreads() 和 Py_RunMain()。

全局配置变量

Python 有负责控制全局配置中不同特性和选项的变量。这些标志默认被 命令行选项。

当一个选项设置一个旗标时，该旗标的值将是设置选项的次数。 例如，`-b` 会将 Py_BytesWarningFlag 设为 1 而 `-bb` 会将 Py_BytesWarningFlag 设为 2.

int **Py_BytesWarningFlag**

此 API 仅为向下兼容而保留：应当改为设置 PyConfig.bytes_warning，参见 Python 初始化配置。

当将 bytes 或 bytearray 与 str 比较或者将 bytes 与 int 比较时发出警告。 如果大于等于 `2` 则报错。

由 -b 选项设置。

> *Deprecated since version 3.12, will be removed in version 3.15.*

int **Py_DebugFlag**

此 API 仅为向下兼容而保留：应当改为设置 PyConfig.parser_debug，参见 Python 初始化配置。

开启解析器调试输出（限专家使用，依赖于编译选项）。

由 -d 选项和 PYTHONDEBUG 环境变量设置。

> *Deprecated since version 3.12, will be removed in version 3.15.*

int **Py_DontWriteBytecodeFlag**

此 API 仅为向下兼容而保留：应当改为设置 PyConfig.write_bytecode，参见 Python 初始化配置。

如果设置为非零, Python 不会在导入源代码时尝试写入 `.pyc` 文件

由 `-B` 选项和 `PYTHONDONTWRITEBYTECODE` 环境变量设置。

> *Deprecated since version 3.12, will be removed in version 3.15.*

`int` **Py_FrozenFlag**

此 API 仅为向下兼容而保留：应当改为设置 `PyConfig.pathconfig_warnings`，参见 Python 初始化配置。

当在 `Py_GetPath()` 中计算模块搜索路径时屏蔽错误消息。

由 `_freeze_importlib` 和 `frozenmain` 程序使用的私有旗标。

> *Deprecated since version 3.12, will be removed in version 3.15.*

`int` **Py_HashRandomizationFlag**

此 API 仅为向下兼容而保留：应当改为设置 `PyConfig.hash_seed` 和 `PyConfig.use_hash_seed`，参见 Python 初始化配置。

如果 `PYTHONHASHSEED` 环境变量被设为非空字符串则设为 `1`。

如果该旗标为非零值，则读取 `PYTHONHASHSEED` 环境变量来初始化加密哈希种子。

> *Deprecated since version 3.12, will be removed in version 3.15.*

`int` **Py_IgnoreEnvironmentFlag**

此 API 仅为向下兼容而保留：应当改为设置 `PyConfig.use_environment`，参见 Python 初始化配置。

忽略所有 `PYTHON*` 环境变量，例如可能设置的 `PYTHONPATH` 和 `PYTHONHOME`。

由 `-E` 和 `-I` 选项设置。

> *Deprecated since version 3.12, will be removed in version 3.15.*

`int` **Py_InspectFlag**

此 API 被保留用于向下兼容：应当改为采用设置 `PyConfig.inspect`，参见 Python 初始化配置。

当将脚本作为第一个参数传入或是使用了 `-c` 选项时，则会在执行该脚本或命令后进入交互模式，即使在 `sys.stdin` 并非一个终端时也是如此。

由 `-i` 选项和 `PYTHONINSPECT` 环境变量设置。

> *Deprecated since version 3.12, will be removed in version 3.15.*

`int` **Py_InteractiveFlag**

此 API 被保留用于向下兼容：应当改为采用设置 `PyConfig.interactive`，参见 Python 初始化配置。

由 `-i` 选项设置。

> *Deprecated since version 3.12, will be removed in version 3.15.*

`int` **Py_IsolatedFlag**

此 API 被保留用于向下兼容：应当改为设置 `PyConfig.isolated`，参见 Python 初始化配置。

以隔离模式运行 Python. 在隔离模式下 `sys.path` 将不包含脚本的目录或用户的 site-packages 目录。

由 `-I` 选项设置。

> *Added in version 3.4.*

> *Deprecated since version 3.12, will be removed in version 3.15.*

`int` **Py_LegacyWindowsFSEncodingFlag**

此 API 被保留用于向下兼容：应当改为设置 `PyPreConfig.legacy_windows_fs_encoding`，参见 Python 初始化配置。

如果该旗标为非零值，则使用 `mbcs` 编码和 `replace` 错误处理器，而不是 UTF-8 编码和 `surrogatepass` 错误处理器作用 filesystem encoding and error handler。

如果 `PYTHONLEGACYWINDOWSFSENCODING` 环境变量被设为非空字符串则设为 `1`。

更多详情请参阅 **PEP 529**。

Availability: Windows.

> *Deprecated since version 3.12, will be removed in version 3.15.*

`int` **Py_LegacyWindowsStdioFlag**

此 API 被保留用于向下兼容：应当改为设置 `PyConfig.legacy_windows_stdio`，参见 Python 初始化配置。

如果该旗标为非零值，则会使用 `io.FileIO` 而不是 `io._WindowsConsoleIO` 作为 `sys` 标准流。

如果 `PYTHONLEGACYWINDOWSSTDIO` 环境变量被设为非空字符串则设为 `1`。

有关更多详细信息，请参阅 **PEP 528**。

Availability: Windows.

> *Deprecated since version 3.12, will be removed in version 3.15.*

`int` **Py_NoSiteFlag**

此 API 被保留用于向下兼容：应当改为设置 `PyConfig.site_import`，参见 Python 初始化配置。

禁用 site 的导入及其所附带的基于站点对 sys.path 的操作。 如果 site 会在稍后被显式地导入也会禁用这些操作 (如果你希望触发它们则应调用 site.main())。

由 -S 选项设置。

> *Deprecated since version 3.12, will be removed in version 3.15.*

int **Py_NoUserSiteDirectory**

此 API 被保留用于向下兼容：应当改为设置 PyConfig.user_site_directory，参见 Python 初始化配置。

不要将 用户 site-packages 目录 添加到 sys.path。

由 -s 和 -I 选项以及 PYTHONNOUSERSITE 环境变量设置。

> *Deprecated since version 3.12, will be removed in version 3.15.*

int **Py_OptimizeFlag**

此 API 被保留用于向下兼容：应当改为 PyConfig.optimization_level，参见 Python 初始化配置。

由 -O 选项和 PYTHONOPTIMIZE 环境变量设置。

> *Deprecated since version 3.12, will be removed in version 3.15.*

int **Py_QuietFlag**

此 API 被保留用于向下兼容：应当改为设置 PyConfig.quiet，参见 Python 初始化配置。

即使在交互模式下也不显示版权和版本信息。

由 -q 选项设置。

> *Added in version 3.2.*

> *Deprecated since version 3.12, will be removed in version 3.15.*

int **Py_UnbufferedStdioFlag**

此 API 被保留用于向下兼容：应当改为设置 PyConfig.buffered_stdio，参见 Python 初始化配置。

强制 stdout 和 stderr 流不带缓冲。

由 -u 选项和 PYTHONUNBUFFERED 环境变量设置。

> *Deprecated since version 3.12, will be removed in version 3.15.*

int **Py_VerboseFlag**

此 API 被保留用于向下兼容：应当改为设置 PyConfig.verbose，参见 Python 初始化配置。

每次初始化模块时打印一条消息，显示加载模块的位置（文件名或内置模块）。 如果大于或等于 `2`，则为搜索模块时检查的每个文件打印一条消息。 此外还会在退出时提供模块清理信息。

由 `-v` 选项和 `PYTHONVERBOSE` 环境变量设置。

> *Deprecated since version 3.12, will be removed in version 3.15.*

初始化和最终化解释器

`void` **Py_Initialize**()

属于 稳定 ABI.

初始化 Python 解释器。 在嵌入 Python 的应用程序中，它应当在使用任何其他 Python/C API 函数之前被调用；请参阅 在 Python 初始化之前 了解少数的例外情况。

这将初始化已加载的模块表 (`sys.modules`)，并创建基础模块 `builtins`, `__main__` 和 `sys`。 它还会初始化模块搜索路径 (`sys.path`)。 它不会设置 `sys.argv`；对于此设置请使用 Python 初始化配置 API。 当第二次被调用时（在未先调用 `Py_FinalizeEx()` 的情况下）将不会执行任何操作。 它没有返回值；如果初始化失败则会发生致命错误。

使用 `Py_InitializeFromConfig()` 来自定义 Python 初始化配置。

> **备注：** 在 Windows 上，将控制台模式从 `O_TEXT` 改为 `O_BINARY`，这还将影响使用 C 运行时的非 Python 的控制台使用。

`void` **Py_InitializeEx**(`int` initsigs)

属于 稳定 ABI.

如果 *initsigs* 为 `1` 则此函数的工作方式与 `Py_Initialize()` 类似。 如果 *initsigs* 为 `0`，它将跳过信号处理器的初始化注册，这在将 CPython 作为更大应用程序的一部分嵌入时会很有用处。

使用 `Py_InitializeFromConfig()` 来自定义 Python 初始化配置。

`PyStatus` **Py_InitializeFromConfig**(`const PyConfig` *config)

根据 *config* 配置来初始化 Python，如 使用 PyConfig 初始化 中所描述的。.

请参阅 Python 初始化配置 一节了解有关预初始化解释器，填充运行时配置结构体，以及查询所返回的状态结构体的详情。

`int` **Py_IsInitialized**()

属于 稳定 ABI.

如果 Python 解释器已初始化，则返回真值（非零）；否则返回假值（零）。 在调用 `Py_FinalizeEx()` 之后，此函数将返回假值直到 `Py_Initialize()` 再次被调用。

`int` **Py_IsFinalizing**()

属于 稳定 ABI 自 3.13 版起.

如果主 Python 解释器 正在关闭 则返回真（非零）值。 在其他情况下返回假（零）值。

> *Added in version 3.13.*

`int` **Py_FinalizeEx()**

属于 稳定 ABI 自 3.6 版起.

