Python 初始化配置

PyInitConfig C API

> *Added in version 3.14.*

Python 可以用 `Py_InitializeFromInitConfig()` 来初始化。

`Py_RunMain()` 函数可被用来编写定制的 Python 程序。

另请参阅 初始化、最终化与线程。

> **参见：** **PEP 741** "Python 配置 C API"。

示例

运行时始终启用 Python 开发模式 的定制版 Python 的例子；出错时将返回 `-1`:

```c
int init_python(void)
{
    PyInitConfig *config = PyInitConfig_Create();
    if (config == NULL) {
        printf("PYTHON INIT ERROR: memory allocation failed\n");
        return -1;
    }

    // 启用 Python 开发模式
    if (PyInitConfig_SetInt(config, "dev_mode", 1) < 0) {
        goto error;
    }

    // 使用配置初始化 Python
    if (Py_InitializeFromInitConfig(config) < 0) {
        goto error;
    }
    PyInitConfig_Free(config);
    return 0;

error:
    {
        // 显示错误消息
        //
        // 使用这种不常见的花括号风格，因为你不能
        // 将跳转目标指向变量声明。
        const char *err_msg;
        (void)PyInitConfig_GetError(config, &err_msg);
        printf("PYTHON INIT ERROR: %s\n", err_msg);
        PyInitConfig_Free(config);
        return -1;
    }
}
```

创建配置

`struct` **PyInitConfig**
用于配置 Python 初始化的不透明结构体。

PyInitConfig ***PyInitConfig_Create**(void)
使用 隔离配置 的默认值新建一个初始化配置。

它必须由 PyInitConfig_Free() 来释放。

内存分配失败时返回 `NULL`。

void **PyInitConfig_Free**(PyInitConfig *config)
释放初始化配置 *config* 的内存。

如果 *config* 为 `NULL`，则不执行任何操作。

错误处理

int **PyInitConfig_GetError**(PyInitConfig *config, const char **err_msg)
获取 *config* 错误消息。

- 如果设置了一个错误则设置 **err_msg* 并返回 `1`。
- 在其他情况下将 **err_msg* 设为 `NULL` 并返回 `0`。

错误消息是一个 UTF-8 编码的字符串。

如果 *config* 具有一个退出码，则将退出码格式化为错误消息。

错误消息将保持无效直到调用另一个 `PyInitConfig` 并附带 *config*。 调用方不必释放错误消息。

int **PyInitConfig_GetExitCode**(PyInitConfig *config, int *exitcode)
获取 *config* 退出码。

- 如果 *config* 设置了一个退出码则设置 **exitcode* 并返回 `1`。
- 如果 *config* 没有设置退出码则返回 `0`。

如果 `parse_argv` 选项为非零值则只有 `Py_InitializeFromInitConfig()` 函数能设置退出码。

退出码的设置可以在解析命令行失败时 (退出码 `2`) 或是在命令行选项请求显示命令行帮助时 (退出码 `0`) 进行。

获取选项

配置选项的 *name* 形参必须是一个非 NULL 的以空值结束的 UTF-8 编码字符串。 参见 配置选项。

int **PyInitConfig_HasOption**(PyInitConfig *config, const char *name)

检测配置是否具有一个名为 *name* 的选项。

如果选项存在则返回 `1`，否则返回 `0`。

`int` **PyInitConfig_GetInt**(`PyInitConfig *config, const char *name, int64_t *value`)

获取一个整数形式的配置选项。

- 在成功时设置 **value*，并返回 `0`。
- 在出错时在 *config* 中设置一个错误并返回 `-1`。

`int` **PyInitConfig_GetStr**(`PyInitConfig *config, const char *name, char **value`)

获取一个以空值结束的 UTF-8 编码字符串形式的字符串配置选项。

- 在成功时设置 **value*，并返回 `0`。
- 在出错时在 *config* 中设置一个错误并返回 `-1`。

如果选项是一个可选字符串并且选项被取消设置则 **value* 可被设为 `NULL`。

在成功时，字符串如果不为 `NULL` 则必须用 `free(value)` 来释放它。

`int` **PyInitConfig_GetStrList**(`PyInitConfig *config, const char *name, size_t *length, char ***items`)

获取一个由以空值结束的 UTF-8 编码字符串组成的数组形式的字符串列表配置选项。

- 在成功时设置 **length* 和 **value*，并返回 `0`。
- 在出错时在 *config* 中设置一个错误并返回 `-1`。

在成功时，字符串列表必须用 `PyInitConfig_FreeStrList(length, items)` 来释放。

`void` **PyInitConfig_FreeStrList**(`size_t length, char **items`)

释放由 `PyInitConfig_GetStrList()` 创建的字符串列表的内存。

设置选项

配置选项的 *name* 形参必须是一个非 NULL 的以空值结束的 UTF-8 编码字符串。 参见 配置选项。

某些配置选项具有对其他选项的附带影响。 这种逻辑仅限于调用了 `Py_InitializeFromInitConfig()` 的时候，而不被下面的“设置”函数所实现。 例如，将 `dev_mode` 设为 `1` 并不会将 `faulthandler` 设为 `1`。

`int` **PyInitConfig_SetInt**(`PyInitConfig *config, const char *name, int64_t value`)

设置一个整数形式的配置选项。

- 成功时返回 `0`。
- 在出错时在 *config* 中设置一个错误并返回 `-1`。

`int` **PyInitConfig_SetStr**(PyInitConfig *config, const char *name, const char *value)

设置一个以空值结束的 UTF-8 编码字符串形式的字符串配置选项。 该字符串将被拷贝。

- 成功时返回 `0`。
- 在出错时在 *config* 中设置一个错误并返回 `-1`。

`int` **PyInitConfig_SetStrList**(PyInitConfig *config, const char *name, size_t length, char *const *items)

设置一个由以空值结束的 UTF-8 编码字符串组成的数组形式的字符串列表配置选项。 该字符串列表将被拷贝。

- 成功时返回 `0`。
- 在出错时在 *config* 中设置一个错误并返回 `-1`。

模块

`int` **PyInitConfig_AddModule**(PyInitConfig *config, const char *name, PyObject *(*initfunc)(void))

将一个内置扩展模块添加到内置模块表。

这个新模块可按名称 *name* 导入，并使用函数 *initfunc* 作为在首次尝试导入时要调用的初始化函数。

- 成功时返回 `0`。
- 在出错时在 *config* 中设置一个错误并返回 `-1`。

如果 Python 要被多次初始化，则 `PyInitConfig_AddModule()` 必须在每次 Python 初始化时被调用。

类似于 `PyImport_AppendInittab()` 函数。

初始化 Python

`int` **Py_InitializeFromInitConfig**(PyInitConfig *config)

根据初始化配置来初始化 Python。

- 成功时返回 `0`。
- 在出错时在 *config* 中设置一个错误并返回 `-1`。
- 如果 Python 想要退出则在 *config* 中设置一个退出码并返回 `-1`。

