自由线程的 C API 扩展支持

从 3.13 发布版开始，CPython 通过名为 free threading 的配置引入了对于运行时禁用 global interpreter lock (GIL) 的支持。 这份文档描述了如何调整 C API 扩展以支持自由线程。

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。

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。

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,
    ...
};
```

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;
}
```

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**: 访问器宏如 PyList_GET_ITEM, PyList_SET_ITEM，以及 PySequence_Fast_GET_SIZE 这样使用由 PySequence_Fast() 返回的对象的宏不会进行任何错误检查或加锁。 当容器对象可能被并行修改时这些宏不是线程安全的。
- **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。

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();
```

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()`
`PyCell_GET()`	`PyCell_Get()`

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。

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()`。

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。

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。

关键节

在自由线程构建中，CPython提供了一种称为“临界区”的机制来保护原本由GIL保护的数据。虽然扩展作者可能不会直接与内部临界区实现交互，但在使用某些C API函数或在自由线程构建中管理共享状态时，理解它们的行为是至关重要的。

什么是临界区？

从概念上讲，临界区充当建立在简单互斥锁之上的死锁避免层。 每个线程维护一个活动临界区堆栈。 当线程需要获取与临界区相关的锁时（例如，隐式调用线程安全的 C API 函数时，如 `PyDict_SetItem()`，或显式使用宏），它会尝试获取底层互斥锁。

使用临界区

使用临界区的主要API有：

- `Py_BEGIN_CRITICAL_SECTION` 和 `Py_END_CRITICAL_SECTION` - 用于锁定单个对象
- `Py_BEGIN_CRITICAL_SECTION2` 和 `Py_END_CRITICAL_SECTION2` - 用于同时锁定两个对象

这些宏必须成对使用，并且必须出现在同一个C作用域中，因为它们建立了一个新的局部作用域。这些宏在非自由线程构建中是无操作的，因此可以安全地将它们添加到需要支持两种构建类型的代码中。

临界区的一个常见用途是在访问对象的内部属性时锁定对象。例如，如果扩展类型有一个内部计数字段，你可以在读取或写入该字段时使用临界区:

```
// 读取计数，返回对内部计数值的新引用
PyObject *result;
Py_BEGIN_CRITICAL_SECTION(obj);
result = Py_NewRef(obj->count);
Py_END_CRITICAL_SECTION();
return result;

// 写入计数，从new_count中获取引用
Py_BEGIN_CRITICAL_SECTION(obj);
obj->count = new_count;
Py_END_CRITICAL_SECTION();
```

临界区如何运作

与传统锁不同，临界区不能保证在其整个持续时间内的独占访问。如果线程在持有临界区时阻塞（例如，通过获取另一个锁或执行I/O），则临界区被暂时挂起——所有锁被释放——然后在阻塞操作完成时恢复。

此行为类似于当线程执行阻塞型调用时 GIL 的行为。 主要的区别在于:

- 关键节的运作是基于每个对象的而不是全局的
- 关键节遵循设置于每个线程内部的纪律栈（"begin" 和 "end" 宏将应用该纪律栈，因为它们必须成对出现并位于相同作用域中）
- 关键节会针对潜在的阻塞型操作自动释放和重新获取锁

避免死锁

关键节通过两种方式帮助避免死锁:

1. 如果一个线程试图获取某个已被其他线程持有的锁，它会先挂起该线程的所有关键节，临时释放它们的锁。
2. 当阻塞型操作完成时，只有最顶端的关键节会被首先重新获取

这意味着你不能依赖嵌套的关键节来同时锁定多个对象，因为内层的关键节可能挂起外层的关键节。 作为替代，请使用 `Py_BEGIN_CRITICAL_SECTION2` 来同时锁定两个对象。

注意，上面描述的锁只是基于 `PyMutex` 的锁。 临界区实现并不知道或影响其他可能正在使用的锁定机制，比如 POSIX 互斥锁。 还要注意，当任何 `PyMutex` 阻塞时会导致临界区被挂起，只有属于临界区的互斥锁才会被释放。 如果 `PyMutex` 在没有临界区的情况下使用，它不会被释放，因此不会得到同样的死锁避免。

重要考量

- 临界区可以暂时释放它们的锁，允许其他线程修改受保护的数据。在进行可能阻塞的操作之后，要谨慎地假设数据的状态。
- 因为锁可以临时释放（挂起），所以进入临界区并不能保证在整个临界区期间对受保护资源的独占访问。如果临界区内的代码调用另一个阻塞函数（例如，获取另一个锁、执行阻塞I/O），则该

线程通过临界区持有的所有锁将被释放。这类似于在阻塞调用期间释放GIL的方式。

- 在任何给定时间，只有与最近进入（最顶部）的临界区相关的锁才能保证被持有。外部嵌套临界区的锁可能已经挂起。
- 使用这些API最多可以同时锁定两个对象。如果你需要锁定更多的对象，你需要调整你的代码。
- 虽然如果你尝试锁定同一个对象两次，临界区不会死锁，但是对于这种用例，它们的效率不如专门构建的可重入锁。
- 当使用 `Py_BEGIN_CRITICAL_SECTION2` 时，对象的顺序不影响正确性（实现处理死锁避免），但始终以一致的顺序锁定对象是良好的实践。
- 请记住，临界区宏主要用于保护对 *Python对象* 的访问，这些对象可能涉及易受上述死锁场景影响的内部CPython操作。为了保护纯粹的内部扩展状态，标准互斥体或其他同步原语可能更合适。

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。

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。

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。