撤销由 `Py_Initialize()` 完成的所有初始化操作及后续Python/C API函数调用，并销毁自上次调用 `Py_Initialize()` 以来创建但尚未销毁的所有子解释器（参见下文 `Py_NewInterpreter()`）。若在未再次调用 `Py_Initialize()` 的情况下重复执行，该操作将无效。

由于这是 `Py_Initialize()` 的逆向操作，因而它应当在激活同一解释器的同一线程中被调用。这意味着主线程和主解释器。 当 `Py_RunMain()` 仍然运行时则绝不应调用此函数。

通常返回值为 `0`。 如果在最终化（刷新缓冲中的数据）期间发生错误，则返回 `-1`。

请注意，Python会尽最大努力释放解释器分配的所有内存。因此，任何C扩展模块都应确保在后续调用 `Py_Initialize()` 前，正确清理之前分配的所有PyObject对象。否则可能导致安全漏洞和异常行为。

提供此函数的原因有很多。嵌入应用程序可能希望重新启动Python，而不必重新启动应用程序本身。从动态可加载库（或DLL）加载Python解释器的应用程序可能希望在卸载DLL之前释放Python分配的所有内存。在搜索应用程序内存泄漏的过程中，开发人员可能希望在退出应用程序之前释放Python分配的所有内存。

已知问题与注意事项： 模块及模块内对象的销毁顺序是随机的，这可能导致析构函数（`__del__()` 方法）在依赖其他对象（甚至函数）或模块时执行失败； 由Python动态加载的扩展模块不会被卸载； Python解释器分配的少量内存可能无法释放（如发现内存泄漏请提交报告）； 对象间循环引用占用的内存不会被释放； 无论引用计数如何，所有驻留字符串（interned strings）都将被释放； 扩展模块分配的部分内存可能无法释放； 某些扩展在初始化例程被多次调用时可能出现异常行为（当应用程序多次调用 `Py_Initialize()` 和 `Py_FinalizeEx()` 时会发生这种情况）； `Py_FinalizeEx()` 不可被自身递归调用，因此任何可能作为解释器关闭流程一部分的代码（如 `atexit` 处理器、对象终结器、或在刷新stdout/stderr文件时运行的代码）都不得调用该函数。

引发一个不带参数的 审计事件 `cpython._PySys_ClearAuditHooks`。

> *Added in version 3.6.*

`void` **Py_Finalize()**

属于 稳定 ABI.

这是一个不考虑返回值的 `Py_FinalizeEx()` 的向下兼容版本。

`int` **Py_BytesMain**(`int argc, char **argv`)

属于 稳定 ABI 自 3.8 版起.

类似于 `Py_Main()` 但 *argv* 是一个字节串数组，允许调用方应用程序将文本编码步骤委托给 CPython 运行时。

> *Added in version 3.8.*

`int` **Py_Main**(`int argc, wchar_t **argv`)
属于 稳定 ABI.

标准解释器的主程序，封装了完整的初始化/最终化循环，以及一些附加行为以实现从环境和命令行读取配置设置，然后按照 命令行 的规则执行 `__main__`。

这适用于希望支持完整 CPython 命令行界面的程序，而不仅是在更大应用程序中嵌入 Python 运行时。

argc 和 *argv* 形参与传给 C 程序的 `main()` 函数的形参类似，不同之处在于 *argv* 的条目会先使用 `Py_DecodeLocale()` 转换为 `wchar_t`。 还有一个重要的注意事项是参数列表条目可能会被修改为指向并非被传入的字符串（不过，参数列表所指向的字符串内容不会被修改）。

如果参数列表不是表示一个有效的 Python 命令行则返回值为 `2`，否则将与 `Py_RunMain()` 相同。

在记录于 运行时配置 一节的 CPython 运行时配置 API 文档中（不考虑错误处理），`Py_Main` 大致相当于:

```
PyConfig config;
PyConfig_InitPythonConfig(&config);
PyConfig_SetArgv(&config, argc, argv);
Py_InitializeFromConfig(&config);
PyConfig_Clear(&config);

Py_RunMain();
```

在正常使用中，嵌入式应用程序将调用此函数 *而不是* 直接调用 `Py_Initialize()`, `Py_InitializeEx()` 或 `Py_InitializeFromConfig()`，并且所有设置都将如本文档的其他部分所描述的那样被应用。 如果此函数改在某个先前的运行时初始化 API 调用 *之后* 被调用，那么到底那个环境和命令行配置会被更新将取决于具体的版本（因为它要依赖当运行时被初始化时究竟有哪些设置在它们已被设置一次之后是正确地支持被修改的）。

`int` **Py_RunMain**(`void`)
在完整配置的 CPython 运行时中执行主模块。

执行在命令行或配置中指定的命令 (`PyConfig.run_command`)、脚本 (`PyConfig.run_filename`) 或模块 (`PyConfig.run_module`)。 如果这些值均未设置，则使用 `__main__` 模块的全局命令空间来运行交互式 Python 提示符 (REPL)。

如果 `PyConfig.inspect` 未设置（默认），则当解释器正常退出（也就是说未引发异常）时返回值将为 `0`，未处理的 `SystemExit` 的退出状态，或者对于任何其他未处理异常则为 `1`。

如果 PyConfig.inspect 已设置（例如当使用了 -i 选项时），则当解释器退出时执行将不会返回，而是会使用 __main__ 模块的全局命名空间在交互式 Python 提示符 (REPL) 中恢复。 如果解释器附带异常退出，该异常将在 REPL 会话中被立即引发。 随后函数的返回值将由 *REPL 会话* 的终结方式来决定: 0, 1 或者 SystemExit 的状态，如上文所指明的。

此函数总是会在它返回之前最终化 Python 解释器。

请参阅 Python 配置 查看一个使用 Py_RunMain() 在隔离模式下始终运行定制的 Python 的示例。

int **PyUnstable_AtExit**(PyInterpreterState *interp, void (*func)(void*), void *data)

这是 不稳定 API。它可在次发布版中不经警告地改变。

为目标解释器 *interp* 注册一个 atexit 回调。 这与 Py_AtExit() 类似，但它接受一个显式的解释器和用于回调的数据指针。

必须有一个对应 *interp* 的 attached thread state。

Added in version 3.13.

进程级参数

void **Py_SetProgramName**(const wchar_t *name)

属于 稳定 ABI.

此 API 被保留用于向下兼容：应当改为设置 PyConfig.program_name，参见 Python 初始化配置。

如果要调用该函数，应当在首次调用 Py_Initialize() 之前调用它。 它将告诉解释器程序的 main() 函数的 argv[0] 参数的值（转换为宽字符）。 Py_GetPath() 和下面的某些其他函数会使用它在相对于解释器的位置上查找可执行文件的 Python 运行时库。 默认值是 'python'。参数应当指向静态存储中的一个以零值结束的宽字符串，其内容在程序执行期间不会发生改变。 Python 解释器中的任何代码都不会改变该存储的内容。

使用 Py_DecodeLocale() 解码字节串以得到一个 wchar_t* 字符串。

Deprecated since version 3.11, will be removed in version 3.15.

wchar_t *__Py_GetProgramName__()

属于 稳定 ABI.

返回用 Py_SetProgramName() 设置的程序名称，或默认的名称。 返回的字符串指向静态存储；调用者不应修改其值。

此函数不应在 Py_Initialize() 之前被调用，否则将返回 NULL。

在 3.10 版本发生变更: 现在如果它在 Py_Initialize() 之前被调用将返回 NULL。

> *Deprecated since version 3.13, will be removed in version 3.15:* 改用 PyConfig_Get("executable") (sys.executable)。

wchar_t *Py_GetPrefix()

属于 稳定 ABI.

返回针对已安装的独立于平台文件的 *prefix*。 这是通过基于使用 PyConfig.program_name 设置的程序名称和某些环境变量所派生的一系列复杂规则来获取的；举例来说，如果程序名称为 '/usr/local/bin/python'，则 prefix 为 '/usr/local'。 返回的字符串将指向静态存储；调用方不应修改其值。 这对应于最高层级 Makefile 中的 **prefix** 变量以及在编译时传给 **configure** 脚本的 --prefix 参数。 该值将作为 sys.base_prefix 供 Python 代码使用。 它仅适用于 Unix。 另请参见下一个函数。

此函数不应在 Py_Initialize() 之前被调用，否则将返回 NULL。

> *在 3.10 版本发生变更:* 现在如果它在 Py_Initialize() 之前被调用将返回 NULL。

> *Deprecated since version 3.13, will be removed in version 3.15:* 改用 PyConfig_Get("base_prefix") (sys.base_prefix)。 如果需要处理 虚拟环境 则使用 PyConfig_Get("prefix") (sys.prefix)。

wchar_t *Py_GetExecPrefix()

属于 稳定 ABI.

返回针对已安装的 *依赖于* 平台文件的 *exec-prefix*。 这是通过基于使用 PyConfig.program_name 设置的程序名称和某些环境变量所派生的一系列复杂规则来获取的；举例来说，如果程序名称为 '/usr/local/bin/python'，则 exec-prefix 为 '/usr/local'。 返回的字符串将指向静态存储；调用方不应修改其值。 这对应于最高层级 Makefile 中的 **exec_prefix** 变量以及在编译时传给 **configure** 脚本的 --exec-prefix 参数。 该值将作为 sys.base_exec_prefix 供 Python 代码使用。 它仅适用于 Unix。

背景：当依赖于平台的文件（如可执行文件和共享库）是安装于不同的目录树中的时候 exec-prefix 将会不同于 prefix。 在典型的安装中，依赖于平台的文件可能安装于 the /usr/local/plat 子目录树而独立于平台的文件可能安装于 /usr/local。

总而言之，平台是一组硬件和软件资源的组合，例如所有运行 Solaris 2.x 操作系统的 Sparc 机器会被视为相同平台，但运行 Solaris 2.x 的 Intel 机器是另一种平台，而运行 Linux 的 Intel 机器又是另一种平台。 相同操作系统的不同主要发布版通常也会构成不同的平台。 非 Unix 操作系统的情况又有所不同；这类系统上的安装策略差别巨大因此 prefix 和 exec-prefix 是没有意义的，并将被设为空字符串。 请注意已编译的 Python 字节码是独立于平台的（但并不独立于它们编译时所使用的 Python 版本！）

系统管理员知道如何配置 **mount** 或 **automount** 程序以在平台间共享 /usr/local 而让 /usr/local/plat 成为针对不同平台的不同文件系统。

此函数不应在 Py_Initialize() 之前被调用，否则将返回 NULL。

> *在 3.10 版本发生变更:* 现在如果它在 `Py_Initialize()` 之前被调用将返回 `NULL`。

> *Deprecated since version 3.13, will be removed in version 3.15:* 改用 `PyConfig_Get("base_exec_prefix")` (`sys.base_exec_prefix`)。 如果需要处理 虚拟环境 则使用 `PyConfig_Get("exec_prefix")` (`sys.exec_prefix`)。

`wchar_t *`**`Py_GetProgramFullPath()`**

属于 稳定 ABI.

返回 Python 可执行文件的完整程序名称；这是作为基于程序名称（由 `PyConfig.program_name` 设置）派生默认模块搜索路径的附带影响计算得出的。 返回的字符串将指向静态存储；调用方不应修改其值。 该值将以 `sys.executable` 的名称供 Python 代码访问。

此函数不应在 `Py_Initialize()` 之前被调用，否则将返回 `NULL`。

> *在 3.10 版本发生变更:* 现在如果它在 `Py_Initialize()` 之前被调用将返回 `NULL`。

> *Deprecated since version 3.13, will be removed in version 3.15:* 改用 `PyConfig_Get("executable")` (`sys.executable`)。

`wchar_t *`**`Py_GetPath()`**

属于 稳定 ABI.

