Isolando módulos de extensão

Release 3.14.0a1

Guido van Rossum and the Python development team

novembro 05, 2024

Python Software Foundation
Email: docs@python.org

Sumário

1 Quem deveria ler isto **2**

2 Informações preliminares **2**
2.1 Entra o estado por módulo . . . 2
2.2 Objetos de módulo isolados . . . 2
2.3 Casos particulares surpreendentes . . . 3

3 Fazendo módulos seguros com múltiplos interpretadores **3**
3.1 Gerenciando estado global . . . 3
3.2 Gerenciando estado por módulo . . . 3
3.3 Exclusão voluntária: limitando a um objeto de módulo por processo . . . 4
3.4 Acesso ao estado de módulo a partir de funções . . . 4

4 Tipos no heap **5**
4.1 Mudando tipos estáticos para tipos no heap . . . 5
4.2 Definindo tipos no heap . . . 5
4.3 Protocolo de recolhimento de lixo . . . 6
4.4 Acessando o estado do módulo a partir de classes . . . 7
4.5 Acesso ao estado do módulo a partir de métodos regulares . . . 8
4.6 Acesso ao estado do módulo a partir de métodos slot, getters e setters . . . 9
4.7 Tempo de vida do estado do módulo . . . 9

5 Problemas em aberto **9**
5.1 Escopo por classe . . . 10
5.2 Conversão sem perdas para tipos no heap . . . 10

Resumo

Tradicionalmente, o estado que pertence a módulos de extensão do Python era mantido em variáveis `static` em C, que têm escopo em todo o processo. Este documento descreve problemas de tal estado por processo e apresenta um modo mais seguro: o estado por módulo.

O documento também descreve como migrar para o uso do estado por módulo onde for possível. Essa transição envolve alocar espaço para este estado, potencialmente trocar tipos estáticos por tipos no heap, e—talvez o mais importante—acessar o estado por módulo a partir do código.

1 Quem deveria ler isto

Este guia é escrito para mantenedores de extensões que usam a API C que desejam torná-las mais seguras para o uso em aplicações onde o Python em si é usado como uma biblioteca.

2 Informações preliminares

Um *interpretador* é o contexto no qual o código Python é executado. Ele contém estado de configuração (por exemplo o caminho de importação) e de tempo de execução (por exemplo o conjunto de módulos importados).

O Python provê suporte para executar múltiplos interpretadores em um processo. Dois casos devem ser considerados—usuários podem executar interpretadores:

- em sequência, com vários ciclos de `Py_InitializeEx()`/`Py_FinalizeEx()`, e
- em paralelo, gerenciando "sub-interpretadores" usando `Py_NewInterpreter()`/`Py_EndInterpreter()`.

Ambos os casos (e combinações deles) são muito úteis ao embutir o Python em uma biblioteca. Bibliotecas geralmente não devem fazer suposições sobre a aplicação que as usa, o que inclui supor um "interpretador Python principal" para o processo inteiro.

Historicamente, módulos de extensão do Python não lidam bem com este caso de uso. Muitos módulos de extensão (e até alguns módulos da biblioteca padrão) usam estado global *por processo*, uma vez que variáveis `static` do C são extremamente fáceis de se usar. Assim, dados que deveriam ser específicos para um interpretador acabam sendo compartilhados entre interpretadores. A menos que o desenvolvedor da extensão tenha cuidado, é muito fácil criar casos particulares que acabam quebrando o processo quando um módulo é carregado em mais de um interpretador no mesmo processo.

Infelizmente, não é fácil fazer o estado por interpretador. Autores de extensões tendem a não ter múltiplos interpretadores em mente ao desenvolver, e no momento é complicado testar este comportamento.

2.1 Entra o estado por módulo

Ao invés de focar no estado por interpretador, a API C do Python está evoluindo para melhor suportar o estado *por módulo*, que é mais granular. Isso significa que dados a nível do C devem estar atrelados a um *objeto de módulo*. Cada interpretador cria o seu próprio objeto de módulo, garantindo assim a separação dos dados. Para testar o isolamento, múltiplos objetos de módulo correspondentes a uma única extensão podem até ser carregados em um único interpretador.

