自由线程的 C API 扩展支持

发行版本 *3.13.6*

Guido van Rossum and the Python development team

八月 08, 2025

Python Software Foundation
Email: docs@python.org

Contents

从 3.13 发布起，CPython 通过 free threading 配置项引入了禁用 global interpreter lock (GIL) 的实验性支持。这份文档阐述了如何修改 C API 扩展以支持自由线程。

1 在 C 中识别自由线程构建

CPython C API 提供了 `Py_GIL_DISABLED` 宏，它在自由线程构建中被定义为 `1`，而在常规构建中未被定义。你可以使用它让代码仅在自由线程构建中运行：

```
#ifdef Py_GIL_DISABLED
/* 仅在自由线程构建版中运行的代码 */
#endif
```

> **备注**
>
> 在 Windows 上，该宏不会被自动定义，而必须在构建时向编译器指明。`sysconfig.get_config_var()` 函数可被用来确定当前运行的解释器是否定义了该宏。

2 模块初始化

扩展模块需要明确指明它们支持在禁用 GIL 的情况下运行；否则导入扩展模块时会引发警告，并在运行时启用 GIL。

取决于扩展使用多阶段还是单阶段初始化，有两种方式指明扩展模块支持在 GIL 禁用的情况下运行。

2.1 多阶段初始化

使用多阶段初始化（例如 `PyModuleDef_Init()`）的扩展应该在模块定义中添加 `Py_mod_gil` 槽位。如果你的扩展需要支持更老版本的 CPython，请检查 `PY_VERSION_HEX` 以保护槽位。

```
static struct PyModuleDef_Slot module_slots[] = {
    ...
#if PY_VERSION_HEX >= 0x030D0000
    {Py_mod_gil, Py_MOD_GIL_NOT_USED},
#endif
    {0, NULL}
};

static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    .m_slots = module_slots,
    ...
};
```

2.2 单阶段初始化

使用单阶段初始化（即 `PyModule_Create()`）的扩展应该调用 `PyUnstable_Module_SetGIL()` 来表明它们支持在禁用 GIL 的情况下运行。该函数只在自由线程构建中被定义，因此应使用 `#ifdef Py_GIL_DISABLED` 来保护调用，以避免在常规构建中出现编译错误。

```
static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    ...
};

PyMODINIT_FUNC
PyInit_mymodule(void)
{
    PyObject *m = PyModule_Create(&moduledef);
    if (m == NULL) {
        return NULL;
    }
#ifdef Py_GIL_DISABLED
    PyUnstable_Module_SetGIL(m, Py_MOD_GIL_NOT_USED);
#endif
    return m;
}
```

3 通用 API 指南

大多数 C API 是线程安全的，但是也存在例外。

- **结构字段**：如果 Python C API 对象或结构的字段可能被并行修改，那么直接访问这些字段不是线程安全的。
- **宏**：`PyList_GET_ITEM` 以及 `PyList_SET_ITEM` 等访问宏不进行错误检查和上锁，因而容器对象可能被并行修改时它们不是线程安全的。
- **借入引用**：返回 借入引用的 C API 函数如果引用内容可能被并行修改，那么它不是线程安全的。详见借入引用。

3.1 容器相关的线程安全

`PyListObject`, `PyDictObject` 及 `PySetObject` 等容器在自由线程构建中执行内部上锁机制，例如 `PyList_Append()` 在追加对象前会对列表上锁。

PyDict_Next

一个值得注意的例外是 `PyDict_Next()`，它不会锁定目录。在迭代目录时如果该目录可能被并发地修改那么你应当使用 `Py_BEGIN_CRITICAL_SECTION` 来保护它:

```
Py_BEGIN_CRITICAL_SECTION(dict);
PyObject *key, *value;
Py_ssize_t pos = 0;
while (PyDict_Next(dict, &pos, &key, &value)) {
    ...
}
Py_END_CRITICAL_SECTION();
```

4 借入引用

有些 C API 函数返回 borrowed references。如果引用内容可能被并行修改，那么这些 API 不是线程安全的。例如，如果列表可能被并行修改，那么使用 `PyList_GetItem()` 是不安全的。