返回默认模块搜索路径；这是基于程序名称（由 `PyConfig.program_name` 设置）和某些环境变量计算得出的。 返回的字符串由一系列以依赖于平台的分隔符分开的目录名称组成。 此分隔符在 Unix 和 macOS 上为 `':'`，在 Windows 上为 `';'`。 返回的字符串将指向静态存储；调用方不应修改其值。 列表 `sys.path` 将在解释器启动时使用该值来初始化；它可以在随后被修改（并且通常都会被修改）以变更用于加载模块的搜索路径。

此函数不应在 `Py_Initialize()` 之前被调用，否则将返回 `NULL`。

> *在 3.10 版本发生变更:* 现在如果它在 `Py_Initialize()` 之前被调用将返回 `NULL`。

> *Deprecated since version 3.13, will be removed in version 3.15:* 改用 `PyConfig_Get("module_search_paths")` (`sys.path`)。

`const char *`**`Py_GetVersion()`**

属于 稳定 ABI.

返回 Python 解释器的版本。 这将为如下形式的字符串

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

第一个单词（到第一个空格符为止）是当前的 Python 版本；前面的字符是以点号分隔的主要和次要版本号。 返回的字符串将指向静态存储；调用方不应修改其值。 该值将以 `sys.version` 的名称供 Python 代码使用。

另请参阅 `Py_Version` 常量。

const char *Py_GetPlatform()

属于 稳定 ABI.

返回当前平台的平台标识符。 在 Unix 上，这将以操作系统的“官方”名称为基础，转换为小写形式，再加上主版本号；例如，对于 Solaris 2.x，或称 SunOS 5.x，该值将为 `'sunos5'`。 在 macOS 上，它将为 `'darwin'`。 在 Windows 上它将为 `'win'`。 返回的字符串指向静态存储；调用方不应修改其值。 Python 代码可通过 `sys.platform` 获取该值。

const char *Py_GetCopyright()

属于 稳定 ABI.

返回当前 Python 版本的官方版权字符串，例如

`'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'`

返回的字符串指向静态存储；调用者不应修改其值。 Python 代码可通过 `sys.copyright` 获取该值。

const char *Py_GetCompiler()

属于 稳定 ABI.

返回用于编译当前 Python 版本的编译器指令，为带方括号的形式，例如:

```
"[GCC 2.7.2.2]"
```

返回的字符串指向静态存储；调用者不应修改其值。 Python 代码可以从变量 `sys.version` 中获取该值。

const char *Py_GetBuildInfo()

属于 稳定 ABI.

返回有关当前Python解释器实例的序列号和构建日期和时间的信息，例如：

```
"#67, Aug  1 1997, 22:34:28"
```

返回的字符串指向静态存储；调用者不应修改其值。 Python 代码可以从变量 `sys.version` 中获取该值。

void PySys_SetArgvEx(int argc, wchar_t **argv, int updatepath)

属于 稳定 ABI.

此 API 被保留用于向下兼容：应当改为设置 `PyConfig.argv`, `PyConfig.parse_argv` 和 `PyConfig.safe_path`，参见 Python 初始化配置。

根据 *argc* 和 *argv* 设置 `sys.argv`。 这些形参与传给程序的 `main()` 函数的类似，区别在于第一项应当指向要执行的脚本文件而不是 Python 解释器对应的可执行文件。 如果没有要运行的脚本，则 *argv* 中的第一项可以为空字符串。 如果此函数无法初始化 `sys.argv`，则将使用 `Py_FatalError()` 发出严重情况信号。

如果 *updatepath* 为零，此函数将完成操作。 如果 *updatepath* 为非零值，则此函数还将根据以下算法修改 `sys.path`:

- 如果在 `argv[0]` 中传入一个现有脚本，则脚本所在目录的绝对路径将被添加到 `sys.path` 的开头。
- 在其他情况下 (也就是说，如果 *argc* 为 `0` 或 `argv[0]` 未指向现有文件名)，则将在 `sys.path` 的开头添加一个空字符串，这等价于添加当前工作目录 (`"."`)。

使用 `Py_DecodeLocale()` 解码字节串以得到一个 `wchar_t*` 字符串。

另请参阅 Python 初始化配置 的 `PyConfig.orig_argv` 和 `PyConfig.argv` 成员。

备注: 建议在出于执行单个脚本以外的目的嵌入 Python 解释器的应用传入 `0` 作为 *updatepath*，并在需要时更新 `sys.path` 本身。 参见 **CVE 2008-5983**。

在 3.1.3 之前的版本中，你可以通过在调用 `PySys_SetArgv()` 之后手动弹出第一个 `sys.path` 元素，例如使用:

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

Added in version 3.1.3.

Deprecated since version 3.11, will be removed in version 3.15.

`void` **PySys_SetArgv**(`int argc, wchar_t **argv`)

属于 稳定 ABI.

此 API 仅为向下兼容而保留：应当改为设置 `PyConfig.argv` 并改用 `PyConfig.parse_argv`，参见 Python 初始化配置。

此函数相当于 `PySys_SetArgvEx()` 设置了 *updatepath* 为 `1` 除非 **python** 解释器启动时附带了 `-I`。

使用 `Py_DecodeLocale()` 解码字节串以得到一个 `wchar_t*` 字符串。

另请参阅 Python 初始化配置 的 `PyConfig.orig_argv` 和 `PyConfig.argv` 成员。

在 3.4 版本发生变更: *updatepath* 值依赖于 `-I`。

Deprecated since version 3.11, will be removed in version 3.15.

`void` **Py_SetPythonHome**(`const wchar_t *home`)

属于 稳定 ABI.

此 API 被保留用于向下兼容：应当改为设置 `PyConfig.home`，参见 Python 初始化配置。

设置默认的 "home" 目录，也就是标准 Python 库所在的位置。 请参阅 `PYTHONHOME` 了解该参数字符串的含义。

此参数应当指向静态存储中一个以零值结束的字符串，其内容在程序执行期间将保持不变。Python 解释器中的代码绝不会修改此存储中的内容。

使用 Py_DecodeLocale() 解码字节串以得到一个 wchar_t* 字符串。

> *Deprecated since version 3.11, will be removed in version 3.15.*

wchar_t *Py_GetPythonHome()

属于 稳定 ABI.

返回默认的 "home"，就是由 PyConfig.home 所设置的值，或者在设置了 PYTHONHOME 环境变量的情况下则为该变量的值。

此函数不应在 Py_Initialize() 之前被调用，否则将返回 NULL。

> *在 3.10 版本发生变更:* 现在如果它在 Py_Initialize() 之前被调用将返回 NULL。

> *Deprecated since version 3.13, will be removed in version 3.15:* 改用 PyConfig_Get("home") 或 PYTHONHOME 环境变量。

线程状态和全局解释器锁

除非使用 自由线程 构建的 CPython 版本，否则 Python 解释器并非完全线程安全。 为支持多线程 Python 程序，系统设置了名为 global interpreter lock 或 GIL 的全局锁 —— 当前线程必须获取该锁后才能安全操作Python对象。 若未持有此锁，即便是最简单的操作也可能引发多线程程序问题：例如当两个线程同时递增同一对象的引用计数时，最终引用计数可能只增加一次而非两次。

因此，规则要求只有获得 GIL 的线程才能在 Python对象上执行操作或调用 Python/C API 函数。 为了模拟并发执行，解释器会定期尝试切换线程 (参见 sys.setswitchinterval())。 锁也会在读写文件等可能造成阻塞的 I/O 操作时释放，以便其他 Python 线程可以同时运行。

Python解释器将线程特定的簿记信息存储在名为 PyThreadState 的数据结构中，该结构被称为 thread state。 每个操作系统线程都拥有一个线程本地指针指向 PyThreadState，被该指针引用的线程状态被视为 已附加。

一个线程同一时间只能拥有一个 attached thread state。 已附加线程状态通常等同于持有 GIL，但在 自由线程 构建中例外。 在启用 GIL 的构建版本中，附加 线程状态会阻塞直至获取到 GIL。 但需注意，即使在禁用 GIL 的构建版本中，调用大多数 C API 仍需要已附加线程状态。

通常情况下，使用 Python C API 时总会存在一个 attached thread state。 仅在某些特定情况下（例如处于 Py_BEGIN_ALLOW_THREADS 代码块中），线程才不会有已附加线程状态。 如不确定，可通过检查 PyThreadState_GetUnchecked() 是否返回 NULL 来确认。

从扩展代码分离线程状态

大多数操作 thread state 的扩展代码具有以下简单结构:

```
将线程状态保存到一个局部变量中。
... 执行某些阻塞式的 I/O 操作 ...
```

```
从局部变量中恢复线程状态。
```

这是如此常用因此增加了一对宏来简化它:

```
Py_BEGIN_ALLOW_THREADS
... 执行某些阻塞式的 I/O 操作 ...
Py_END_ALLOW_THREADS
```

Py_BEGIN_ALLOW_THREADS 宏将打开一个新块并声明一个隐藏的局部变量；Py_END_ALLOW_THREADS 宏将关闭这个块。

上面的代码块可扩展为下面的代码:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... 执行某些阻塞式的 I/O 操作 ...
PyEval_RestoreThread(_save);
```

下面介绍这些函数是如何运作的:

attached thread state 持有整个解释器的 GIL。当分离 attached thread state 时，GIL 会被释放，允许其他线程将线程状态附加到自己的线程上，从而获取 GIL 并开始执行。先前 attached thread state 的指针会被存储为局部变量。当执行到 Py_END_ALLOW_THREADS 时，先前 已附加 的线程状态会被传递给 PyEval_RestoreThread()。 该函数将阻塞直到其他线程释放其 线程状态，从而使旧的 线程状态 能够重新附加，并再次调用 C API。

对于 自由线程 构建版本，通常无需考虑 GIL，但在阻塞I/O和长时操作中仍需分离 线程状态。不同之处在于，线程无需等待 GIL 释放即可附加其线程状态，从而实现真正的多核并行。

备注: 调用系统I/O函数是分离 线程状态 的最常见场景，但在执行无需访问Python对象的长时计算（如针对内存缓冲区的压缩或加密运算）前，分离线程状态同样有益。例如标准库中的 `zlib` 和 `hashlib` 模块在压缩或哈希数据时就会分离 线程状态。

非Python创建的线程

当线程通过 Python 专用 API（如 `threading` 模块）创建时，系统会自动为其关联线程状态，因此上述代码是正确的。 然而，当线程直接从 C 创建时（例如通过自带线程管理的第三方库），这些线程不会持有 GIL，因为它们没有 attached thread state。

若需从这些线程调用 Python 代码（常见于前述第三方库提供的回调 API 中），必须首先创建 attached thread state 向解释器注册线程，然后才能使用 Python/C API。 操作完成后，应当分离 线程状态 并最终释放该线程。

PyGILState_Ensure() 和 PyGILState_Release() 函数会自动完成上述的所有操作。 从 C 线程调用到 Python 的典型方式如下:

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();
```

```
/* 在此执行 Python 动作。 */
result = CallSomeFunction();
/* 评估结果或处理异常 */

/* 释放线程。 在此之后不再允许 Python API。 */
PyGILState_Release(gstate);
```