请参阅 `PyInitConfig_GetExitcode()` 了解退出码用例。

配置选项

属性	PyConfig/PyPreConfig 成员	类型	可见性
`"allocator"`	`allocator`	`int`	只读

属性	PyConfig/PyPreConfig 成员	类型	可见性
"argv"	argv	list[str]	公有
"base_exec_prefix"	base_exec_prefix	str	公有
"base_executable"	base_executable	str	公有
"base_prefix"	base_prefix	str	公有
"buffered_stdio"	buffered_stdio	bool	只读
"bytes_warning"	bytes_warning	int	公有
"check_hash_pycs_mode"	check_hash_pycs_mode	str	只读
"code_debug_ranges"	code_debug_ranges	bool	只读
"coerce_c_locale"	coerce_c_locale	bool	只读
"coerce_c_locale_warn"	coerce_c_locale_warn	bool	只读
"configure_c_stdio"	configure_c_stdio	bool	只读
"configure_locale"	configure_locale	bool	只读
"cpu_count"	cpu_count	int	公有
"dev_mode"	dev_mode	bool	只读
"dump_refs"	dump_refs	bool	只读
"dump_refs_file"	dump_refs_file	str	只读
"exec_prefix"	exec_prefix	str	公有
"executable"	executable	str	公有
"faulthandler"	faulthandler	bool	只读
"filesystem_encoding"	filesystem_encoding	str	只读
"filesystem_errors"	filesystem_errors	str	只读
"hash_seed"	hash_seed	int	只读
"home"	home	str	只读
"import_time"	import_time	int	只读
"inspect"	inspect	bool	公有
"install_signal_handlers"	install_signal_handlers	bool	只读
"int_max_str_digits"	int_max_str_digits	int	公有
"interactive"	interactive	bool	公有

属性	PyConfig/PyPreConfig 成员	类型	可见性
"isolated"	isolated	bool	只读
"legacy_windows_fs_encoding"	legacy_windows_fs_encoding	bool	只读
"legacy_windows_stdio"	legacy_windows_stdio	bool	只读
"malloc_stats"	malloc_stats	bool	只读
"module_search_paths"	module_search_paths	list[str]	公有
"optimization_level"	optimization_level	int	公有
"orig_argv"	orig_argv	list[str]	只读
"parse_argv"	parse_argv	bool	只读
"parser_debug"	parser_debug	bool	公有
"pathconfig_warnings"	pathconfig_warnings	bool	只读
"perf_profiling"	perf_profiling	bool	只读
"platlibdir"	platlibdir	str	公有
"prefix"	prefix	str	公有
"program_name"	program_name	str	只读
"pycache_prefix"	pycache_prefix	str	公有
"quiet"	quiet	bool	公有
"run_command"	run_command	str	只读
"run_filename"	run_filename	str	只读
"run_module"	run_module	str	只读
"run_presite"	run_presite	str	只读
"safe_path"	safe_path	bool	只读
"show_ref_count"	show_ref_count	bool	只读
"site_import"	site_import	bool	只读
"skip_source_first_line"	skip_source_first_line	bool	只读
"stdio_encoding"	stdio_encoding	str	只读
"stdio_errors"	stdio_errors	str	只读
"stdlib_dir"	stdlib_dir	str	公有
"tracemalloc"	tracemalloc	int	只读

属性	PyConfig/PyPreConfig 成员	类型	可见性
`"use_environment"`	use_environment	`bool`	公有
`"use_frozen_modules"`	use_frozen_modules	`bool`	只读
`"use_hash_seed"`	use_hash_seed	`bool`	只读
`"use_system_logger"`	use_system_logger	`bool`	只读
`"user_site_directory"`	user_site_directory	`bool`	只读
`"utf8_mode"`	utf8_mode	`bool`	只读
`"verbose"`	verbose	`int`	公有
`"warn_default_encoding"`	warn_default_encoding	`bool`	只读
`"warnoptions"`	warnoptions	`list[str]`	公有
`"write_bytecode"`	write_bytecode	`bool`	公有
`"xoptions"`	xoptions	`dict[str, str]`	公有
`"_pystats"`	_pystats	`bool`	只读

可见性：

- 公有：可由 PyConfig_Get() 获取并由 PyConfig_Set() 设置。
- 只读：可由 PyConfig_Get() 获取，但不可由 PyConfig_Set() 设置。

运行时 Python 配置 API

在运行时，有可能使用 PyConfig_Get() 和 PyConfig_Set() 函数来获取和设置配置选项。

配置选项的 *name* 形参必须是一个非 NULL 的以空值结束的 UTF-8 编码字符串。 参见 配置选项。

有些选项是从 sys 的属性读取的。 例如，选项 `"argv"` 是从 sys.argv 读取的。

PyObject *__PyConfig_Get__(const char *name)

获取一个配置选项的以 Python 对象表示的当前运行时值。

- 成功时返回一个新的引用。
- 出错时设置一个异常并返回 `NULL`。

对象类型取决于具体的配置选项。 它可以是：

- `bool`
- `int`
- `str`
- `list[str]`

- `dict[str, str]`

调用方必须拥有一个 attached thread state。 这个函数不能在 Python 初始化之前或 Python 最终后之后被调用。

> *Added in version 3.14.*

`int` **PyConfig_GetInt**(`const char *name, int *value`)

类似于 `PyConfig_Get()`，但会获取 C int 形式的值。

- 成功时返回 `0`。
- 出错时设置一个异常并返回 `-1`。

> *Added in version 3.14.*

`PyObject` ***PyConfig_Names**(`void`)

获取以一个 `frozenset` 表示的全部配置选项名称。

- 成功时返回一个新的引用。
- 出错时设置一个异常并返回 `NULL`。

调用方必须拥有一个 attached thread state。 这个函数不能在 Python 初始化之前或 Python 最终后之后被调用。

> *Added in version 3.14.*

`int` **PyConfig_Set**(`const char *name, PyObject *value`)

设置一个配置选项的当前运行时值。

- 如果没有选项 *name* 则会引发 `ValueError`。
- 如果 *value* 是一个无效的值则会引发 `ValueError`。
- 如果选项是只读的（无法被设置）则会引发 `ValueError`。
- 如果 *value* 的类型不正确则会引发 `TypeError`。

调用方必须拥有一个 attached thread state。 这个函数不能在 Python 初始化之前或 Python 最终后之后被调用。

引发一个 审计事件 `cpython.PyConfig_Set` 并附带参数 `name`, `value`。

> *Added in version 3.14.*

PyConfig C API

> *Added in version 3.8.*

Python 可以使用 `Py_InitializeFromConfig()` 和 `PyConfig` 结构体来初始化。 它可以使用 `Py_PreInitialize()` 和 `PyPreConfig` 结构体来预初始化。

有两种配置方式：

- Python 配置 可被用于构建一个定制的 Python，其行为与常规 Python 类似。 例如，环境变量和命令行参数可被用于配置 Python。
- 隔离配置 可被用于将 Python 嵌入到应用程序。 它将 Python 与系统隔离开来。 例如，环境变量将被忽略，LC_CTYPE 语言区域设置保持不变并且不会注册任何信号处理器。

`Py_RunMain()` 函数可被用来编写定制的 Python 程序。

另请参阅 初始化、最终化与线程。

> **参见：** **PEP 587** "Python 初始化配置".