O estado por módulo fornece um modo fácil de pensar sobre tempos de vida e posse de recursos: o módulo de extensão será inicializado quando um objeto de módulo for criado, e limpado quando ele for liberado. Nesse sentido, um módulo funciona como qualquer outro `PyObject*`; não há ganchos "de desligamento do interpretador" a serem considerados—ou esquecidos.

Note que há casos de uso para diferentes tipos de "objetos globais": estado por processo, por interpretador, por thread, ou por tarefa. Com o estado por módulo como padrão, as outras formas ainda são possíveis, mas devem ser tratadas como casos excepcionais: se você precisar delas, você deve tomar cuidados adicionais e escrever mais testes. (Note que este guia não cobre tais medidas.)

2.2 Objetos de módulo isolados

O ponto chave de se manter em mente ao desenvolver um módulo de extensão é que vários objetos de módulo podem ser criados a partir de uma única biblioteca compartilhada. Por exemplo:

```
>>> import sys
>>> import binascii
>>> old_binascii = binascii
>>> del sys.modules['binascii']
>>> import binascii  # cria um novo objeto do módulo
>>> old_binascii == binascii
False
```

Como regra geral, os dois módulos devem ser completamente independentes. Todos os objetos e o estado específicos do módulo devem ser encapsulados no objeto de módulo, não devem ser compartilhados com outros objetos de módulo, e devem ser limpados quando o objeto de módulo for desalocado. Uma vez que esta é somente uma regra geral, exceções são possíveis (veja *Gerenciando estado global*), mas elas necessitam mais cuidado e atenção a casos especiais.

Enquanto alguns módulos funcionariam bem com restrições menos rigorosas, isolar os módulos torna mais fácil definir expectativas claras e diretrizes que dão certo em uma variedade de casos de uso.

2.3 Casos particulares surpreendentes

Note que módulos isolados criam alguns casos particulares que podem acabar surpreendendo. O mais notável é que, tipicamente, cada objeto de módulo não vai compartilhar as suas classes e exceções com outros módulos similares. Continuando o *exemplo acima*, note `old_binascii.Error` e `binascii.Error` são objetos separados. No código a seguir, a exceção *não* é capturada:

```
>>> old_binascii.Error == binascii.Error
False
>>> try:
...     old_binascii.unhexlify(b'qwertyuiop')
... except binascii.Error:
...     print('boo')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
binascii.Error: Non-hexadecimal digit found
```

Isso é esperado. Repare que módulos Python-puro se comportam do mesmo jeito: isso é parte de como o Python funciona.

O objetivo é fazer módulos de extensão seguros no nível do C, e não fazer gambiarras se comportarem de forma intuitiva. Modificar o `sys.modules` "manualmente" conta como uma gambiarra.

3 Fazendo módulos seguros com múltiplos interpretadores

3.1 Gerenciando estado global

Às vezes, o estado associado a um módulo Python não é específico àquele módulo, mas ao processo inteiro (ou a alguma outra coisa "mais global" que um módulo). Por exemplo:

- O módulo `readline` gerencia *o* terminal.
- Um módulo executando em uma placa de circuito quer controlar *o* componente LED.

Nestes casos, o módulo Python deve prover *acesso* ao estado global, ao invés de *possuí-lo*. Se possível, escreva o módulo de forma que múltiplas cópias dele possam acessar o estado independentemente (junto com outras bibliotecas, sejam elas do Python ou de outras linguagens). Se isso não for possível, considere usar travas explícitas.

Se for necessário usar estado global para o processo, o jeito mais simples de evitar problemas com múltiplos interpretadores é prevenir explicitamente que o módulo seja carregado mais de uma vez por processo—veja *Exclusão voluntária: limitando a um objeto de módulo por processo*.

3.2 Gerenciando estado por módulo

Para usar estado por módulo, use inicialização multifásica de módulos de extensão. Assim, você sinaliza que o seu módulo suporta múltiplos interpretadores corretamente.