请注意 `PyGILState_*` 系列函数基于单全局解释器假设（由 `Py_Initialize()` 自动创建）。Python虽支持创建附加解释器（通过 `Py_NewInterpreter()`），但混合使用多解释器与 `PyGILState_*` API 不受支持。这是因为 `PyGILState_Ensure()` 及类似函数默认将 thread state 附加 到主解释器，导致线程无法安全地与调用方的子解释器交互。

在非Python线程中支持子解释器

如需在非Python创建的线程中支持子解释器，必须改用 `PyThreadState_*` API 替代传统的 `PyGILState_*` API。

特别需要注意的是，必须从调用函数中保存解释器状态，并将其传递给 `PyThreadState_New()`，该函数会确保新建的 thread state 正确关联目标解释器:

```
/* 创建该线程的函数中 PyInterpreterState_Get() 的返回值*/
PyInterpreterState *interp = ThreadData->interp;
PyThreadState *tstate = PyThreadState_New(interp);
PyThreadState_Swap(tstate);

/* 此时已持有子解释器的GIL，
可在此执行Python操作。 */
result = CallSomeFunction();
/* 评估结果或处理异常 */

/* 销毁线程状态。此后禁止调用任何 Python API。*/
PyThreadState_Clear(tstate);
PyThreadState_DeleteCurrent();
```

有关 fork() 的注意事项

有关线程的另一个需要注意的重要问题是它们在面对 C `fork()` 调用时的行为。 在大多数支持 `fork()` 的系统中，当一个进程执行 fork 之后将只有发出 fork 的线程存在。 这对需要如何处理锁以及CPython 的运行时内所有的存储状态都会有实质性的影响。

只保留"当前"线程这一事实意味着任何由其他线程所持有的锁永远不会被释放。 Python 通过在 fork 之前获取内部使用的锁，并随后释放它们的方式为 `os.fork()` 解决了这个问题。 此外，它还会重置子进程中的任何 Lock 对象。 在扩展或嵌入 Python 时，没有办法通知 Python 在 fork 之前或之后需要获取或重置的附加（非 Python）锁。 需要使用 OS 工具例如 `pthread_atfork()` 来完成同样的事情。 此外，在扩展或嵌入 Python 时，直接调用 `fork()` 而不是通过 `os.fork()` (并返回到或调用至 Python 中) 调用可能会导致某个被 fork 之后失效的线程所持有的 Python 内部锁发生死锁。 `PyOS_AfterFork_Child()` 会尝试重置必要的锁，但并不总是能够做到。

所有其他线程都将结束这一事实也意味着 CPython 的运行时状态必须妥善清理，`os.fork()` 就是这样做的。 这意味着最终化归属于当前解释器的所有其他 `PyThreadState` 对象以及所有其他

PyInterpreterState 对象。 由于这一点以及 "main" 解释器 的特殊性质，fork() 应当只在该解释器 的 "main" 线程中被调用，而 CPython 全局运行时最初就是在该线程中初始化的。 只有当 exec() 将随后立即被调用的情况是唯一的例外。

有关运行时最终化的注意事项

在 interpreter shutdown 的后期阶段，系统会先尝试等待非守护线程退出（此过程可能被 KeyboardInterrupt 中断），并执行 atexit 注册的函数。 此时运行时状态会被标记为 *正在终结*。 Py_IsFinalizing() 和 sys.is_finalizing() 均返回真值。 在此状态下，仅允许发起终结流程的 *终结线程* （通常为主线程）获取 GIL。

如果非终结线程的其他线程在终结阶段尝试显式或隐式附加 thread state，该线程将进入 **永久阻塞状态** ——直至程序退出前都无法恢复。 多数情况下这不会造成危害，但如果终结过程的后续阶段试图获取被阻塞线程持有的锁，或以其他方式等待该线程响应，则可能引发死锁。

粗暴？确实如此。 但这样做能避免随机崩溃，以及当这些线程在 CPython 3.13 及更早版本中被强制退出时，调用栈上游可能出现的 C++ 资源未释放问题。 CPython 运行时的 thread state C API在设计之初就未考虑在 thread state 附加阶段提供错误报告或处理机制，因此无法优雅处理这种情况。 若要改变现状，就需要新增稳定的 C API，并重写 CPython 生态中绝大多数 C 代码来适配这些带错误处理的新 API。

高阶 API

这些是在编写 C 扩展代码或在嵌入 Python 解释器时最常用的类型和函数：

type **PyInterpreterState**

属于 受限 API （作为不透明的结构体）.

该数据结构代表多个合作线程所共享的状态。 属于同一解释器的线程将共享其模块管理以及其他一些内部条目。 该结构体中不包含公有成员。

最初归属于不同解释器的线程不会共享任何东西，但进程状态如可用内存、打开的文件描述符等等除外。 全局解释器锁也会被所有线程共享，无论它们归属于哪个解释器。

> *在 3.12 版本发生变更:* **PEP 684** 引入了 单解释器 GIL 的可能性。请参阅 Py_NewInterpreterFromConfig()。

type **PyThreadState**

属于 受限 API （作为不透明的结构体）.

该数据结构代表单个线程的状态。 唯一的公有数据成员为：

PyInterpreterState ***interp**

该线程的解释器状态。

void **PyEval_InitThreads**()

属于 稳定 ABI.

不执行任何操作的已弃用函数。

在 Python 3.6 及更老的版本中，此函数会在 GIL 不存在时创建它。

> *在 3.9 版本发生变更:* 此函数现在不执行任何操作。

> *在 3.7 版本发生变更:* 该函数现在由 `Py_Initialize()` 调用，因此你无需再自行调用它。

> *在 3.2 版本发生变更:* 此函数已不再被允许在 `Py_Initialize()` 之前调用。

> *自 3.9 版本弃用.*

`PyThreadState` ***PyEval_SaveThread**()*

属于 稳定 ABI.

分离当前线程的 attached thread state 并返回该状态对象。调 用此函数返回后，当前线程将不再关联任何 thread state。

`void` **PyEval_RestoreThread**(`PyThreadState` *tstate)

属于 稳定 ABI.

将 attached thread state 设置为 *tstate*。传入的 thread state **不应** 处于 已附加 状态，否则会导致死锁。 调用此函数返回后，*tstate* 将被附加到当前线程。

> **备注:** 当运行时处于终结阶段时，若从某个线程调用此函数，该线程将被挂起直至程序退出，即便是由非 Python 创建的线程也不例外。更多详情请参考 有关运行时最终化的注意事项。

> *在 3.14 版本发生变更:* 如果在解释器处于终结阶段时调用此函数，当前线程将被挂起而非终止。

`PyThreadState` ***PyThreadState_Get**()*

属于 稳定 ABI.

返回当前线程的 attached thread state。 如果线程没有已附加的线程状态（例如，当处于 `Py_BEGIN_ALLOW_THREADS` 代码块内部时），则会触发致命错误 (因此调用者无需检查返回值是否为 `NULL`)。

另请参阅 `PyThreadState_GetUnchecked()`。

`PyThreadState` ***PyThreadState_GetUnchecked**()*

与 `PyThreadState_Get()` 类似，但如果其为 NULL 则不会杀死进程并设置致命错误。 调用方要负责检查结果是否为 NULL。

> *Added in version 3.13:* 在 Python 3.5 到 3.12 中，此函数是私有的并且命名为 `_PyThreadState_UncheckedGet()`。

`PyThreadState` ***PyThreadState_Swap**(`PyThreadState` *tstate)*

属于 稳定 ABI.

将 attached thread state 设置为 *tstate*，并返回调用此函数前已附加的 thread state。

此函数在没有 attached thread state 的情况下调用也是安全的；此时它会直接返回 `NULL`，表示之前不存在线程状态。

参见： `PyEval_ReleaseThread()`

备注： 与 `PyGILState_Ensure()` 类似，当运行时处于终结阶段时，调用此函数会导致线程挂起。

下列函数使用线程级本地存储，并且不能兼容子解释器：

type **PyGILState_STATE**

属于 稳定 ABI.

由 `PyGILState_Ensure()` 返回并传递给 `PyGILState_Release()` 的值的类型。

enumerator **PyGILState_LOCKED**

当调用 `PyGILState_Ensure()` 时 GIL 已经被持有。

enumerator **PyGILState_UNLOCKED**

当调用 `PyGILState_Ensure()` 时 GIL 尚未被持有。

`PyGILState_STATE` **PyGILState_Ensure**()

属于 稳定 ABI.

确保当前线程可以调用 Python C API，无论 Python 的当前状态或 attached thread state 如何。只要每个调用都与对 `PyGILState_Release()` 的调用相匹配，线程就可以根据需要多次调用此函数。通常，只要线程状态在调用 Release() 之前恢复到其先前状态，就可以在 `PyGILState_Ensure()` 和 `PyGILState_Release()` 调用之间使用其他与线程相关的 API。例如，可以正常使用 `Py_BEGIN_ALLOW_THREADS` 和 `Py_END_ALLOW_THREADS` 宏。

返回值是一个不透明的"句柄"，指向调用 `PyGILState_Ensure()` 时的 attached thread state，必须将其传递给 `PyGILState_Release()` 以确保 Python 恢复到相同状态。尽管允许递归调用，但这些句柄 *不能* 共享 — 每次对 `PyGILState_Ensure()` 的独立调用都必须保存其对应的句柄，用于后续调用 `PyGILState_Release()`。

当此函数返回时，将存在一个 attached thread state，并且线程将能够调用任意 Python 代码。若操作失败则会引发致命错误。

警告： 当运行时处于终结阶段时调用此函数是不安全的。这样做要么会使线程挂起直至程序结束，在极少数情况下还可能导致解释器完全崩溃。更多详情请参考 有关运行时最终化的注意事项。

> *在 3.14 版本发生变更:* 如果在解释器处于终结阶段时调用此函数，当前线程将被挂起而非终止。

`void` **PyGILState_Release**(PyGILState_STATE)

属于 稳定 ABI.