示例

定制的 Python 的示例总是会以隔离模式运行:

```
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

    /* 解码命令行参数。
       隐式地预初始化 Python (隔离模式)。 */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();

exception:
    PyConfig_Clear(&config);
    if (PyStatus_IsExit(status)) {
        return status.exitcode;
    }
    /* 显示错误消息然后退出进程
       并设置非零值退出码 */
    Py_ExitStatusException(status);
}
```

PyWideStringList

type **PyWideStringList**

由 `wchar_t*` 字符串组成的列表。

如果 *length* 为非零值，则 *items* 必须不为 `NULL` 并且所有字符串均必须不为 `NULL`。

方法

PyStatus **PyWideStringList_Append**(PyWideStringList *list, const wchar_t *item)

将 *item* 添加到 *list*。

Python 必须被预初始化以便调用此函数。

PyStatus **PyWideStringList_Insert**(PyWideStringList *list, Py_ssize_t index, const wchar_t *item)

将 *item* 插入到 *list* 的 *index* 位置上。

如果 *index* 大于等于 *list* 的长度，则将 *item* 添加到 *list*。

index 必须大于等于 `0`。

Python 必须被预初始化以便调用此函数。

结构体字段:

Py_ssize_t **length**

List 长度。

wchar_t ***items**

列表项目。

PyStatus

type **PyStatus**

存储初始函数状态：成功、错误或退出的结构体。

对于错误，它可以存储造成错误的 C 函数的名称。

结构体字段:

int **exitcode**

退出码。 传给 `exit()` 的参数。

const char ***err_msg**

错误信息

const char ***func**

造成错误的函数的名称，可以为 `NULL`。

创建状态的函数:

PyStatus **PyStatus_Ok**(void)

完成。

PyStatus **PyStatus_Error**(const char *err_msg)

带消息的初始化错误。

err_msg 不可为 `NULL`。

PyStatus **PyStatus_NoMemory**(void)

内存分配失败（内存不足）。

PyStatus **PyStatus_Exit**(int exitcode)

以指定的退出代码退出 Python。

处理状态的函数:

int **PyStatus_Exception**(PyStatus status)

状态为错误还是退出？如为真值，则异常必须被处理；例如通过调用 Py_ExitStatusException()。

int **PyStatus_IsError**(PyStatus status)

结果错误吗?

int **PyStatus_IsExit**(PyStatus status)

结果是否退出?

void **Py_ExitStatusException**(PyStatus status)

如果 *status* 是一个退出码则调用 `exit(exitcode)`。如果 *status* 是一个错误码则打印错误消息并设置一个非零退出码再退出。 必须在 `PyStatus_Exception(status)` 为非零值时才能被调用。

备注: 在内部，Python 将使用设置 `PyStatus.func` 的宏，而创建状态的函数则会将 `func` 设为 `NULL`。

示例:

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
    return PyStatus_Ok();
}

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

```
{
    void *ptr;
    PyStatus status = alloc(&ptr, 16);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
    PyMem_Free(ptr);
    return 0;
}
```

PyPreConfig

type **PyPreConfig**

用于预初始化 Python 的结构体。

用于初始化预先配置的函数:

void **PyPreConfig_InitPythonConfig**(PyPreConfig *preconfig)

通过 Python 配置 来初始化预先配置。

void **PyPreConfig_InitIsolatedConfig**(PyPreConfig *preconfig)

通过 隔离配置 来初始化预先配置。

结构体字段:

int **allocator**

Python 内存分配器名称:

- `PYMEM_ALLOCATOR_NOT_SET` (`0`): 不改变内存分配器 (使用默认)。
- `PYMEM_ALLOCATOR_DEFAULT` (`1`): 默认内存分配器。
- `PYMEM_ALLOCATOR_DEBUG` (`2`): 默认内存分配器 附带 调试钩子。
- `PYMEM_ALLOCATOR_MALLOC` (`3`): 使用 C 库的 `malloc()`。
- `PYMEM_ALLOCATOR_MALLOC_DEBUG` (`4`): 强制使用 `malloc()` 附带 调试钩子。
- `PYMEM_ALLOCATOR_PYMALLOC` (`5`): Python pymalloc 内存分配器。
- `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` (`6`): Python pymalloc 内存分配器 附带 调试钩子。
- `PYMEM_ALLOCATOR_MIMALLOC` (`6`): 使用 `mimalloc`，一个快速的 malloc 替代。
- `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` (`7`): 使用 `mimalloc`，一个快速的 malloc 替代，它带有 调试钩子。

如果 Python 是 使用 `--without-pymalloc` 进行配置 则 `PYMEM_ALLOCATOR_PYMALLOC` 和 `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` 将不被支持。

如果 Python 是 使用 `--without-mimalloc` 进行配置 或者如果下层的原子化支持不可用则 `PYMEM_ALLOCATOR_MIMALLOC` 和 `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` 将不被支持。

参见 Memory Management.

默认值: `PYMEM_ALLOCATOR_NOT_SET`。

`int` **`configure_locale`**

将 LC_CTYPE 语言区域设为用户选择的语言区域。

如果等于 `0`，则将 `coerce_c_locale` 和 `coerce_c_locale_warn` 的成员设为 `0`。

参见 locale encoding。

默认值: 在 Python 配置中为 `1`，在隔离配置中为 `0`。

`int` **`coerce_c_locale`**

如果等于 `2`，强制转换 C 语言区域。

如果等于 `1`，则读取 LC_CTYPE 语言区域来确定其是否应当被强制转换。

参见 locale encoding。

默认值: 在 Python 配置中为 `-1`，在隔离配置中为 `0`。

`int` **`coerce_c_locale_warn`**

如为非零值，则会在 C 语言区域被强制转换时发出警告。

默认值: 在 Python 配置中为 `-1`，在隔离配置中为 `0`。

`int` **`dev_mode`**

Python 开发模式: 参见 `PyConfig.dev_mode`。

默认值: 在 Python 模式中为 `-1`，在隔离模式中为 `0`。

`int` **`isolated`**

隔离模式：参见 `PyConfig.isolated`。

默认值: 在 Python 模式中为 `0`，在隔离模式中为 `1`。

`int` **`legacy_windows_fs_encoding`**

如为非零值:

- 设置 `PyPreConfig.utf8_mode` 为 `0`,
- 设置 `PyConfig.filesystem_encoding` 为 `"mbcs"`,
- 设置 `PyConfig.filesystem_errors` 为 `"replace"`.

基于 `PYTHONLEGACYWINDOWSFSENCODING` 环境变量值完成初始化。

仅在 Windows 上可用。 `#ifdef MS_WINDOWS` 宏可被用于 Windows 专属的代码。

默认值: `0`.

`int` **`parse_argv`**

如为非零值，`Py_PreInitializeFromArgs()` 和 `Py_PreInitializeFromBytesArgs()` 将以与常规 Python 解析命令行参数的相同方式解析其 `argv` 参数：参见 命令行参数。

默认值: 在 Python 配置中为 `1`，在隔离配置中为 `0`。

`int` **use_environment**

使用 环境变量? 参见 `PyConfig.use_environment`。

默认值: 在 Python 配置中为 `1` 而在隔离配置中为 `0`。

`int` **utf8_mode**

如为非零值，则启用 Python UTF-8 模式。

通过 `-X utf8` 命令行选项和 `PYTHONUTF8` 环境变量设为 `0` 或 `1`。

如果 `LC_CTYPE` 语言区域为 `C` 或 `POSIX` 也会被设为 `1`。

默认值: 在 Python 配置中为 `-1` 而在隔离配置中为 `0`。

使用 PyPreConfig 预初始化 Python

Python 的预初始化:

- 设置 Python 内存分配器 (`PyPreConfig.allocator`)
- 配置 LC_CTYPE 语言区域 (locale encoding)
- 设置 Python UTF-8 模式 (`PyPreConfig.utf8_mode`)

当前的预配置 (`PyPreConfig` 类型) 保存在 `_PyRuntime.preconfig` 中。

用于预初始化 Python 的函数:

`PyStatus` **Py_PreInitialize**(`const PyPreConfig *preconfig`)