Defina `PyModuleDef.m_size` como um número positivo *N* para requerer *N* bytes de armazenamento local para o módulo. Geralmente, *N* será o tamanho de alguma `struct` específica para o módulo, a qual pode guardar todo o estado a nível de C do módulo. Em particular, é nela que você deve colocar ponteiros para classes (incluindo

exceções, mas excluindo tipos estáticos) e configurações (por exemplo, `field_size_limit` no módulo `csv`) que o código C precisa para funcionar.

> **Nota**
>
> Outra opção é guardar estado no `__dict__` do módulo, mas você deve evitar quebrar quando usuários modificarem o `__dict__` a partir do código Python. Isso geralmente significa verificar tipos e erros no nível do C, o que é fácil de ser feito incorretamente e difícil de se testar suficientemente.
>
> Entretanto, se o estado do módulo não for necessário para o código C, guardá-lo somente no `__dict__` é uma boa ideia.

Se o estado do módulo inclui ponteiros para `PyObject`, o objeto de módulo deve conter referências a tais objetos e implementar os ganchos a nível de módulo `m_traverse`, `m_clear` e `m_free`. Eles funcionam como os `tp_traverse`, `tp_clear` e `tp_free` de uma classe. Adicioná-los requer algum trabalho e torna o código mais longo; é o preço de módulos que podem ser descarregados de forma limpa.

Um exemplo de módulo com estado por módulo está disponível atualmente como xxlimited; há um exemplo de inicialização do módulo no final do arquivo.

3.3 Exclusão voluntária: limitando a um objeto de módulo por processo

Um `PyModuleDef.m_size` não-negativo sinaliza que um módulo admite múltiplos interpretadores corretamente. Se este ainda não é o caso para o seu módulo, you pode explicitamente torná-lo carregável somente uma vez por processo. Por exemplo:

```
static int loaded = 0;

static int
exec_module(PyObject* module)
{
    if (loaded) {
        PyErr_SetString(PyExc_ImportError,
                        "cannot load module more than once per process");
        return -1;
    }
    loaded = 1;
    // ... o resto da inicialização
}
```

3.4 Acesso ao estado de módulo a partir de funções

É trivial acessar o estado a partir de funções a nível do módulo. Funções recebem o objeto de módulo como o primeiro argumento; para extrair o estado, você pode usar `PyModule_GetState`:

```
static PyObject *
func(PyObject *module, PyObject *args)
{
    my_struct *state = (my_struct*)PyModule_GetState(module);
    if (state == NULL) {
        return NULL;
    }
    // ... o resto da lógica
}
```

> **Nota**

`PyModule_GetState` pode retornar `NULL` sem definir uma exceção se não houver estado de módulo, ou seja se `PyModuleDef.m_size` for zero. No seu próprio módulo, você controla o `m_size`, de forma que isso é fácil de prevenir.

4 Tipos no heap

Tradicionalmente, tipos definidos em C são *estáticos*; isto é, estruturas `static PyTypeObject` definidas diretamente em código e inicializadas usando `PyType_Ready()`.

Tais tipos são necessariamente compartilhados pelo processo inteiro. Compartilhá-los entre objetos de módulo requer atenção a qualquer estado que eles possuam ou acessem. Para limitar potenciais problemas, tipos estáticos são imutáveis a nível do Python: por exemplo, você não pode atribuir `str.meuatributo = 123`.

Detalhes da implementação do CPython: Não há problema em compartilhar objetos verdadeiramente imutáveis entre interpretadores, desde que através deles não seja possível acessar outros objetos mutáveis. De toda forma, no CPython, todo objeto Python tem um detalhe de implementação mutável: o contador de referências. Mudanças no refcount são protegidas pelo GIL. Logo, todo código que compartilha um objeto Python entre interpretadores depende implicitamente do atual GIL do CPython (que é global a nível de processo).

Por ser imutável e global no processo, um tipo estático não pode acessar o estado do "seu" módulo. Se um método de tal tipo precisar de acesso ao estado do módulo, o tipo precisa ser convertido para um *tipo alocado no heap*, ou, abreviando, *tipo no heap*. Tipos no heap correspondem mais fielmente a classes criadas pela instrução `class` do Python.