释放之前获取的任何资源。 在此调用之后，Python 的状态将与其在对相应 `PyGILState_Ensure()` 调用之前的一样（但是通常此状态对调用方来说将是未知的，对 GILState API 的使用也是如此）。

对 `PyGILState_Ensure()` 的每次调用都必须与在同一线程上对 `PyGILState_Release()` 的调用相匹配。

PyThreadState *__PyGILState_GetThisThreadState__()

属于 稳定 ABI.

获取当前线程的 attached thread state。如果当前线程尚未使用任何 GILState API，则可能返回 `NULL`。 请注意，主线程始终拥有这样的线程状态，即使尚未在主线程上进行任何自动线程状态调用。此函数主要用作辅助/诊断工具。

备注: This function may return non-`NULL` even when the thread state is detached. Prefer `PyThreadState_Get()` or `PyThreadState_GetUnchecked()` for most cases.

参见: `PyThreadState_Get()`

`int` **PyGILState_Check**()

如果当前线程持有 GIL 则返回 `1`，否则返回 `0`。此函数可随时从任何线程调用。只有当线程的线程状态 通过 `PyGILState_Ensure()` 初始化后，它才会返回 `1`。此函数主要用作辅助/诊断工具。例如，在回调函数上下文或内存分配函数中，了解 GIL 是否被锁定可以让调用者执行敏感操作或以不同方式运行时，这个函数就会很有用。

备注: 如果当前 Python 进程曾经创建过子解释器，则此函数 *始终* 返回 `1`。在大多数情况下，建议使用 `PyThreadState_GetUnchecked()`。

Added in version 3.4.

以下的宏被使用时通常不带末尾分号；请在 Python 源代码发布包中查看示例用法。

Py_BEGIN_ALLOW_THREADS

属于 稳定 ABI.

此宏会扩展为 `{ PyThreadState *_save; _save = PyEval_SaveThread();`。 请注意它包含一个开头花括号；它必须与后面的 `Py_END_ALLOW_THREADS` 宏匹配。 有关此宏的进一步讨论请参阅上文。

Py_END_ALLOW_THREADS

属于 稳定 ABI.

此宏扩展为 `PyEval_RestoreThread(_save); }`。 注意它包含一个右花括号；它必须与之前的 Py_BEGIN_ALLOW_THREADS 宏匹配。 请参阅上文以进一步讨论此宏。

Py_BLOCK_THREADS

属于 稳定 ABI.

这个宏扩展为 `PyEval_RestoreThread(_save);`: 它等价于没有关闭花括号的 Py_END_ALLOW_THREADS。

Py_UNBLOCK_THREADS

属于 稳定 ABI.

这个宏扩展为 `_save = PyEval_SaveThread();`: 它等价于没有开始花括号和变量声明的 Py_BEGIN_ALLOW_THREADS。

底层级 API

下列所有函数都必须在 Py_Initialize() 之后被调用。

> *在 3.7 版本发生变更:* 现在 Py_Initialize() 会初始化 GIL 并设置一个 attached thread state。

PyInterpreterState ***PyInterpreterState_New**()

属于 稳定 ABI.

新建一个解释器状态对象。 不需要有 attached thread state，但如果有必要序列化对此函数的调用则可能选择有。

引发一个不带参数的 审计事件 `cpython.PyInterpreterState_New`。

void **PyInterpreterState_Clear**(PyInterpreterState *interp)

属于 稳定 ABI.

重置解释器状态对象中的所有信息。解释器必须存在一个 attached thread state。

引发一个不带参数的 审计事件 `cpython.PyInterpreterState_Clear`。

void **PyInterpreterState_Delete**(PyInterpreterState *interp)

属于 稳定 ABI.

销毁一个解释器状态对象。目标解释器 **不应** 存在 attached thread state。 在调用此函数之前，必须先调用 PyInterpreterState_Clear() 重置解释器状态。

PyThreadState ***PyThreadState_New**(PyInterpreterState *interp)

属于 稳定 ABI.

创建一个属于指定解释器对象的新线程状态对象。此操作不需要存在 attached thread state。

void **PyThreadState_Clear**(PyThreadState *tstate)

属于 稳定 ABI.

重置 thread state 对象中的所有信息。 *tstate* 必须处于 已附加 状态。

> *在 3.9 版本发生变更:* This function now calls the `PyThreadState.on_delete` callback. Previously, that happened in `PyThreadState_Delete()`.

> *在 3.13 版本发生变更:* `PyThreadState.on_delete` 回调已被移除。

void **PyThreadState_Delete**(PyThreadState *tstate)

属于 稳定 ABI.

销毁一个 thread state 对象。*tstate* 不应被 已附加 到任何线程。*tstate* 必须在之前通过调用 `PyThreadState_Clear()` 进行过重置。

void **PyThreadState_DeleteCurrent**(void)

分离 attached thread state (该状态必须已通过先前调用 `PyThreadState_Clear()` 进行重置)，然后销毁它。

返回时将不会有任何 thread state 处于 已附加 状态。

PyFrameObject ***PyThreadState_GetFrame**(PyThreadState *tstate)

属于 稳定 ABI 自 3.10 版起.

获取 Python 线程状态 *tstate* 的当前帧。

返回一个 strong reference。 如果没有当前执行的帧则返回 `NULL`。

另请参阅 `PyEval_GetFrame()`。

tstate 不得为 `NULL`，并且必须处于 已附加 状态。

> *Added in version 3.9.*

uint64_t **PyThreadState_GetID**(PyThreadState *tstate)

属于 稳定 ABI 自 3.10 版起.

获取 Python 线程状态 *tstate* 的唯一 thread state 标识符。

tstate 不得为 `NULL`，并且必须处于 已附加 状态。

> *Added in version 3.9.*

PyInterpreterState ***PyThreadState_GetInterpreter**(PyThreadState *tstate)

属于 稳定 ABI 自 3.10 版起.

获取 Python 线程状态 *tstate* 对应的解释器。

tstate 不得为 `NULL`，并且必须处于 已附加 状态。

> *Added in version 3.9.*

`void` **PyThreadState_EnterTracing**(PyThreadState *tstate)

暂停 Python 线程状态 *tstate* 中的追踪和性能分析。

使用 PyThreadState_LeaveTracing() 函数来恢复它们。

> *Added in version 3.11.*

`void` **PyThreadState_LeaveTracing**(PyThreadState *tstate)

恢复 Python 线程状态 *tstate* 中被 PyThreadState_EnterTracing() 函数暂停的追踪和性能分析。

另请参阅 PyEval_SetTrace() 和 PyEval_SetProfile() 函数。

> *Added in version 3.11.*

PyInterpreterState *__PyInterpreterState_Get__(`void`)

属于 稳定 ABI 自 3.9 版起.

获取当前解释器。

如果不存在 attached thread state，则触发致命错误。此函数不会返回 NULL。

> *Added in version 3.9.*

`int64_t` **PyInterpreterState_GetID**(PyInterpreterState *interp)

属于 稳定 ABI 自 3.7 版起.

返回解释器的唯一 ID。 如果执行过程中发生任何错误则将返回 `-1` 并设置错误。

调用方必须有已附加的线程状态 attached thread state。

> *Added in version 3.7.*

PyObject *__PyInterpreterState_GetDict__(PyInterpreterState *interp)

返回值：借入的引用。 属于 稳定 ABI 自 3.8 版起.

返回一个存储解释器专属数据的字典。 如果此函数返回 `NULL` 则没有任何异常被引发并且调用方应当将解释器专属字典视为不可用。

这不是 PyModule_GetState() 的替代，扩展仍应使用它来存储解释器专属的状态信息。

返回的字典是从解释器借入的并将保持可用直到解释器关闭。

> *Added in version 3.8.*

`typedef` PyObject *(*__PyFrameEvalFunction__)(PyThreadState *tstate, _PyInterpreterFrame *frame, `int` throwflag)

帧评估函数的类型

throwflag 形参将由生成器的 `throw()` 方法来使用：如为非零值，则处理当前异常。

> *在 3.9 版本发生变更:* 此函数现在可接受一个 *tstate* 形参。

> *在 3.11 版本发生变更:* *frame* 形参由 `PyFrameObject*` 改为 `_PyInterpreterFrame*`。

`_PyFrameEvalFunction`
_PyInterpreterState_GetEvalFrameFunc(`PyInterpreterState` *interp)
获取帧评估函数。

请参阅 **PEP 523** "Adding a frame evaluation API to CPython"。

> *Added in version 3.9.*

`void` **_PyInterpreterState_SetEvalFrameFunc**(`PyInterpreterState` *interp, `_PyFrameEvalFunction` eval_frame)
设置帧评估函数。

请参阅 **PEP 523** "Adding a frame evaluation API to CPython"。

> *Added in version 3.9.*

`PyObject` ***PyThreadState_GetDict**()
返回值：借入的引用。 属于 稳定 ABI.

返回一个字典，扩展模块可在其中存储线程特定的状态信息。 每个扩展模块应使用唯一的键来在此字典中存储状态。当没有 thread state 处于 已附加 状态时，调用此函数是安全的。 如果此函数返回 `NULL`，则表示没有引发异常，调用者应假定没有线程状态被附加。

`int` **PyThreadState_SetAsyncExc**(`unsigned long` id, `PyObject` *exc)
属于 稳定 ABI.

在一个线程中异步引发异常。*id* 参数是目标线程的线程 ID；*exc* 是要引发的异常对象。此函数不会窃取对 *exc* 的引用。 为防止误用，必须编写自己的 C 扩展来调用此函数。必须在 attached thread state 下调用此函数。 返回被修改的线程状态数量；通常为 1，如果找不到线程 ID 则返回 0。如果 *exc* 为 `NULL`，则清除该线程的待处理异常（如果有）。 此函数不会引发任何异常。

> *在 3.7 版本发生变更:* *id* 形参的类型已从 `long` 变为 `unsigned long`。

`void` **PyEval_AcquireThread**(`PyThreadState` *tstate)
属于 稳定 ABI.

将 *tstate* 附加 到当前线程，当前线程不能为 `NULL` 或者已经 附加线程状态。

调用方线程不能已经具有 attached thread state。

> **备注:** 当运行时处于终结阶段时，若从某个线程调用此函数，该线程将被挂起直至程序退出，即便是由非 Python 创建的线程也不例外。更多详情请参考 有关运行时最终化的注意事项。

> *在 3.8 版本发生变更:* 已被更新为与 `PyEval_RestoreThread()`，`Py_END_ALLOW_THREADS()` 和 `PyGILState_Ensure()` 保持一致，如果在解释器正在最终化时被调用则会终结当前线程。

> *在 3.14 版本发生变更:* 如果在解释器处于终结阶段时调用此函数，当前线程将被挂起而非终止。

`PyEval_RestoreThread()` 是一个始终可用的（即使线程尚未初始化）更高层级函数。

`void` **PyEval_ReleaseThread**(`PyThreadState` *tstate)

属于 稳定 ABI.