根据 *preconfig* 预配置来预初始化 Python。

preconfig 不可为 `NULL`。

`PyStatus` **Py_PreInitializeFromBytesArgs**(`const PyPreConfig *preconfig, int argc, char *const *argv`)

根据 *preconfig* 预配置来预初始化 Python。

如果 *preconfig* 的 `parse_argv` 为非零值则解析 *argv* 命令行参数（字节串）。

preconfig 不可为 `NULL`。

`PyStatus` **Py_PreInitializeFromArgs**(`const PyPreConfig *preconfig, int argc, wchar_t *const *argv`)

根据 *preconfig* 预配置来预初始化 Python。

如果 *preconfig* 的 `parse_argv` 为非零值则解析 *argv* 命令行参数（宽字符串）。

preconfig 不可为 `NULL`。

调用方要负责使用 `PyStatus_Exception()` 和 `Py_ExitStatusException()` 来处理异常（错误或退出）。

对于 Python 配置 (`PyPreConfig_InitPythonConfig()`)，如果 Python 是用命令行参数初始化的，那么在预初始化 Python 时也必须传递命令行参数，因为它们会对编码格式等预配置产生影响。 例如，`-X utf8` 命令行选项将启用 Python UTF-8 模式。

`PyMem_SetAllocator()` 可在 `Py_PreInitialize()` 之后、`Py_InitializeFromConfig()` 之前被调用以安装自定义的内存分配器。 如果 `PyPreConfig.allocator` 被设为 `PYMEM_ALLOCATOR_NOT_SET` 则可在 `Py_PreInitialize()` 之前被调用。

像 `PyMem_RawMalloc()` 这样的 Python 内存分配函数不能在 Python 预初始化之前使用，而直接调用 `malloc()` 和 `free()` 则始终会是安全的。 `Py_DecodeLocale()` 不能在 Python 预初始化之前被调用。

使用预初始化来启用 Python UTF-8 模式 的例子:

```
PyStatus status;
PyPreConfig preconfig;
PyPreConfig_InitPythonConfig(&preconfig);

preconfig.utf8_mode = 1;

status = Py_PreInitialize(&preconfig);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}

/* 此时，Python 将使用 UTF-8 */

Py_Initialize();
/* ... 在此使用 Python API ... */
Py_Finalize();
```

PyConfig

type **PyConfig**

包含了大部分用于配置 Python 的形参的结构体。

在完成后，必须使用 `PyConfig_Clear()` 函数来释放配置内存。

结构体方法:

void **PyConfig_InitPythonConfig**(PyConfig *config)

通过 Python 配置 来初始化配置。

void **PyConfig_InitIsolatedConfig**(PyConfig *config)

通过 隔离配置 来初始化配置。

PyStatus **PyConfig_SetString**(PyConfig *config, wchar_t *const *config_str, const wchar_t *str)

将宽字符串 *str* 拷贝至 `*config_str`。

在必要时 预初始化 Python。

PyStatus **PyConfig_SetBytesString**(PyConfig *config, wchar_t *const *config_str, const char *str)

使用 Py_DecodeLocale() 对 *str* 进行解码并将结果设置到 `*config_str`。

在必要时 预初始化 Python。

PyStatus **PyConfig_SetArgv**(PyConfig *config, int argc, wchar_t *const *argv)

根据宽字符串列表 *argv* 设置命令行参数 (*config* 的 argv 成员)。

在必要时 预初始化 Python。

PyStatus **PyConfig_SetBytesArgv**(PyConfig *config, int argc, char *const *argv)

根据字节串列表 *argv* 设置命令行参数 (*config* 的 argv 成员)。 使用 Py_DecodeLocale() 对字节串进行解码。

在必要时 预初始化 Python。

PyStatus **PyConfig_SetWideStringList**(PyConfig *config, PyWideStringList *list, Py_ssize_t length, wchar_t **items)

将宽字符串列表 *list* 设置为 *length* 和 *items*。

在必要时 预初始化 Python。

PyStatus **PyConfig_Read**(PyConfig *config)

读取所有 Python 配置。

已经初始化的字段会保持不变。

调用此函数时不再计算或修改用于 路径配置 的字段，如 Python 3.11 那样。

PyConfig_Read() 函数对 PyConfig.argv 参数只会解析一次：在参数解析完成后 PyConfig.parse_argv 将被设为 `2`。 由于 Python 参数是从 PyConfig.argv 提取的，因此解析参数两次会将应用程序选项解析为 Python 选项。

在必要时 预初始化 Python。

> *在 3.10 版本发生变更:* PyConfig.argv 参数现在只会被解析一次，在参数解析完成后，PyConfig.parse_argv 将被设为 2，只有当 PyConfig.parse_argv 等于 1 时才会解析参数。

> *在 3.11 版本发生变更:* PyConfig_Read() 不会再计算所有路径，因此在 Python 路径配置 下列出的字段可能不会再更新直到 Py_InitializeFromConfig() 被调用。

void **PyConfig_Clear**(PyConfig *config)

释放配置内存

如有必要大多数 PyConfig 方法将会 预初始化 Python。 在这种情况下，Python 预初始化配置 (PyPreConfig) 将以 PyConfig 为基础。 如果要调整与 PyPreConfig 相同的配置字段，它们必须在调用 PyConfig 方法之前被设置:

- PyConfig.dev_mode
- PyConfig.isolated
- PyConfig.parse_argv
- PyConfig.use_environment

此外，如果使用了 PyConfig_SetArgv() 或 PyConfig_SetBytesArgv()，则必须在调用其他方法之前调用该方法，因为预初始化配置取决于命令行参数（如果 parse_argv 为非零值）。

这些方法的调用者要负责使用 PyStatus_Exception() 和 Py_ExitStatusException() 来处理异常（错误或退出）。

结构体字段:

PyWideStringList **argv**

根据 argv 设置 sys.argv 命令行参数。 这些形参与传给程序的 main() 函数的类似，区别在于其中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，而 argv 中的第一项可以为空字符串。

将 parse_argv 设为 1 将以与普通 Python 解析 Python 命令行参数相同的方式解析 argv 再从 argv 中剥离 Python 参数。

如果 argv 为空，则会添加一个空字符串以确保 sys.argv 始终存在并且永远不为空。

默认值: NULL.

另请参阅 orig_argv 成员。

int **safe_path**

如果等于零，Py_RunMain() 会在启动时向 sys.path 开头添加一个可能不安全的路径:

- 如果 argv[0] 等于 L"-m" (python -m module)，则添加当前工作目录。
- 如果是运行脚本 (python script.py)，则添加脚本的目录。 如果是符号链接，则会解析符号链接。

- 在其他情况下（`python -c code` 和 `python`），将添加一个空字符串，这表示当前工作目录。

通过 `-P` 命令行选项和 `PYTHONSAFEPATH` 环境变量设置为 `1`。

默认值：Python 配置中为 `0`，隔离配置中为 `1`。

> *Added in version 3.11.*

`wchar_t *`**`base_exec_prefix`**

`sys.base_exec_prefix`.

默认值: `NULL`.

Python 路径配置 的一部分。

另请参阅 `PyConfig.exec_prefix`。

`wchar_t *`**`base_executable`**

Python 基础可执行文件: `sys._base_executable`。

由 `__PYVENV_LAUNCHER__` 环境变量设置。

如为 `NULL` 则从 `PyConfig.executable` 设置。

默认值: `NULL`.