Para módulos novos, usar tipos no heap por padrão é uma boa regra geral.

4.1 Mudando tipos estáticos para tipos no heap

Tipos estáticos podem ser convertidos para tipos no heap, mas note que a API de tipos no heap não foi projetada para conversão "sem perda" de tipos estáticos—isto é, para criar um tipo que funciona exatamente como um dado tipo estático. Então, ao reescrever a definição de classe em uma nova API, é provável que você altere alguns detalhes sem querer (por exemplo, se o tipo é serializável em pickle ou não, ou slots herdados). Sempre teste os detalhes que são importantes para você.

Fique atento em particular aos dois pontos a seguir (mas note the esta não é uma lista completa):

- Ao contrário de tipos estáticos, tipos no heap são mutáveis por padrão. Use o sinalizador `Py_TPFLAGS_IMMUTABLETYPE` para impedir a mutabilidade.
- Tipos no heap herdam `tp_new` por padrão, e portanto eles podem passar a ser instanciáveis a partir de código Python. Você pode impedir isso com o sinalizador `Py_TPFLAGS_DISALLOW_INSTANTIATION`.

4.2 Definindo tipos no heap

Tipos no heap podem ser criados preenchendo uma estrutura `PyType_Spec`, uma descrição ou "diagrama" de uma classe, e chamando `PyType_FromModuleAndSpec()` para construir um novo objeto classe.

Nota

Outras funções, como `PyType_FromSpec()`, também podem criar tipos no heap, mas `PyType_FromModuleAndSpec()` associa a classe ao módulo, permitindo acesso ao estado do módulo a partir dos métodos.

A classe deve em geral ser guardada *tanto* no estado do módulo (para acesso seguro a partir do C) *quanto* no `__dict__` do módulo (para acesso a partir de código Python).

4.3 Protocolo de recolhimento de lixo

Instâncias de tipos no heap contêm referências aos seus tipos. Isso garante que o tipo não é destruído antes que todas as suas instâncias sejam, mas pode resultar em ciclos de referência que precisam ser quebrados pelo coletor de lixo.

Para evitar vazamentos de memória, instâncias de tipos no heap precisam implementar o protocolo de recolhimento de lixo. Isto é, tipos no heap devem:

- Ter o sinalizador `Py_TPFLAGS_HAVE_GC`.
- Definir uma função de travessia usando `Py_tp_traverse`, que visita o tipo (por exemplo, usando `Py_VISIT(Py_TYPE(self))`).

Por favor, veja a documentação de `Py_TPFLAGS_HAVE_GC` e de `tp_traverse` para considerações adicionais.

A API para definir tipos no heap cresceu organicamente, o que resultou em um status quo no qual usá-la pode ser um pouco confuso. As seções a seguir vão lhe guiar pelos problemas mais comuns.

`tp_traverse` no Python 3.8 e anteriores

O requerimento de o `tp_traverse` visitar o tipo foi adicionado no Python 3.9. Se você suporta Python 3.8 e anteriores, a função de travessia *não* deve visitar o tipo, de forma que ela precisa ser mais complicada:

```
static int my_traverse(PyObject *self, visitproc visit, void *arg)
{
    if (Py_Version >= 0x03090000) {
        Py_VISIT(Py_TYPE(self));
    }
    return 0;
}
```

Infelizmente, o símbolo `Py_Version` foi adicionado somente no Python 3.11. Para substituí-lo, use:

- `PY_VERSION_HEX`, caso não esteja usando a ABI estável, ou
- `sys.version_info` (via `PySys_GetObject()` e `PyArg_ParseTuple()`).

Delegando a função `tp_traverse`

Se a sua função de travessia delega para a `tp_traverse` da sua classe base (ou de outro tipo), certifique-se de que `Py_TYPE(self)` seja visitado apenas uma vez. Observe que somente tipos no heap devem visitar o tipo em `tp_traverse`.