分离 attached thread state。 *tstate* 参数必须不为 `NULL`，该参数仅被用于检查它是否代表 attached thread state --- 如果不是，则会报告一个致命级错误。

`PyEval_SaveThread()` 是一个始终可用的（即使线程尚未初始化）更高层级函数。

子解释器支持

虽然在大多数用例中，你都只会嵌入一个单独的 Python 解释器，但某些场景需要你在同一个进程甚至同一个线程中创建多个独立的解释器。 子解释器让你能够做到这一点。

“主”解释器是在运行时初始化时创建的第一个解释器。 它通常是一个进程中唯一的 Python 解释器。与子解释器不同，主解释器具有唯一的进程全局责任比如信号处理等。 它还负责在运行时初始化期间的执行并且通常还是运行时最终化期间的活动解释器。 `PyInterpreterState_Main()` 函数将返回一个指向其状态的指针。

你可以使用 `PyThreadState_Swap()` 函数在子解释器之间进行切换。 你可以使用下列函数来创建和销毁它们:

`type` **PyInterpreterConfig**

包含用于配置子解释器的大部分形参的结构体。 其值仅在 `Py_NewInterpreterFromConfig()` 中被使用而绝不会被运行时所修改。

> *Added in version 3.12.*

结构体字段:

`int` **use_main_obmalloc**

如果该值为 `0` 则子解释器将使用自己的“对象”分配器状态。 否则它将使用（共享）主解释器的状态。

如果该值为 `0` 则 `check_multi_interp_extensions` 必须为 `1` (非零值)。 如果该值为 `1` 则 `gil` 不可为 `PyInterpreterConfig_OWN_GIL`。

`int` **allow_fork**

如果该值为 `0` 则运行时将不支持在当前激活了子解释器的任何线程中 fork 进程。 否则 fork 将不受限制。

请注意当 fork 被禁止时 subprocess 模块将仍然可用。

int **allow_exec**

如果该值为 0 则运行时将不支持在当前激活了子解释器的任何线程中通过 exec (例如 os.execv()) 替换当前进程。 否则 exec 将不受限制。

请注意当 exec 被禁止时 subprocess 模块将仍然可用。

int **allow_threads**

如果该值为 0 则子解释器的 threading 模块将不会创建线程。 否则线程将被允许。

int **allow_daemon_threads**

如果该值为 0 则子解释器的 threading 模块将不会创建守护线程。 否则将允许守护线程（只要 allow_threads 是非零值）。

int **check_multi_interp_extensions**

如果该值为 0 则所有扩展模块均可在当前子解释器被激活的任何线程中被导入，包括旧式的 (单阶段初始化) 模块。 否则将只有多阶段初始化扩展模块 (参见 **PEP 489**) 可以被导入。 (另请参阅 Py_mod_multiple_interpreters。)

如果 use_main_obmalloc 为 0 则该值必须为 1 (非零值)。

int **gil**

这将确定针对子解释器的 GIL 操作方式。 它可以是以下的几种之一：

PyInterpreterConfig_DEFAULT_GIL

使用默认选择 (PyInterpreterConfig_SHARED_GIL)。

PyInterpreterConfig_SHARED_GIL

使用（共享）主解释器的 GIL。

PyInterpreterConfig_OWN_GIL

使用子解释器自己的 GIL。

如果该值为 PyInterpreterConfig_OWN_GIL 则 PyInterpreterConfig.use_main_obmalloc 必须为 0。

PyStatus **Py_NewInterpreterFromConfig**(PyThreadState **tstate_p, const PyInterpreterConfig *config)

新建一个子解释器。 这是一个 (几乎) 完全隔离的 Python 代码执行环境。 特别需要注意，新的子解释器具有全部已导入模块的隔离的、独立的版本，包括基本模块 builtins, __main__ 和 sys 等。 已加载模块表 (sys.modules) 和模块搜索路径 (sys.path) 也是隔离的。 新环境没有 sys.argv 变量。 它具有新的标准 I/O 流文件对象 sys.stdin, sys.stdout 和 sys.stderr (不过这些对象都指向相同的底层文件描述符)。

给定的 *config* 控制着初始化解释器所使用的选项。

成功后，*tstate_p* 将被设为新的子解释器中创建的第一个 thread state。 该线程状态是 已附加的。 请注意并没有真实的线程被创建；请参阅下文有关线程状态的讨论。 如果新解释器的创建没有成功，则 *tstate_p* 将被设为 `NULL`；不会设置任何异常因为异常状态是存储在 attached thread state 中的，而它并不一定存在。

与所有其他 Python/C API 函数一样，调用此函数前必须存在 attached thread state，但返回时该状态可能会被分离。 成功时，返回的线程状态将处于 已附加 状态。 如果子解释器使用自己的 GIL 创建，则调用解释器的 attached thread state 将被分离。 当函数返回时，新解释器的 thread state 将 已附加 到当前线程，而先前解释器的 attached thread state 将保持分离状态。

> *Added in version 3.12.*

子解释器在彼此相互隔离，并让特定功能受限的情况下是最有效率的:

```
PyInterpreterConfig config = {
    .use_main_obmalloc = 0,
    .allow_fork = 0,
    .allow_exec = 0,
    .allow_threads = 1,
    .allow_daemon_threads = 0,
    .check_multi_interp_extensions = 1,
    .gil = PyInterpreterConfig_OWN_GIL,
};
PyThreadState *tstate = NULL;
PyStatus status = Py_NewInterpreterFromConfig(&tstate, &config);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}
```

请注意该配置只会被短暂使用而不会被修改。 在初始化期间配置的值会被转换成各种 `PyInterpreterState` 值。 配置的只读副本可以被内部存储于 `PyInterpreterState` 中。

扩展模块将以如下方式在（子）解释器之间共享：

- 对于使用多阶段初始化的模块，例如 `PyModule_FromDefAndSpec()`，将为每个解释器创建并初始化一个单独的模块对象。 只有 C 层级的静态和全局变量能在这些模块 对象之间共享。

- 对于使用单阶段初始化的模块，例如 `PyModule_Create()`，当特定扩展被首次导入时，它将被正常初始化，并会保存其模块字典的一个 (浅) 拷贝。 当同一扩展被另一个 (子) 解释器导入时，将初始化一个新模块并填充该拷贝的内容；扩展的 `init` 函数不会被调用。 因此模块字典中的对象最终会被 (子) 解释器所共享，这可能会导致预期之外的行为 (参见下文的 Bugs and caveats)。

  请注意这不同于在调用 `Py_FinalizeEx()` 和 `Py_Initialize()` 完全重新初始化解释器之后导入扩展时所发生的情况；对于那种情况，扩展的 `initmodule` 函数 *会被* 再次调用。 与多阶段初始化一样，这意味着只有 C 层级的静态和全局变量能在这些模块之间共享。

`PyThreadState` ***Py_NewInterpreter**(`void`)

属于 稳定 ABI.

新建一个子解释器。 这在本质上只是针对 Py_NewInterpreterFromConfig() 的包装器，其配置保留了现有的行为。 结果是一个未隔离的子解释器，它会共享主解释器的 GIL，允许 fork/exec，允许守护线程，也允许单阶段初始化模块。

void **Py_EndInterpreter**(PyThreadState *tstate)

属于 稳定 ABI.

销毁由给定 thread state 表示的（子）解释器。给定的线程状态必须处于 已附加 状态。调用返回时，将不存在任何 attached thread state。 与此解释器关联的所有线程状态都会被销毁。

Py_FinalizeEx() 将销毁所有在当前时间点上尚未被明确销毁的子解释器。

解释器级 GIL

使用 Py_NewInterpreterFromConfig() 你将可以创建一个与其他解释器完全隔离的子解释器，包括具有自己的 GIL。 这种隔离带来的最大好处在于这样的解释器执行 Python 代码时不会被其他解释器所阻塞或者阻塞任何其他解释器。 因此在运行 Python 代码时单个 Python 进程可以真正地利用多个 CPU 核心。 这种隔离还能鼓励开发者采取不同于仅使用线程的并发方式。 (参见 **PEP 554** 和 **PEP 684**)。

使用隔离的解释器要求谨慎地保持隔离状态。 尤其是意味着不要在未确保线程安全的情况下共享任何对象或可变的状态。 由于引用计数的存在即使是在其他情况下不可变的对象 (例如 None, (1, 5)) 通常也不可被共享。 针对此问题的一种简单但效率较低的解决方式是在使用某些状态 (或对象) 时总是使用一个全局锁。 或者，对象实际上不可变的对象 (如整数或字符串) 可以通过将其设为 immortal 对象而无视其引用计数来确保其安全性。 事实上，对于内置单例、小整数和其他一些内置对象都是这样做的。

如果你能保持隔离状态那么你将能获得真正的多核计算能力而不会遇到自由线程所带来的复杂性。 如果未能保持隔离状态那么你将面对自由线程所带来的全部后果，包括线程竞争和难以调试的崩溃。

除此之外，使用多个相互隔离的解释器的一个主要挑战是如何在它们之间安全 (不破坏隔离状态)、高效地进行通信。 运行时和标准库还没有为此提供任何标准方式。 未来的标准库模块将会帮助减少保持隔离状态所需的工作量并为解释器之间的数据通信（和共享）公开有效的工具。

> *Added in version 3.12.*

错误和警告

由于子解释器 (以及主解释器) 都是同一个进程的组成部分，它们之间的隔离状态并非完美 --- 举例来说，使用低层级的文件操作如 os.close() 时它们可能 (无意或恶意地) 影响它们各自打开的文件。 由于 (子) 解释器之间共享扩展的方式，某些扩展可能无法正常工作；在使用单阶段初始化或者 (静态) 全局变量时尤其如此。 在一个子解释器中创建的对象有可能被插入到另一个 (子) 解释器的命名空间中；这种情况应当尽可能地避免。

应当特别注意避免在子解释器之间共享用户自定义的函数、方法、实例或类，因为由这些对象执行的导入 操作可能会影响错误的已加载模块的 (子) 解释器的字典。 同样重要的一点是应当避免共享可

被上述对象访问的对象 。

还要注意的一点是将此功能与 `PyGILState_*` API 结合使用是很微妙的，因为这些 API 会假定 Python线程状态与操作系统级线程之间存在双向投影关系，而子解释器的存在打破了这一假定。 强烈建议你不要在一对互相匹配的 `PyGILState_Ensure()` 和 `PyGILState_Release()` 调用之间切换子解释器。 此外，使用这些 API 以允许从非 Python 创建的线程调用 Python 代码的扩展 (如 `ctypes`) 在使用子解释器时很可能会出现问题。

异步通知

提供了一种向主解释器线程发送异步通知的机制。 这些通知将采用函数指针和空指针参数的形式。

```
int Py_AddPendingCall(int (*func)(void*), void *arg)
```

属于 稳定 ABI.