Python 路径配置 的一部分。

另请参阅 `PyConfig.executable`。

`wchar_t *`**`base_prefix`**

`sys.base_prefix`.

默认值: `NULL`.

Python 路径配置 的一部分。

另请参阅 `PyConfig.prefix`。

`int` **`buffered_stdio`**

如果等于 `0` 且 `configure_c_stdio` 为非零值，则禁用 C 数据流 stdout 和 stderr 的缓冲。

通过 `-u` 命令行选项和 `PYTHONUNBUFFERED` 环境变量设置为 `0`。

stdin 始终以缓冲模式打开。

默认值: `1`.

`int` **bytes_warning**

如果等于 `1`，则在将 `bytes` 或 `bytearray` 与 `str` 进行比较，或将 `bytes` 与 `int` 进行比较时发出警告。

如果大于等于 `2`，则在这些情况下引发 `BytesWarning` 异常。

由 `-b` 命令行选项执行递增。

默认值: `0`.

`int` **warn_default_encoding**

如为非零值，则在 `io.TextIOWrapper` 使用默认编码格式时发出 `EncodingWarning` 警告。 详情请参阅 选择性的 EncodingWarning。

默认值: `0`.

> *Added in version 3.10.*

`int` **code_debug_ranges**

如果等于 `0`，则禁用在代码对象中包括末尾行和列映射。 并且禁用在特定错误位置打印回溯标记。

通过 `PYTHONNODEBUGRANGES` 环境变量和 `-X no_debug_ranges` 命令行选项设置为 `0`。

默认值: `1`.

> *Added in version 3.11.*

`wchar_t *`**check_hash_pycs_mode**

控制基于哈希值的 `.pyc` 文件的验证行为: `--check-hash-based-pycs` 命令行选项的值。

有效的值:

- `L"always"`: 无论 'check_source' 旗标的值是什么都会对源文件进行哈希验证。
- `L"never"`: 假定基于哈希值的 pyc 始终是有效的。
- `L"default"`: 基于哈希值的 pyc 中的 'check_source' 旗标确定是否验证无效。

默认值: `L"default"`。

参见 **PEP 552** "Deterministic pycs"。

`int` **configure_c_stdio**

如为非零值，则配置 C 标准流:

- 在 Windows 中，在 stdin, stdout 和 stderr 上设置二进制模式 (`O_BINARY`)。
- 如果 `buffered_stdio` 等于零，则禁用 stdin, stdout 和 stderr 流的缓冲。
- 如果 `interactive` 为非零值，则启用 stdin 和 stdout 上的流缓冲（Windows 中仅限 stdout）。

默认值: 在 Python 配置中为 `1`，在隔离配置中为 `0`。

`int` **dev_mode**

如果为非零值，则启用 Python 开发模式。

通过 `-X dev` 选项和 `PYTHONDEVMODE` 环境变量设置为 `1`。

默认值: 在 Python 模式中为 `-1`，在隔离模式中为 `0`。

`int` **dump_refs**

转储 Python 引用?

如果为非零值，则转储所有在退出时仍存活的对象。

由 `PYTHONDUMPREFS` 环境变量设置为 `1`。

需要定义了 `Py_TRACE_REFS` 宏的特殊 Python 编译版：参见 `configure --with-trace-refs 选项`。

默认值: `0`.

`wchar_t *` **dump_refs_file**

转储 Python 引用的目标文件名。

由 `PYTHONDUMPREFSFILE` 环境变量设置。

默认值: `NULL`.

> *Added in version 3.11.*

`wchar_t *` **exec_prefix**

安装依赖于平台的 Python 文件的站点专属目录前缀: `sys.exec_prefix`。

默认值: `NULL`.

Python 路径配置 的一部分。

另请参阅 `PyConfig.base_exec_prefix`。

`wchar_t *` **executable**

Python 解释器可执行二进制文件的绝对路径: `sys.executable`。

默认值: `NULL`.

Python 路径配置 的一部分。

另请参阅 `PyConfig.base_executable`。

`int` **faulthandler**

启用 faulthandler?

如果为非零值，则在启动时调用 `faulthandler.enable()`。

通过 `-X faulthandler` 和 `PYTHONFAULTHANDLER` 环境变量设为 `1`。

默认值: 在 Python 模式中为 `-1`，在隔离模式中为 `0`。

`wchar_t *`**`filesystem_encoding`**

文件系统编码格式: `sys.getfilesystemencoding()`。

在 macOS, Android 和 VxWorks 上：默认使用 `"utf-8"`。

在 Windows 上：默认使用 `"utf-8"`，或者如果 `PyPreConfig` 的 `legacy_windows_fs_encoding` 为非零值则使用 `"mbcs"`。

在其他平台上的默认编码格式：

- 如果 `PyPreConfig.utf8_mode` 为非零值则使用 `"utf-8"`。
- 如果 Python 检测到 `nl_langinfo(CODESET)` 声明为 ASCII 编码格式，而 `mbstowcs()` 是从其他的编码格式解码（通常为 Latin1）则使用 `"ascii"`。
- 如果 `nl_langinfo(CODESET)` 返回空字符串则使用 `"utf-8"`。
- 在其他情况下，使用 locale encoding: `nl_langinfo(CODESET)` 的结果。

在 Python 启动时，编码格式名称会规范化为 Python 编解码器名称。 例如，`"ANSI_X3.4-1968"` 将被替换为 `"ascii"`。

参见 `filesystem_errors` 的成员。

`wchar_t *`**`filesystem_errors`**

文件系统错误处理器: `sys.getfilesystemencodeerrors()`。

在 Windows 上：默认使用 `"surrogatepass"`，或者如果 `PyPreConfig` 的 `legacy_windows_fs_encoding` 为非零值则使用 `"replace"`。

在其他平台上：默认使用 `"surrogateescape"`。

支持的错误处理器：

- `"strict"`
- `"surrogateescape"`
- `"surrogatepass"` (仅支持 UTF-8 编码格式)

参见 `filesystem_encoding` 的成员。

`int` **`use_frozen_modules`**

如为非零值，则使用冻结模块。

由 `PYTHON_FROZEN_MODULES` 环境变量设置。

默认值：在发布构建版中为 `1`，而在 调试构建版 中为 `0`。

`unsigned long` **hash_seed**

`int` **use_hash_seed**

随机化的哈希函数种子。

如果 `use_hash_seed` 为零，则在 Python 启动时随机选择一个种子，并忽略 `hash_seed`。

由 `PYTHONHASHSEED` 环境变量设置。

默认的 *use_hash_seed* 值：在 Python 模式下为 `-1`，在隔离模式下为 `0`。

`wchar_t *`**home**

设置默认的 Python "home" 目录，即标准 Python 库所在的位置 (参见 `PYTHONHOME`)。

由 `PYTHONHOME` 环境变量设置。

默认值: `NULL`.

Python 路径配置 输入的一部分。

`int` **import_time**

如为 `1`，则会针对导入时间进行性能分析。 如为 `2`，则会包括提示当被导入模块已被加载的额外输出。

通过 `-X importtime` 选项和 `PYTHONPROFILEIMPORTTIME` 环境变量设置。

默认值: `0`.

> *在 3.14 版本发生变更:* 添加了对 `import_time = 2` 的支持

`int` **inspect**

在执行脚本或命令之后进入交互模式。

如果大于 `0` ，则启用检查：当脚本作为第一个参数传入或使用了 -c 选项时，在执行脚本或命令后进入交互模式，即使在 `sys.stdin` 看来并非一个终端时也是如此。

通过 `-i` 命令行选项执行递增。 如果 `PYTHONINSPECT` 环境变量为非空值则设为 `1`。

默认值: `0`.

