
Suporte a extensões da API C para threads livres

Release 3.13.2

Guido van Rossum and the Python development team

março 25, 2025

Python Software Foundation
Email: docs@python.org

Sumário

1	Identificando a construção com threads livres em C	1
2	Inicialização de módulo	2
2.1	Inicialização multifásica	2
2.2	Inicialização monofásica	2
3	Diretrizes gerais de API	3
3.1	Segurança de threads de contêineres	3
4	Referências emprestadas	3
5	APIs de alocação de memória	4
6	APIs da GIL e de estado de threads	4
7	Protegendo o estado interno das extensões	4
8	Construindo extensões para a construção com threads livres	4
8.1	API C limitada e ABI estável	5
8.2	Windows	5

A partir da versão 3.13, o CPython tem suporte experimental para execução com a trava global do interpretador (GIL) desabilitada em uma configuração chamada threads livres. Este documento descreve como adaptar extensões da API C para ter suporte a threads livres.

1 Identificando a construção com threads livres em C

A API C do CPython expõe a macro `Py_GIL_DISABLED`: na construção com threads livres ela está definida como 1, e na construção regular ela não está definida. Você pode usá-la para ativar o código que é executado apenas na construção de threads livres:

```
#ifdef Py_GIL_DISABLED
/* código que só vai ser executado na construção de threads livres */
#endif
```

2 Inicialização de módulo

Os módulos de extensão precisam indicar explicitamente que têm suporte à execução com a GIL desabilitada; caso contrário, importar a extensão levantará um aviso e ativará a GIL em tempo de execução.

Há duas maneiras de indicar que um módulo de extensão tem suporte à execução com a GIL desabilitada, dependendo se a extensão usa inicialização monofásica ou multifásica.

2.1 Inicialização multifásica

Extensões que usam inicialização multifásica (ou seja, `PyModuleDef_Init()`) devem adicionar um slot `Py_mod_gil` na definição do módulo. Se sua extensão tem suporte a versões mais antigas do CPython, você deve proteger o slot com uma verificação de `PY_VERSION_HEX`.

```
static struct PyModuleDef_Slot module_slots[] = {
    ...
#ifdef 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 Inicialização monofásica

Extensões que usam inicialização monofásica (ou seja, `PyModule_Create()`) devem chamar `PyUnstable_Module_SetGIL()` para indicar que têm suporte à execução com a GIL desabilitada. A função é definida apenas na construção com threads livres, então você deve proteger a chamada com `#ifdef Py_GIL_DISABLED` para evitar erros de compilação na construção regular.

```
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 Diretrizes gerais de API

A maior parte da API C é segura para thread, mas há algumas exceções.

- **Campos de structs:** acessar campos em objetos ou structs da API C do Python diretamente não é seguro para thread se o campo puder ser modificado simultaneamente.
- **Macros:** Macros de acesso como `PyList_GET_ITEM` e `PyList_SET_ITEM` não executam nenhuma verificação por erros e nenhum travamento. Essas macros não são seguras para thread se o objeto contêiner puder ser modificado simultaneamente.
- **Referências emprestadas:** As funções da API C que retornam referências emprestadas podem não ser seguras para thread se o objeto que as contém for modificado simultaneamente. Veja a seção sobre *referências emprestadas* para mais informações.

3.1 Segurança de threads de contêineres

Contêineres como `PyListObject`, `PyDictObject` e `PySetObject` executam uma trava interna na construção com threads livres. Por exemplo, `PyList_Append()` irá travar a lista antes de anexar um item.