将一个函数加入从主解释器线程调用的计划任务。 成功时，将返回 `0` 并将 *func* 加入要被主线程调用的等待队列。 失败时，将返回 `-1` 但不会设置任何异常。

当成功加入队列后，*func* 将 *最终* 附带参数 *arg* 被主解释器线程调用。 对于正常运行的 Python 代码来说它将被异步地调用，但要同时满足以下两个条件：

- 位于 bytecode 的边界上；
- 主线程持有一个 attached thread state (因此 *func* 可以使用完整的 C API)。

func 必须在成功时返回 `0`，或在失败时返回 `-1` 并设置一个异常集合。 *func* 不会被中断来递归地执行另一个异步通知，但如果 thread state 已被分离则它仍可被中断以切换线程。

此函数不需要 attached thread state。 不过，要在子解释器中调用此函数，调用方必须具有 attached thread state。 否则，函数 *func* 可能会被安排给错误的解释器来调用。

警告: 这是一个低层级函数，只在非常特殊的情况下有用。 不能保证 *func* 会尽快被调用。 如果主线程忙于执行某个系统调用，*func* 将不会在系统调用返回之前被调用。 此函数 通常 **不适合** 从任意 C 线程调用 Python 代码。 作为替代，请使用 PyGILStateAPI。

> *Added in version 3.1.*

> *在 3.9 版本发生变更:* 如果此函数在子解释器中被调用，则函数 *func* 将被安排在子解释器中调用，而不是在主解释器中调用。现在每个子解释器都有自己的计划调用列表。

> *在 3.12 版本发生变更:* 此函数现在总是会安排 *func* 在主解释器中运行。to be run in the main interpreter.

分析和跟踪

Python 解释器为附加的性能分析和执行跟踪工具提供了一些低层级的支持。 它们可被用于性能分析、调试和覆盖分析工具。

这个 C 接口允许性能分析或跟踪代码避免调用 Python 层级的可调用对象带来的开销，它能直接执行 C 函数调用。 此工具的基本属性没有变化；这个接口允许针对每个线程安装跟踪函数，并且向跟踪函数报告的基本事件与之前版本中向 Python 层级跟踪函数报告的事件相同。

typedef int (*Py_tracefunc)(PyObject *obj, PyFrameObject *frame, int what, PyObject *arg)

使用 PyEval_SetProfile() 和 PyEval_SetTrace() 注册的跟踪函数的类型。 第一个形参是作为 *obj* 传递给注册函数的对象，*frame* 是与事件相关的帧对象，*what* 是常量 PyTrace_CALL, PyTrace_EXCEPTION, PyTrace_LINE, PyTrace_RETURN, PyTrace_C_CALL, PyTrace_C_EXCEPTION, PyTrace_C_RETURN 或 PyTrace_OPCODE 中的一个，而 *arg* 将依赖于 *what* 的值：

***what* 的值**	***arg* 的含义**
PyTrace_CALL	总是 Py_None.
PyTrace_EXCEPTION	sys.exc_info() 返回的异常信息。
PyTrace_LINE	总是 Py_None.
PyTrace_RETURN	返回给调用方的值，或者如果是由异常导致的则返回 NULL。
PyTrace_C_CALL	正在调用函数对象。
PyTrace_C_EXCEPTION	正在调用函数对象。
PyTrace_C_RETURN	正在调用函数对象。
PyTrace_OPCODE	总是 Py_None.

int **PyTrace_CALL**

当对一个函数或方法的新调用被报告，或是向一个生成器增加新条目时传给 Py_tracefunc 函数的 *what* 形参的值。 请注意针对生成器函数的迭代器的创建情况不会被报告因为在相应的帧中没有向 Python字节码转移控制权。

int **PyTrace_EXCEPTION**

当一个异常被引发时传给 Py_tracefunc 函数的 *what* 形参的值。 在处理完任何字节码之后将附带 *what* 的值调用回调函数，在此之后该异常将会被设置在正在执行的帧中。 这样做的效果是当异常传播导致 Python 栈展开时，被调用的回调函数将随异常传播返回到每个帧。 只有跟踪函数才会接收到这些事件；性能分析器并不需要它们。

int **PyTrace_LINE**

当一个行编号事件被报告时传给 Py_tracefunc 函数 (但不会传给性能分析函数) 的 *what* 形参的值。 它可以通过将 f_trace_lines 设为 *0* 在某个帧中被禁用。

int **PyTrace_RETURN**

当一个调用即将返回时传给 Py_tracefunc 函数的 *what* 形参的值。

int **PyTrace_C_CALL**

当一个 C 函数即将被调用时传给 Py_tracefunc 函数的 *what* 形参的值。

int **PyTrace_C_EXCEPTION**
当一个 C 函数引发异常时传给 Py_tracefunc 函数的 *what* 形参的值。

int **PyTrace_C_RETURN**
当一个 C 函数返回时传给 Py_tracefunc 函数的 *what* 形参的值。

int **PyTrace_OPCODE**
当一个新操作码即将被执行时传给 Py_tracefunc 函数 (但不会传给性能分析函数) 的 *what* 形参的值。 在默认情况下此事件不会被发送：它必须通过在某个帧上将 f_trace_opcodes 设为 *1* 来显式地请求。

void **PyEval_SetProfile**(Py_tracefunc func, PyObject *obj)
将性能分析器函数设为 *func*。 *obj* 形参将作为第一个形参传给该函数，它可以是任意 Python 对象或为 NULL。 如果性能分析函数需要维护状态，则为每个线程的 *obj* 使用不同的值将提供一个方便而线程安全的存储位置。 这个性能分析函数将针对除 PyTrace_LINE PyTrace_OPCODE 和 PyTrace_EXCEPTION 以外的所有被监控事件进行调用。

另请参阅 sys.setprofile() 函数。

调用方必须有已附加的线程状态 attached thread state。

void **PyEval_SetProfileAllThreads**(Py_tracefunc func, PyObject *obj)
类似于 PyEval_SetProfile() 但会在属于当前解释器的所有在运行线程中设置性能分析函数而不是仅在当前线程上设置。

调用方必须有已附加的线程状态 attached thread state。

与 PyEval_SetProfile() 一样，该函数会忽略任何被引发的异常同时在所有线程中设置性能分析函数。

> *Added in version 3.12.*

void **PyEval_SetTrace**(Py_tracefunc func, PyObject *obj)
将跟踪函数设为 *func*。 这类似于 PyEval_SetProfile()，区别在于跟踪函数会接收行编号事件和操作码级事件，但不会接收与被调用的 C 函数对象相关的任何事件。 使用 PyEval_SetTrace() 注册的任何跟踪函数将不会接收 PyTrace_C_CALL、PyTrace_C_EXCEPTION 或 PyTrace_C_RETURN 作为 *what* 形参的值。

另请参阅 sys.settrace() 函数。

调用方必须有已附加的线程状态 attached thread state。

void **PyEval_SetTraceAllThreads**(Py_tracefunc func, PyObject *obj)
类似于 PyEval_SetTrace() 但会在属于当前解释器的所有在运行线程中设置跟踪函数而不是仅在当前线程上设置。

调用方必须有已附加的线程状态 attached thread state。

与 PyEval_SetTrace() 一样，该函数会忽略任何被引发的异常同时在所有线程中设置跟踪函数。

Added in version 3.12.

引用追踪

Added in version 3.13.

typedef int (*PyRefTracer)(PyObject*, int event, void *data)

使用 PyRefTracer_SetTracer() 注册的追踪函数的类型。 第一个形参是刚创建（当 **event** 被设为 PyRefTracer_CREATE 时）或将销毁（当 **event** 被设为 PyRefTracer_DESTROY 时）的 Python 对象。 **data** 参数是当 PyRefTracer_SetTracer() 被调用时所提供的不透明指针。

Added in version 3.13.

int **PyRefTracer_CREATE**

当一个 Python 对象被创建时传给 PyRefTracer 函数的 *event* 形参。

int **PyRefTracer_DESTROY**

当一个 Python 对象被销毁时传给 PyRefTracer 函数的 *event* 形参。

int **PyRefTracer_SetTracer**(PyRefTracer tracer, void *data)

注册一个引用追踪函数。 该函数将在新的 Python 对象被创建或对象被销毁时被调用。 如果提供了 **data** 则它必须是一个当追踪函数被调用时所提供的不透明指针。 成功时返回 `0`。 发生错误时将设置一个异常并返回 `-1`。

追踪函数 **不可** 在内部创建 Python 对象否则调用将被重入。 追踪器也 **不可** 清除任何现有异常或设置异常。 每次当追踪器被调用时都将激活一个 thread state。

当调用此函数时必须有一个 attached thread state。

Added in version 3.13.

PyRefTracer **PyRefTracer_GetTracer**(void **data)

获取已注册的引用追踪函数以及当 PyRefTracer_SetTracer() 被调用时所注册的不透明数据指针的值。 如果未注册任何追踪器则此函数将返回 NULL 并将 **data** 指针设为 NULL。

当调用此函数时必须有一个 attached thread state。

Added in version 3.13.

高级调试器支持

这些函数仅供高级调试工具使用。

PyInterpreterState ***PyInterpreterState_Head**()

将解释器状态对象返回到由所有此类对象组成的列表的开头。

PyInterpreterState *PyInterpreterState_Main()
返回主解释器状态对象。

PyInterpreterState *PyInterpreterState_Next(PyInterpreterState *interp)
从由解释器状态对象组成的列表中返回 *interp* 之后的下一项。

PyThreadState *PyInterpreterState_ThreadHead(PyInterpreterState *interp)
在由与解释器 *interp* 相关联的线程组成的列表中返回指向第一个 PyThreadState 对象的指针。

PyThreadState *PyThreadState_Next(PyThreadState *tstate)
从由属于同一个 PyInterpreterState 对象的线程状态对象组成的列表中返回 *tstate* 之后的下一项。

线程本地存储支持

Python 解释器提供也对线程本地存储 (TLS) 的低层级支持，它对下层的原生 TLS 实现进行了包装以支持 Python 层级的线程本地存储 API (`threading.local`)。 CPython 的 C 层级 API 与 pthreads 和 Windows 所提供的类似：使用一个线程键和函数来为每个线程关联一个 `void*` 值。

当调用这些函数时 *不需要* 附加 一个 thread state；它们会提供自己的锁机制。

请注意 `Python.h` 并不包括 TLS API 的声明，你需要包括 `pythread.h` 来使用线程本地存储。

备注： 这些 API 函数都不会为 `void*` 的值处理内存管理问题。 你需要自己分配和释放它们。 如果 `void*` 值碰巧为 PyObject*，这些函数也不会对它们执行引用计数操作。

线程专属存储 (TSS) API

引入 TSSAPI 是为了取代 CPython 解释器中现有 TLS API 的使用。 该 API 使用一个新类型 Py_tss_t 而不是 `int` 来表示线程键。

> *Added in version 3.7.*

参见： "A New C-API for Thread-Local Storage in CPython" (**PEP 539**)

type **Py_tss_t**
该数据结构表示线程键的状态，其定义可能依赖于下层的 TLS 实现，并且它有一个表示键初始化状态的内部字段。 该结构体中不存在公有成员。

当未定义 Py_LIMITED_API 时，允许由 Py_tss_NEEDS_INIT 执行此类型的静态分配。

Py_tss_NEEDS_INIT
这个宏将扩展为 Py_tss_t 变量的初始化器。 请注意这个宏不会用 Py_LIMITED_API 来定义。

动态分配

Py_tss_t 的动态分配，在使用 Py_LIMITED_API 编译的扩展模块中是必须的，在这些模块由于此类型的实现在编译时是不透明的因此它不可能静态分配。

Py_tss_t *PyThread_tss_alloc()
属于 稳定 ABI 自 3.7 版起.