`int` **install_signal_handlers**

安装 Python 信号处理器?

默认值：在 Python 模式下为 `1`，在隔离模式下为 `0`。

`int` **interactive**

如果大于 `0`，则启用交互模式（REPL）。

由 `-i` 命令行选项执行递增。

默认值: `0`.

`int` **int_max_str_digits**

配置 整数字符串转换长度限制。 初始值为 `-1` 表示该值将从命令行或环境获取否则默认为 4300 (`sys.int_info.default_max_str_digits`)。 值为 `0` 表示禁用限制。 大于 0 但小于 640 (`sys.int_info.str_digits_check_threshold`) 的值将不被支持并会产生错误。

通过 `-X int_max_str_digits` 命令行旗标或 `PYTHONINTMAXSTRDIGITS` 环境变量配置。

默认值：在 Python 模式下为 `-1` 。 在孤立模式下为 4300 (`sys.int_info.default_max_str_digits`)。

> *Added in version 3.12.*

`int` **cpu_count**

如果 `cpu_count` 的值不为 `-1` 则它将覆盖 `os.cpu_count()`, `os.process_cpu_count()` 和 `multiprocessing.cpu_count()` 的返回值。

通过 `-X cpu_count=n|default` 命令行旗标或 `PYTHON_CPU_COUNT` 环境变量来配置。

默认值: `-1`。

> *Added in version 3.13.*

`int` **isolated**

如果大于 `0` ，则启用隔离模式：

- 将 `safe_path` 设为 `1`: 在 Python 启动时将不在 `sys.path` 前添加有潜在不安全性的路径，如当前目录、脚本所在目录或空字符串。
- 将 `use_environment` 设为 `0`: 忽略 `PYTHON` 环境变量。
- 将 `user_site_directory` 设为 `0`: 不要将用户级站点目录添加到 `sys.path`。
- Python REPL 将不导入 `readline` 也不在交互提示符中启用默认的 readline 配置。

通过 `-I` 命令行选项设置为 `1`。

默认值: 在 Python 模式中为 `0`，在隔离模式中为 `1`。

另请参阅 隔离配置 和 `PyPreConfig.isolated`。

`int` **legacy_windows_stdio**

如为非零值，则使用 `io.FileIO` 代替 `io._WindowsConsoleIO` 作为 `sys.stdin`、`sys.stdout` 和 `sys.stderr`。

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

仅在 Windows 上可用。 `#ifdef MS_WINDOWS` 宏可被用于 Windows 专属的代码。

默认值: `0`.

另请参阅 **PEP 528** (将 Windows 控制台编码格式更改为 UTF-8)。

`int` **malloc_stats**

如为非零值，则在退出时转储 Python pymalloc 内存分配器 的统计数据。

由 `PYTHONMALLOCSTATS` 环境变量设置为 `1`。

如果 Python 是 使用 `--without-pymalloc` 选项进行配置 则该选项将被忽略。

默认值: `0`.

`wchar_t *`**platlibdir**

平台库目录名称: `sys.platlibdir`。

由 `PYTHONPLATLIBDIR` 环境变量设置。

默认值：由 `configure --with-platlibdir` 选项 设置的 `PLATLIBDIR` 宏的值 (默认值: `"lib"`，在 Windows 上则为 `"DLLs"`)。

Python 路径配置 输入的一部分。

> *Added in version 3.9.*

> *在 3.11 版本发生变更:* 目前在 Windows 系统中该宏被用于定位标准库扩展模块，通常位于 `DLLs` 下。 不过，出于兼容性考虑，请注意在任何非标准布局包括树内构建和虚拟环境中，该值都将被忽略。

`wchar_t *`**pythonpath_env**

模块搜索路径 (`sys.path`) 为一个用 `DELIM` (`os.pathsep`) 分隔的字符串。

由 `PYTHONPATH` 环境变量设置。

默认值: `NULL`.

Python 路径配置 输入的一部分。

`PyWideStringList` **module_search_paths**

`int` **module_search_paths_set**

模块搜索路径: `sys.path`。

如果 `module_search_paths_set` 等于 `0`，`Py_InitializeFromConfig()` 将替代 `module_search_paths` 并将 `module_search_paths_set` 设为 `1`。

默认值：空列表 (`module_search_paths`) 和 `0` (`module_search_paths_set`)。

Python 路径配置 的一部分。

int **optimization_level**

编译优化级别：

- 0: Peephole 优化器，将 __debug__ 设为 True。
- 1: 0 级，移除断言，将 __debug__ 设为 False。
- 2: 1 级，去除文档字符串。

通过 -O 命令行选项递增。 设置为 PYTHONOPTIMIZE 环境变量值。

默认值: 0.

PyWideStringList **orig_argv**

传给 Python 可执行程序的原始命令行参数列表: sys.orig_argv。

如果 orig_argv 列表为空并且 argv 不是一个只包含空字符串的列表，PyConfig_Read() 将在修改 argv 之前把 argv 拷贝至 orig_argv (如果 parse_argv 不为空)。

另请参阅 argv 成员和 Py_GetArgcArgv() 函数。

默认值：空列表。

> *Added in version 3.10.*

int **parse_argv**

解析命令行参数?

如果等于 1，则以与常规 Python 解析 命令行参数 相同的方式解析 argv，并从 argv 中剥离 Python 参数。

PyConfig_Read() 函数对 PyConfig.argv 参数只会解析一次：在参数解析完成后 PyConfig.parse_argv 将被设为 2。 由于 Python 参数是从 PyConfig.argv 提取的，因此解析参数两次会将应用程序选项解析为 Python 选项。

默认值：在 Python 模式下为 1，在隔离模式下为 0。

> *在 3.10 版本发生变更:* 现在只有当 PyConfig.parse_argv 等于 1 时才会解析 PyConfig.argv 参数。

int **parser_debug**

解析器调试模式。 如果大于 0，则打开解析器调试输出（仅针对专家，取决于编译选项）。

通过 -d 命令行选项递增。 设置为 PYTHONDEBUG 环境变量值。

需要 Python 调试编译版 (必须已定义 Py_DEBUG 宏)。

默认值: `0`.

`int` **pathconfig_warnings**

如为非零值，则允许计算路径配置以将警告记录到 `stderr` 中。 如果等于 `0`，则抑制这些警告。

默认值：在 Python 模式下为 `1`，在隔离模式下为 `0`。

Python 路径配置 输入的一部分。

> *在 3.11 版本发生变更:* 现在也适用于 Windows。

`wchar_t *`**prefix**

安装依赖于平台的 Python 文件的站点专属目录前缀: `sys.prefix`。

默认值: `NULL`.

Python 路径配置 的一部分。

另请参阅 `PyConfig.base_prefix`。

`wchar_t *`**program_name**

用于初始化 `executable` 和在 Python 初始化期间早期错误消息中使用的程序名称。

- 在 macOS 上，如果设置了 `PYTHONEXECUTABLE` 环境变量则会使用它。
- 如果定义了 `WITH_NEXT_FRAMEWORK` 宏，当设置了 `__PYVENV_LAUNCHER__` 环境变量时将会使用它。
- 如果 `argv` 的 `argv[0]` 可用并且不为空值则会使用它。
- 否则，在 Windows 上将使用 `L"python"`，在其他平台上将使用 `L"python3"`。

默认值: `NULL`.