Por exemplo, se a sua função de travessia incluir:

```
base->tp_traverse(self, visit, arg)
```

…e `base` puder ser um tipo estático, então ela também precisa incluir:

```
if (base->tp_flags & Py_TPFLAGS_HEAPTYPE) {
    // uma tp_traverse do tipo heap já visitou Py_TYPE(self)
} else {
    if (Py_Version >= 0x03090000) {
        Py_VISIT(Py_TYPE(self));
    }
}
```

Não é necessário mexer na contagem de referências do tipo em `tp_new` e `tp_clear`.

Definindo `tp_dealloc`

Se o seu tipo tem uma função `tp_dealloc` customizada, ele precisa:

- chamar `PyObject_GC_UnTrack()` antes que quaisquer campos sejam invalidados, e
- decrementar o contador de referências do tipo.

Para que o tipo permaneça válido durante o `tp_free`, o refcount do tipo precisa ser decrementado *depois* de a instância ser liberada. Por exemplo:

```
static void my_dealloc(PyObject *self)
{
    PyObject_GC_UnTrack(self);
    ...
    PyTypeObject *type = Py_TYPE(self);
    type->tp_free(self);
    Py_DECREF(type);
}
```

A função `tp_dealloc` padrão faz isso, de forma que se o seu tipo *não* a substitui você não precisa se preocupar.

Evitar substituir `tp_free`

O slot `tp_free` de um tipo no heap deve ser `PyObject_GC_Del()`. Este é o padráo; não o substitua.

Evitar `PyObject_New`

Objetos rastreados pelo GC precisam ser alocados usando funções que reconheçam o GC.

Se você usaria `PyObject_New()` ou `PyObject_NewVar()`:

- Se possível, chame o slot `tp_alloc` do tipo. Isto é, troque `TYPE *o = PyObject_New(TYPE, typeobj)` por:

  ```
  TYPE *o = typeobj->tp_alloc(typeobj, 0);
  ```

  No lugar de `o = PyObject_NewVar(TYPE, typeobj, size)`, use também a forma acima, mas com `size` ao invés do `0`.

- Se isso não for possível (por exemplo, dentro de um `tp_alloc` customizado), chame `PyObject_GC_New()` or `PyObject_GC_NewVar()`:

  ```
  TYPE *o = PyObject_GC_New(TYPE, typeobj);

  TYPE *o = PyObject_GC_NewVar(TYPE, typeobj, size);
  ```

4.4 Acessando o estado do módulo a partir de classes

Dado um objeto de tipo definido com `PyType_FromModuleAndSpec()`, você pode chamar `PyType_GetModule()` para acessar o módulo associado, e então `PyModule_GetState()` para acessar o estado do módulo.

Para evitar o tedioso código de tratamento de erros de sempre, você pode combinar essas duas etapas com o `PyType_GetModuleState()` assim:

```
my_struct *state = (my_struct*)PyType_GetModuleState(type);
if (state == NULL) {
    return NULL;
}
```

4.5 Acesso ao estado do módulo a partir de métodos regulares

Acessar o estado do módulo a partir de métodos de uma classe já é um pouco mais complicado, mas passou a ser possível graças à API introduzida no Python 3.9. Para conseguir o estado, é necessário primeiro acessar a *classe definidora*, e então obter o estado do módulo a partir dela.

O maior obstáculo é encontrar *a classe na qual um método foi definido*, ou, abreviando, a *classe definidora* desse método. A classe definidora pode guardar uma referência para o módulo do qual ela é parte.

Não confunda a classe definidora com `Py_TYPE(self)`. Se o método for chamado em uma *subclasse* do seu tipo, `Py_TYPE(self)` será uma referência àquela subclasse, a qual pode ter sido definida em um módulo diferente do seu.