返回一个与使用 Py_tss_NEEDS_INIT 初始化的值的状态相同的值，或者当动态分配失败时则返回 NULL。

void PyThread_tss_free(Py_tss_t *key)
属于 稳定 ABI 自 3.7 版起.

在首次调用 PyThread_tss_delete() 以确保任何相关联的线程局部变量已被撤销赋值之后释放由 PyThread_tss_alloc() 所分配的给定的 *key*。 如果 *key* 参数为 NULL 则这将无任何操作。

> **备注：** 被释放的 key 将变成一个悬空指针。 你应当将 key 重置为 NULL。

方法

这些函数的形参 *key* 不可为 NULL。 并且，如果给定的 Py_tss_t 还未被 PyThread_tss_create() 初始化则 PyThread_tss_set() 和 PyThread_tss_get() 的行为将是未定义的。

int PyThread_tss_is_created(Py_tss_t *key)
属于 稳定 ABI 自 3.7 版起.

如果给定的 Py_tss_t 已通过has been initialized by PyThread_tss_create() 被初始化则返回一个非零值。

int PyThread_tss_create(Py_tss_t *key)
属于 稳定 ABI 自 3.7 版起.

当成功初始化一个 TSS 键时将返回零值。 如果 *key* 参数所指向的值未被 Py_tss_NEEDS_INIT 初始化则其行为是未定义的。 此函数可在相同的键上重复调用 -- 在已初始化的键上调用它将不执行任何操作并立即成功返回。

void PyThread_tss_delete(Py_tss_t *key)
属于 稳定 ABI 自 3.7 版起.

销毁一个 TSS 键以便在所有线程中遗忘与该键相关联的值，并将该键的初始化状态改为未初始化的。 已销毁的键可以通过 PyThread_tss_create() 再次被初始化。 此函数可以在同一个键上重复调用 -- 但在一个已被销毁的键上调用将是无效的。

int PyThread_tss_set(Py_tss_t *key, void *value)
属于 稳定 ABI 自 3.7 版起.

返回零值来表示成功将一个 void* 值与当前线程中的 TSS 键相关联。 每个线程都有一个从键到 void* 值的独立映射。

`void *PyThread_tss_get(Py_tss_t *key)`

属于 稳定 ABI 自 3.7 版起.

返回当前线程中与一个 TSS 键相关联的 `void*` 值。 如果当前线程中没有与该键相关联的值则返回 `NULL`。

线程本地存储 (TLS) API

> *自 3.7 版本弃用:* 此 API 已被 线程专属存储 (TSS) API 所取代。

备注: 这个 API 版本不支持原生 TLS 键采用无法被安全转换为 `int` 的的定义方式的平台。 在这样的平台上，`PyThread_create_key()` 将立即返回一个失败状态，并且其他 TLS 函数在这样的平台上也都无效。

由于上面提到的兼容性问题，不应在新代码中使用此版本的API。

`int PyThread_create_key()`

属于 稳定 ABI.

`void PyThread_delete_key(int key)`

属于 稳定 ABI.

`int PyThread_set_key_value(int key, void *value)`

属于 稳定 ABI.

`void *PyThread_get_key_value(int key)`

属于 稳定 ABI.

`void PyThread_delete_key_value(int key)`

属于 稳定 ABI.

`void PyThread_ReInitTLS()`

属于 稳定 ABI.

同步原语

C-API 提供了一个基本的互斥锁。

`type PyMutex`

一个互斥锁。 `PyMutex` 应当被初始化为零以代表未加锁状态。 例如:

```
PyMutex mutex = {0};
```

`PyMutex` 的实例不应被拷贝或移动。 `PyMutex` 的内容和地址都是有意义的，它必须在内存中保持一个固定的、可写的位置。

备注: `PyMutex` 目前占用一个字节，但这个大小应当被视为是不稳定的。 这个大小可能在未来的 Python 发布版中发生改变而不会设置弃用期。

> *Added in version 3.13.*

`void PyMutex_Lock(PyMutex *m)`
锁定互斥锁 *m*。 如果另一个线程已经锁定了它，调用方线程将会阻塞直到互斥锁被解锁。 在阻塞期间，如果线程存在 线程状态 则会临时释放它。

> *Added in version 3.13.*

`void PyMutex_Unlock(PyMutex *m)`
解锁互斥锁 *m*。 该互斥锁必须已被锁定 --- 否则，此函数将发生致命错误。

> *Added in version 3.13.*

`int PyMutex_IsLocked(PyMutex *m)`
若互斥锁 *m* 当前处于锁定状态，则返回非零值；否则返回零。

> **备注：** 此函数仅适用于断言和调试场景，请勿将其用于并发控制决策，因为锁状态可能在检查后立即发生变化。

> *Added in version 3.14.*

Python 关键节 API

此关键节 API 为 自由线程 CPython 的每对象锁之上提供了一个死锁避免层。 它们旨在替代对 global interpreter lock 的依赖，而在具有全局解释器锁的 Python 版本上将不做任何操作。

关键节被设计用于在 C-API 扩展中实现的自定义类型。 它们通常不应当被用于内置类型如 `list` 和 `dict` 因为它们的公有 C-API 已经在内部使用了关键节，一个显著的例外是 `PyDict_Next()`，它需要在外部获取关键节。

关键节是通过隐式地挂起活动关键节来避免死锁的，因此，它们并不提供传统锁如 `PyMutex` 所提供的那种独占访问。 当一个关键节启动时，将会获取对象的每对象锁。 如果关键节内部执行的代码调用了 C-API 函数那么它可以挂起关键节从而释放这个每对象锁，这样其他线程就可以获取同一个对象的每对象锁。

此外，还提供了接受 `PyMutex` 指针（而非 Python 对象）的函数变体。当你处于没有 `PyObject` 的场景中时（例如，处理一个既未继承也未封装 `PyObject` 的 C 类型，但仍需以可能导致死锁的方式调用 C API），请使用这些变体来启动临界区。

宏所使用的函数和结构体是针对 C 宏不可用的场景而公开的。 它们应当仅被用于给定的宏扩展中。请注意这些结构体的大小和内容在未来的 Python 版本中可能发生改变。

> **备注：** 需要同时锁定两个对象的操作必须使用 `Py_BEGIN_CRITICAL_SECTION2`。 你 *不可* 使用嵌套的关键节来同时锁定一个以上的对象，因为内层的关键节可能会挂起外层的关键节。 这个 API 没有提供同时锁定两个以上对象的办法。

用法示例:

```
static PyObject *
set_field(MyObject *self, PyObject *value)
{
   Py_BEGIN_CRITICAL_SECTION(self);
   Py_SETREF(self->field, Py_XNewRef(value));
   Py_END_CRITICAL_SECTION();
   Py_RETURN_NONE;
}
```

在上面的例子中，Py_SETREF 调用了 Py_DECREF，它可以通过一个对象的取消分配函数来调用任意代码。 当由最终化器触发的代码发生阻塞并调用 PyEval_SaveThread() 时关键节 API 将通过允许运行临时挂起关键节来避免由于重入和锁顺序导致的潜在死锁。

Py_BEGIN_CRITICAL_SECTION(op)

为对象 *op* 获取每对象锁并开始一个关键节。

在自由线程构建版中，该宏将扩展为:

```
{
    PyCriticalSection _py_cs;
    PyCriticalSection_Begin(&_py_cs, (PyObject*)(op))
```

在默认构建版中，该宏将扩展为 `{`。

> *Added in version 3.13.*

Py_BEGIN_CRITICAL_SECTION_MUTEX(m)

锁定互斥锁 *m* 并开始一个临界区。

在自由线程构建版中，该宏将扩展为:

```
{
     PyCriticalSection _py_cs;
     PyCriticalSection_BeginMutex(&_py_cs, m)
```

需要注意的是，与 Py_BEGIN_CRITICAL_SECTION 不同，此宏的参数无需类型转换——它必须是一个 PyMutex 指针。

在默认构建版中，该宏将扩展为 `{`。

> *Added in version 3.14.*

Py_END_CRITICAL_SECTION()

结束关键节并释放每对象锁。

在自由线程构建版中，该宏将扩展为:

```
    PyCriticalSection_End(&_py_cs);
}
```

在默认构建版中，该宏将扩展为 `}`。

> *Added in version 3.13.*

Py_BEGIN_CRITICAL_SECTION2(a, b)

为对象 *a* 和 *b* 获取每对象锁并开始一个关键节。 这些锁是按连续顺序获取的（最低的地址在最前）以避免锁顺序列死锁。

在自由线程构建版中，该宏将扩展为:

```
{
    PyCriticalSection2 _py_cs2;
    PyCriticalSection2_Begin(&_py_cs2, (PyObject*)(a), (PyObject*)(b))
```

在默认构建版中，该宏将扩展为 `{`。

> *Added in version 3.13.*

Py_BEGIN_CRITICAL_SECTION2_MUTEX(m1, m2)

锁定互斥锁 *m1* 和 *m2* 并开始一个临界区。

在自由线程构建版中，该宏将扩展为:

```
{
     PyCriticalSection2 _py_cs2;
     PyCriticalSection2_BeginMutex(&_py_cs2, m1, m2)
```

需要注意的是，与 `Py_BEGIN_CRITICAL_SECTION2` 不同，此宏的参数无需类型转换——它们必须是 `PyMutex` 指针。

在默认构建版中，该宏将扩展为 `{`。

> *Added in version 3.14.*

Py_END_CRITICAL_SECTION2()

结束关键节并释放每对象锁。

在自由线程构建版中，该宏将扩展为:

```
    PyCriticalSection2_End(&_py_cs2);
}
```

在默认构建版中，该宏将扩展为 `}`。

> *Added in version 3.13.*