Python 路径配置 输入的一部分。

`wchar_t *`**pycache_prefix**

缓存 `.pyc` 文件被写入到的目录: `sys.pycache_prefix`。

通过 `-X pycache_prefix=PATH` 命令行选项和 `PYTHONPYCACHEPREFIX` 环境变量设置。 命令行选项优先级更高。

如果为 `NULL`，则 `sys.pycache_prefix` 将被设为 `None`。

默认值: `NULL`.

`int` **quiet**

安静模式。 如果大于 `0`，则在交互模式下启动 Python 时不显示版权和版本。

由 `-q` 命令行选项执行递增。

默认值: `0`.

`wchar_t *`**`run_command`**

`-c` 命令行选项的值。

由 `Py_RunMain()` 使用。

默认值: `NULL`.

`wchar_t *`**`run_filename`**

通过命令行传入的文件名：不包含 `-c` 或 `-m` 的附加命令行参数。 它会被 `Py_RunMain()` 函数使用。

例如，对于命令行 `python3 script.py arg` 它将被设为 `script.py`。

另请参阅 `PyConfig.skip_source_first_line` 选项。

默认值: `NULL`.

`wchar_t *`**`run_module`**

`-m` 命令行选项的值。

由 `Py_RunMain()` 使用。

默认值: `NULL`.

`wchar_t *`**`run_presite`**

`package.module` 模块路径，它应当在运行 `site.py` 之前被导入。

通过 `-X presite=package.module` 命令行选项和 `PYTHON_PRESITE` 环境变量设置。 命令行选项优先级更高。

需要 Python 调试编译版 (必须已定义 `Py_DEBUG` 宏)。

默认值: `NULL`.

`int` **`show_ref_count`**

是否要在退出时显示总引用计数 (不包括 immortal 对象)?

通过 `-X showrefcount` 命令行选项设置为 `1`。

需要 Python 调试编译版 (必须已定义 `Py_REF_DEBUG` 宏)。

默认值: `0`.

`int` **`site_import`**

在启动时导入 `site` 模块?

如果等于零，则禁用模块站点的导入以及由此产生的与站点相关的 `sys.path` 操作。

如果以后显式地导入 site 模块也要禁用这些操作（如果你希望触发这些操作，请调用 site.main() 函数）。

通过 -S 命令行选项设置为 0。

sys.flags.no_site 会被设为 site_import 取反后的值。

默认值: 1.

int **skip_source_first_line**

如为非零值，则跳过 PyConfig.run_filename 源的第一行。

它将允许使用非 Unix 形式的 #!cmd。 这是针对 DOS 专属的破解操作。

通过 -x 命令行选项设置为 1。

默认值: 0.

wchar_t ***stdio_encoding**

wchar_t ***stdio_errors**

sys.stdin、sys.stdout 和 sys.stderr 的编码格式和编码格式错误（但 sys.stderr 将始终使用 "backslashreplace" 错误处理器）。

如果 PYTHONIOENCODING 环境变量非空则会使用它。

默认编码格式：

- 如果 PyPreConfig.utf8_mode 为非零值则使用 "UTF-8"。
- 在其他情况下，使用 locale encoding。

默认错误处理器：

- 在 Windows 上：使用 "surrogateescape"。
- 如果 PyPreConfig.utf8_mode 为非零值，或者如果 LC_CTYPE 语言区域为 "C" 或 "POSIX" 则使用 "surrogateescape"。
- 在其他情况下则使用 "strict"。

另请参阅 PyConfig.legacy_windows_stdio。

int **tracemalloc**

启用 tracemalloc?

如果为非零值，则在启动时调用 tracemalloc.start()。

通过 -X tracemalloc=N 命令行选项和 PYTHONTRACEMALLOC 环境变量设置。

默认值: 在 Python 模式中为 -1，在隔离模式中为 0。

int **perf_profiling**

是否启用 Linux `perf` 性能分析器支持?

如果等于 `1`，则启用对 Linux `perf` 性能分析器的支持。

如果等于 `2`，则启用对带有 DWARF JIT 支持的 Linux `perf` 性能分析器的支持。

通过 `-X perf` 命令行选项和 `PYTHONPERFSUPPORT` 环境变量设置为 `1`。

通过 `-X perf_jit` 命令行选项和 `PYTHON_PERF_JIT_SUPPORT` 环境变量设置为 `2`。

默认值: `-1`。

参见: 请参阅 Python 对 Linux perf 性能分析器的支持 了解详情。

Added in version 3.12.

wchar_t ***stdlib_dir**

Python 标准库的目录。

默认值: `NULL`.

Added in version 3.11.

int **use_environment**

使用 环境变量?

如果等于零，则忽略 环境变量。

由 `-E` 环境变量设置为 `0`。

默认值: 在 Python 配置中为 `1` 而在隔离配置中为 `0`。

int **use_system_logger**

如为非零值，则 `stdout` 和 `stderr` 将被重定向到系统日志。

仅在 macOS 10.12 和更新版本，以及 iOS 上可用。

默认值: 在 macOS 上为 `0` (不使用系统日志); 在 iOS 上为 `1` (使用系统日志)。

Added in version 3.14.

int **user_site_directory**

如果为非零值，则将用户站点目录添加到 `sys.path`。

通过 `-s` 和 `-I` 命令行选项设置为 `0`。

由 `PYTHONNOUSERSITE` 环境变量设置为 `0`。

默认值：在 Python 模式下为 `1`，在隔离模式下为 `0`。

`int` **verbose**

详细模式。 如果大于 `0`，则每次导入模块时都会打印一条消息，显示加载模块的位置（文件名或内置模块）。

如果大于等于 `2`，则为搜索模块时每个被检查的文件打印一条消息。 还在退出时提供关于模块清理的信息。

由 `-v` 命令行选项执行递增。

通过 `PYTHONVERBOSE` 环境变量值设置。

默认值: `0`.

`PyWideStringList` **warnoptions**

`warnings` 模块用于构建警告过滤器的选项，优先级从低到高: `sys.warnoptions`。

`warnings` 模块以相反的顺序添加 `sys.warnoptions`: 最后一个 `PyConfig.warnoptions` 条目将成为 `warnings.filters` 的第一个条目并将最先被检查（最高优先级）。

`-W` 命令行选项会将其值添加到 `warnoptions` 中，它可以被多次使用。

`PYTHONWARNINGS` 环境变量也可被用于添加警告选项。 可以指定多个选项，并以逗号 (`,`) 分隔。

默认值：空列表。

`int` **write_bytecode**

如果等于 `0`，Python 将不会尝试在导入源模块时写入 `.pyc` 文件。

通过 `-B` 命令行选项和 `PYTHONDONTWRITEBYTECODE` 环境变量设置为 `0`。

`sys.dont_write_bytecode` 会被初始化为 `write_bytecode` 取反后的值。

默认值: `1`.

`PyWideStringList` **xoptions**

`-X` 命令行选项的值: `sys._xoptions`。

默认值：空列表。

`int` **_pystats**

如为非零值，则在 Python 退出时写入性能统计数据。

需要带有 `Py_STATS` 宏的特殊构建版：参见 `--enable-pystats`。

默认值: `0`.