`PyDict_Next`

Uma exceção notável é `PyDict_Next()`, que não trava o dicionário. Você deve usar `Py_BEGIN_CRITICAL_SECTION` para proteger o dicionário enquanto itera sobre ele se o dicionário puder ser modificado simultaneamente:

```
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 Referências emprestadas

Algumas funções da API C retornam referências emprestadas. Essas APIs não são seguras para thread se o objeto que as contém for modificado simultaneamente. Por exemplo, não é seguro usar `PyList_GetItem()` se a lista puder ser modificada simultaneamente.

A tabela a seguir lista algumas APIs de referências emprestadas e suas substituições que retornam referências fortes.

API de referências emprestadas	API de referências fortes
<code>PyList_GetItem()</code>	<code>PyList_GetItemRef()</code>
<code>PyDict_GetItem()</code>	<code>PyDict_GetItemRef()</code>
<code>PyDict_GetItemWithError()</code>	<code>PyDict_GetItemRef()</code>
<code>PyDict_GetItemString()</code>	<code>PyDict_GetItemStringRef()</code>
<code>PyDict_SetDefault()</code>	<code>PyDict_SetDefaultRef()</code>
<code>PyDict_Next()</code>	nenhuma (veja <i>PyDict_Next</i>)
<code>PyWeakref_GetObject()</code>	<code>PyWeakref_GetRef()</code>
<code>PyWeakref_GET_OBJECT()</code>	<code>PyWeakref_GetRef()</code>
<code>PyImport_AddModule()</code>	<code>PyImport_AddModuleRef()</code>

Nem todas as APIs que retornam referências emprestadas são problemáticas. Por exemplo, `PyTuple_GetItem()` é segura porque as tuplas são imutáveis. Da mesma forma, nem todos os usos das APIs acima são problemáticos. Por exemplo, `PyDict_GetItem()` é frequentemente usado para analisar dicionários de argumentos nomeados em chamadas de função; esses dicionários de argumentos nomeados efetivamente privados (não acessíveis por outros threads), portanto, usar referências emprestadas nesse contexto é seguro.

Algumas dessas funções foram adicionadas no Python 3.13. Você pode usar o pacote [pythoncapi-compat](#) para fornecer implementações dessas funções para versões mais antigas do Python.

5 APIs de alocação de memória

A API C de gerenciamento de memória do Python fornece funções em três domínios de alocação diferentes: “raw”, “mem” e “object”. Para segurança de thread, a construção de threads livres requer que apenas objetos Python sejam alocados usando o domínio do objeto e que todos os objetos Python sejam alocados usando esse domínio. Isso difere das versões anteriores do Python, onde era apenas uma melhor prática e não um requisito estrito.

Nota

Procure usos de `PyObject_Malloc()` em sua extensão e verifique se a memória alocada é usada para objetos Python. Use `PyMem_Malloc()` para alocar buffers em vez de `PyObject_Malloc()`.

6 APIs da GIL e de estado de threads

Python fornece um conjunto de funções e macros para gerenciar o estado de threads e a GIL, como:

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

Estas funções ainda devem ser usadas na construção com threads livres para gerenciar o estado de threads mesmo quando a GIL está desabilitada. Por exemplo, se você criar uma thread fora do Python, você deve chamar `PyGILState_Ensure()` antes de chamar a API Python para garantir que a thread tenha um estado de thread válido para o Python.

Você deve continuar a chamar `PyEval_SaveThread()` ou `Py_BEGIN_ALLOW_THREADS` em torno de operações de travamento, como E/S ou aquisições de trava, para permitir que outras threads executem o coletor de lixo cíclico.

7 Protegendo o estado interno das extensões

Sua extensão pode ter um estado interno que foi previamente protegido pela GIL. Talvez seja necessário adicionar uma trava para proteger esse estado. A abordagem dependerá da sua extensão, mas alguns padrões comuns incluem:

- **Caches:** caches globais são uma fonte comum de estado compartilhado. Considere usar uma trava para proteger o cache ou desativá-lo na construção com threads livres se o cache não for crítico para o desempenho.
- **Estado global:** o estado global pode precisar ser protegido por uma trava ou movido para o armazenamento local do thread. C11 e C++11 fornecem `thread_local` ou `_Thread_local` para [armazenamento local de thread](#).

8 Construindo extensões para a construção com threads livres

As extensões da API C precisam ser construídas especificamente para a construção com threads livres. As wheels, bibliotecas compartilhadas e binários são indicados por um sufixo `t`.

- [pypa/manylinux](#) oferece suporte à construção com threads livres, com o sufixo `t`, como `python3.13t`.
- [pypa/cibuildwheel](#) tem suporte à construção com threads livres se você definir `CIBW_FREE_THREADED_SUPPORT`.

8.1 API C limitada e ABI estável

A construção com threads livres não tem suporte atualmente à API C limitada ou à ABI estável. Se você usar `setuptools` para construir sua extensão e atualmente definir `py_limited_api=True`, você pode usar `py_limited_api=not sysconfig.get_config_var("Py_GIL_DISABLED")` para não usar a API limitada ao construir com a construção com threads livres.

Nota

Você precisará construir wheels separadas especificamente para a construção com threads livres. Se você usa atualmente a ABI estável, poderá continuar a construir uma única wheel para várias versões do Python sem threads livres.

8.2 Windows

Devido a uma limitação do instalador oficial do Windows, você precisará definir manualmente `Py_GIL_DISABLED=1` ao construir extensões a partir do código-fonte.

Ver também

[Portando módulos de extensão para oferecer suporte a threads livres \(em inglês\)](#): Um guia de portabilidade mantido pela comunidade para autores de extensões.