下表列出了一些返回借入引用的 API 及它们返回 强引用的替代版本。

借入引用 API	强引用 API
`PyList_GetItem()`	`PyList_GetItemRef()`
`PyList_GET_ITEM()`	`PyList_GetItemRef()`
`PyDict_GetItem()`	`PyDict_GetItemRef()`
`PyDict_GetItemWithError()`	`PyDict_GetItemRef()`
`PyDict_GetItemString()`	`PyDict_GetItemStringRef()`
`PyDict_SetDefault()`	`PyDict_SetDefaultRef()`
`PyDict_Next()`	无 (参见*PyDict_Next*)
`PyWeakref_GetObject()`	`PyWeakref_GetRef()`
`PyWeakref_GET_OBJECT()`	`PyWeakref_GetRef()`
`PyImport_AddModule()`	`PyImport_AddModuleRef()`

返回借用引用的 API 不一定都有问题。例如，`PyTuple_GetItem()` 是安全的，因为元组是不可变的。同样，上述 API 的使用不一定都有问题。例如，`PyDict_GetItem()` 通常用于解析函数调用中的关键字参数字典；这些关键字参数字典实际上是私有（其他线程无法访问）的，因此在这种情况下使用借入引用是安全的。

上述函数中有的是在 Python 3.13 中添加的。在旧 Python 版本上您可以使用提供这些函数实现的 pythoncapi-compat 包。

5 内存分配 API

Python 的内存管理 C API 提供了三个不同 分配域的函数: ”raw”, ”mem” 和”object”。为了保证线程安全，自由线程构建版要求只有 Python 对象使用 object 域来分配，并且所有 Python 对象都应使用该域来分配。这不同于之前的 Python 版本，因为在此之前这只是一个最佳实践而不是硬性要求。

> **备注**
>
> 搜索 `PyObject_Malloc()` 在您的扩展中的使用，并检查分配的内存是否用于 Python 对象。使用 `PyMem_Malloc()` 来分配缓冲区，而不是 `PyObject_Malloc()`。

6 线程状态与 GIL API

Python 提供了一系列函数和宏来管理线程状态和 GIL，例如：

- `PyGILState_Ensure()` 与 `PyGILState_Release()`
- `PyEval_SaveThread()` 与 `PyEval_RestoreThread()`
- `Py_BEGIN_ALLOW_THREADS` 与 `Py_END_ALLOW_THREADS`

即使 GIL 被禁用，仍应在自由线程构建中使用这些函数管理线程状态。例如，如果在 Python 之外创建线程，则必须在调用 Python API 前调用 `PyGILState_Ensure()`，以确保线程具有有效的 Python 线程状态。

你应该继续在阻塞操作（如输入/输出或获取锁）前调用 `PyEval_SaveThread()` 或 `Py_BEGIN_ALLOW_THREADS`，以允许其他线程运行 循环垃圾回收器。

7 保护内部扩展状态

您的扩展可能有以前受 GIL 保护的内部状态。您可能需要上锁来保护内部状态。具体方法取决于您的扩展，但一些常见的模式包括：

- **缓存**：全局缓存是共享状态的常见来源。如果缓存对性能并不重要，可考虑使用锁来保护缓存，或在自由线程构建中禁用缓存。
- **全局状态**：全局状态可能需要用锁保护或移至线程本地存储。C11 和 C++11 提供了 `thread_local` 或 `_Thread_local` 用于 线程本地存储。

8 为自由线程构建进行扩展构建

C API 扩展需要专门为自由线程构建进行构建。构建的 wheel、共享库和二进制文件用后缀 `t` 指示。

- pypa/manylinux 支持后缀为 `t` 的自由线程构建，如 `python3.13t`。
- 如果你设置了 cpython-freethreading 的 CIBW_ENABLE 则 pypa/cibuildwheel 将支持自由线程构建版。

8.1 受限的 C API 与稳定 ABI

自由线程构建目前不支持受限 C API 或稳定 ABI。如果当前您使用 setuptools 来构建您的扩展，并且设置了 `py_limited_api=True`，您可以使用 `py_limited_api=not sysconfig.get_config_var("Py_GIL_DISABLED")` 在使用自由线程构建进行构建时不使用受限 API。

> **备注**
>
> 您需要为自由线程构建单独构建 wheel。如果您当前使用稳定 ABI，则可以继续构建适用于多个非自由线程 Python 版本的单个 wheel。

8.2 Windows

由于 Windows 官方安装程序的限制，从源代码构建扩展时需要手动定义 `Py_GIL_DISABLED=1`。

参见

Porting Extension Modules to Support Free-Threading: 一份由社区维护的针对扩展开发者的移植指南。