如果 parse_argv 为非零值，则 argv 参数将以与常规 Python 解析 命令行参数 相同的方式被解析，并从 argv 中剥离 Python 参数。

xoptions 选项将会被解析以设置其他选项：参见 -X 命令行选项。

> *在 3.9 版本发生变更:* show_alloc_count 字段已被移除。

使用 PyConfig 初始化

基于一个已填充内容的配置结构体初始化解释器是通过调用 Py_InitializeFromConfig() 来处理的。

调用方要负责使用 PyStatus_Exception() 和 Py_ExitStatusException() 来处理异常（错误或退出）。

如果使用了 PyImport_FrozenModules()、PyImport_AppendInittab() 或 PyImport_ExtendInittab()，则必须在 Python 预初始化之后、Python 初始化之前设置或调用它们。 如果 Python 被多次初始化，则必须在每次初始化 Python 之前调用 PyImport_AppendInittab() 或 PyImport_ExtendInittab()。

当前的配置 (PyConfig 类型) 保存在 PyInterpreterState.config 中。

设置程序名称的示例:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* 设置程序名称。 隐式地预初始化 Python。 */
    status = PyConfig_SetString(&config, &config.program_name,
                                L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);
    return;

exception:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

更完整的示例会修改默认配置，读取配置，然后覆盖某些参数。 请注意自 3.11 版开始，许多参数在初始化之前不会被计算，因此无法从配置结构体中读取值。在调用初始化之前设置的任何值都将不会被初始化操作改变:

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* 在读取配置之前设置程序名称
       (基于语言区域的编码格式来解码字节串)。

       Implicitly preinitialize Python. */
    status = PyConfig_SetBytesString(&config, &config.program_name,
                                     program_name);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* 一次性读取所有配置 */
    status = PyConfig_Read(&config);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* 显式地指定 sys.path */
    /* 如果你希望修改默认的路径集合,
       可先完成初始化再使用 PySys_GetObject("path") */
    config.module_search_paths_set = 1;
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/stdlib");
    if (PyStatus_Exception(status)) {
        goto done;
    }
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/more/modules");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Override executable computed by PyConfig_Read() */
    status = PyConfig_SetString(&config, &config.executable,
                                L"/path/to/my_executable");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    status = Py_InitializeFromConfig(&config);

done:
    PyConfig_Clear(&config);
    return status;
}
```

隔离配置

PyPreConfig_InitIsolatedConfig() 和 PyConfig_InitIsolatedConfig() 函数会创建一个配置来将 Python 与系统隔离开来。 例如，将 Python 嵌入到某个应用程序。

该配置将忽略全局配置变量、环境变量、命令行参数 (PyConfig.argv 将不会被解析) 和用户站点目录。 C 标准流 (例如 stdout) 和 LC_CTYPE 语言区域将保持不变。 信号处理器将不会被安装。

该配置仍然会使用配置文件来确定未被指明的路径。 请确保指定了 PyConfig.home 以避免计算默认的路径配置。

Python 配置

PyPreConfig_InitPythonConfig() 和 PyConfig_InitPythonConfig() 函数会创建一个配置来构建一个行为与常规 Python 相同的自定义 Python。

环境变量和命令行参数将被用于配置 Python，而全局配置变量将被忽略。

此函数将根据 LC_CTYPE 语言区域、PYTHONUTF8 和 PYTHONCOERCECLOCALE 环境变量启用 C 语言区域强制转换 (**PEP 538**) 和 Python UTF-8 模式 (**PEP 540**)。

Python 路径配置

PyConfig 包含多个用于路径配置的字段：

- 路径配置输入：
  - PyConfig.home
  - PyConfig.platlibdir
  - PyConfig.pathconfig_warnings
  - PyConfig.program_name
  - PyConfig.pythonpath_env
  - 当前工作目录：用于获取绝对路径
  - PATH 环境变量用于获取程序的完整路径 (来自 PyConfig.program_name)
  - __PYVENV_LAUNCHER__ 环境变量
  - （仅限 Windows only） 注册表 HKEY_CURRENT_USER 和 HKEY_LOCAL_MACHINE 的 "SoftwarePythonPythonCoreX.YPythonPath" 项下的应用程序目录（其中 X.Y 为 Python 版本）。
- 路径配置输出字段：
  - PyConfig.base_exec_prefix
  - PyConfig.base_executable
  - PyConfig.base_prefix
  - PyConfig.exec_prefix
  - PyConfig.executable
  - PyConfig.module_search_paths_set, PyConfig.module_search_paths
  - PyConfig.prefix

如果至少有一个“输出字段”未被设置，Python 就会计算路径配置来填充未设置的字段。 如果 module_search_paths_set 等于 0，则 module_search_paths 将被覆盖并且 module_search_paths_set 将被设置为 1。

通过显式地设置上述所有路径配置输出字段可以完全忽略计算默认路径配置的函数。 即使字符串不为空也会被视为已设置。 如果 `module_search_paths_set` 被设为 `1` 则 `module_search_paths` 会被视为已设置。 在这种情况下，`module_search_paths` 将不加修改地被使用。

将 `pathconfig_warnings` 设为 `0` 以便在计算路径配置时抑制警告（仅限 Unix，Windows 不会记录任何警告）。

如果 `base_prefix` 或 `base_exec_prefix` 字段未设置，它们将分别从 `prefix` 和 `exec_prefix` 继承其值。

`Py_RunMain()` 和 `Py_Main()` 将修改 `sys.path`:

- 如果 `run_filename` 已设置并且是一个包含 `__main__.py` 脚本的目录，则会将 `run_filename` 添加到 `sys.path` 的开头。
- 如果 `isolated` 为零:
  - 如果设置了 `run_module`，则将当前目录添加到 `sys.path` 的开头。 如果无法读取当前目录则不执行任何操作。
  - 如果设置了 `run_filename`，则将文件名的目录添加到 `sys.path` 的开头。
  - 在其他情况下，则将一个空字符串添加到 `sys.path` 的开头。

如果 `site_import` 为非零值，则 `sys.path` 可通过 `site` 模块修改。 如果 `user_site_directory` 为非零值且用户的 site-package 目录存在，则 `site` 模块会将用户的 site-package 目录附加到 `sys.path`。

路径配置会使用以下配置文件:

- `pyvenv.cfg`
- `._pth` 文件 (例如: `python._pth`)
- `pybuilddir.txt` (仅Unix)

如果存在 `._pth` 文件:

- 将 `isolated` 设为 `1`。
- 将 `use_environment` 设为 `0`。
- 将 `site_import` 设为 `0`。
- 将 `safe_path` 设为 `1`。

如果未设置 `home` 并且 `executable` 的目录或其父目录下存在 `pyvenv.cfg` 文件，则 `prefix` 和 `exec_prefix` 将被设为该目录，`base_prefix` 和 `base_exec_prefix` 仍会保持其原值，即指向基础安装目录。 请参阅 虚拟环境 了解详情。

使用 `__PYVENV_LAUNCHER__` 环境变量来设置 `PyConfig.base_executable`。

> *在 3.14 版本发生变更:* 现在 `prefix` 和 `exec_prefix` 会被设为 `pyvenv.cfg` 目录。 在之前版本中这是由 `site` 完成的，因而会受 `-S` 影响。

Py_GetArgcArgv()

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

在 Python 修改原始命令行参数之前，获取这些参数。

另请参阅 `PyConfig.orig_argv` 成员。

延迟主模块的执行

在某些嵌入式用例中，可能会需要将解释器初始化和主模块的执行分隔开来。

这种分隔可通过在初始化期间将 `PyConfig.run_command` 设为空字符串来达成（以避免解释器进入交互提示符），然后再使用 `__main__.__dict__` 作为全局命名空间来执行所需的主模块代码。