Nota

O código Python a seguir ilustra esse conceito. `Base.get_defining_class` retorna `Base` mesmo quando `type(self) == Sub`:

```
class Base:
    def get_type_of_self(self):
        return type(self)

    def get_defining_class(self):
        return __class__

class Sub(Base):
    pass
```

Para um método acessar a sua "classe definidora", ele precisa usar a `convenção de chamada` METH_METHOD | METH_FASTCALL | METH_KEYWORDS e a assinatura `PyCMethod` correspondente:

```
PyObject *PyCMethod(
    PyObject *self,               // objeto onde o módulo foi chamado
    PyTypeObject *defining_class, // classe definidora
    PyObject *const *args,        // vetor C de argumentos
    Py_ssize_t nargs,             // comprimento de "args"
    PyObject *kwnames)            // NULL, ou dicionário de argumentos nomeados
```

Uma vez que vc tem a classe definidora, chame `PyType_GetModuleState()` para obter o estado do módulo associado a ela.

Por exemplo:

```
static PyObject *
example_method(PyObject *self,
        PyTypeObject *defining_class,
        PyObject *const *args,
        Py_ssize_t nargs,
        PyObject *kwnames)
{
    my_struct *state = (my_struct*)PyType_GetModuleState(defining_class);
    if (state == NULL) {
        return NULL;
    }
    ... // rest of logic
}

PyDoc_STRVAR(example_method_doc, "...");
```

(continua na próxima página)

(continuação da página anterior)

```
static PyMethodDef my_methods[] = {
    {"example_method",
      (PyCFunction)(void(*)(void))example_method,
      METH_METHOD|METH_FASTCALL|METH_KEYWORDS,
      example_method_doc}
    {NULL},
}
```

4.6 Acesso ao estado do módulo a partir de métodos slot, getters e setters

Nota

Adicionado na versão 3.11

Métodos slot—os métodos rápidos em C equivalentes aos métodos especiais, como `nb_add` para `__add__` ou `tp_new` para inicialização—têm uma API muito simples que não permite passar a classe definidora, ao contrário do `PyCMethod`. O mesmo vale para getters e setters definidos com `PyGetSetDef`.

Para acessar o estado do módulo nesses casos, use a função `PyType_GetModuleByDef()`, e passe a definição do módulo. Uma vez encontrado o módulo, chame `PyModule_GetState()` para obter o estado:

```
PyObject *module = PyType_GetModuleByDef(Py_TYPE(self), &module_def);
my_struct *state = (my_struct*)PyModule_GetState(module);
if (state == NULL) {
    return NULL;
}
```

Essa função `PyType_GetModuleByDef()` funciona procurando na ordem de resolução de métodos (isto é, todas as superclasses) a primeira superclasse que tem um módulo correspondente.

Nota

Em casos muito exóticos (cadeias hereditárias espalhadas através de múltiplos módulos criados a partir da mesma definição), a `PyType_GetModuleByDef()` pode não retornar o módulo da classe definidora correta. De todo modo, essa função sempre vai retornar um módulo com a mesma definição, garantindo um layout de memória C compatível.

4.7 Tempo de vida do estado do módulo

Quando um objeto de módulo é coletado como lixo, o seu estado de módulo é liberado. Para cada ponteiro para o estado do módulo (ou uma parte dele), é necessário possuir uma referência ao objeto de módulo.

Isso não costuma ser um problema, dado que tipos criados com `PyType_FromModuleAndSpec()`, bem como suas instâncias, guardam referências ao módulo. Mesmo assim, é necessário tomar cuidado com a contagem de referências ao referenciar o estado do módulo a partir de outros lugares, como funções de retorno para bibliotecas externas.

5 Problemas em aberto

Vários problemas relacionados aos estados por módulo e aos tipos no heap ainda estão em aberto.

O melhor lugar para discussões sobre como melhorar a situação é a lista de discussão do capi-sig.

5.1 Escopo por classe

Atualmente (desde o Python 3.11) não é possível anexar estado a *tipos* individuais sem depender de detalhes de implementação do CPython (os quais podem mudar no futuro—talvez, ironicamente, para possibilitar uma solução adequada para o escopo por classe).

5.2 Conversão sem perdas para tipos no heap

A API de tipos no heap não foi projetada para conversão "sem perdas" de tipos estáticos. isto é, para criar um tipo que funciona exatamente como um dado tipo estático.