Aislamiento de módulos de extensión

Versión 3.11.8

Guido van Rossum and the Python development team

abril 02, 2024

Python Software Foundation
Email: docs@python.org

Índice general

Resumen

Tradicionalmente, el estado perteneciente a los módulos de extensión de Python se mantuvo en las variables C `static`, que tienen un alcance de todo el proceso. Este documento describe los problemas de dicho estado por proceso y muestra una forma más segura: el estado por módulo.

El documento también describe cómo cambiar al estado por módulo cuando sea posible. Esta transición implica asignar espacio para ese estado, cambiar potencialmente de tipos estáticos a tipos de montón y, quizás lo más importante, acceder al estado por módulo desde el código.

1 ¿Quién debería leer esto?

Esta guía está escrita para los mantenedores de extensiones C-API que deseen hacer que esa extensión sea más segura para usar en aplicaciones donde Python se usa como biblioteca.

2 Trasfondo

Un intérprete (*interpreter*) es el contexto en el que se ejecuta el código de Python. Contiene la configuración (p. ej., la ruta de importación) y el estado de tiempo de ejecución (p. ej., el conjunto de módulos importados).

Python admite la ejecución de varios intérpretes en un solo proceso. Hay dos casos en los que pensar, los usuarios pueden ejecutar intérpretes:

- en secuencia, con varios ciclos `Py_InitializeEx()`/`Py_FinalizeEx()`, y
- en paralelo, gestionando «subintérpretes» mediante `Py_NewInterpreter()`/`Py_EndInterpreter()`.

Ambos casos (y combinaciones de ellos) serían más útiles al incorporar Python dentro de una biblioteca. Las bibliotecas generalmente no deben hacer suposiciones sobre la aplicación que las usa, lo que incluye asumir un «intérprete principal de Python» en todo el proceso.

Históricamente, los módulos de extensión de Python no manejan bien este caso de uso. Muchos módulos de extensión (e incluso algunos módulos stdlib) usan el estado global *por-proceso*, porque las variables C `static` son extremadamente fáciles de usar. Así, los datos que deberían ser específicos de un intérprete acaban siendo compartidos entre intérpretes. A menos que el desarrollador de la extensión tenga cuidado, es muy fácil introducir casos extremos que provocan bloqueos cuando un módulo se carga en más de un intérprete en el mismo proceso.

Desafortunadamente, el estado *por-intérprete* no es fácil de lograr. Los autores de extensiones tienden a no tener en cuenta múltiples intérpretes cuando desarrollan, y actualmente es engorroso probar el comportamiento.

2.1 Ingrese al estado por módulo

En lugar de centrarse en el estado por intérprete, la API C de Python está evolucionando para admitir mejor el estado *por-módulo* más granular. Esto significa que los datos de nivel C se adjuntan a un *module object*. Cada intérprete crea su propio objeto de módulo, manteniendo los datos separados. Para probar el aislamiento, se pueden cargar varios objetos de módulo correspondientes a una sola extensión en un solo intérprete.

El estado por módulo proporciona una manera fácil de pensar en la vida útil y la propiedad de los recursos: el módulo de extensión se inicializará cuando se cree un objeto de módulo y se limpiará cuando se libere. En este sentido, un módulo es como cualquier otro `PyObject*`; no hay ganchos de «apagado del intérprete» para pensar u olvidar.

Tenga en cuenta que hay casos de uso para diferentes tipos de «globales»: por proceso, por intérprete, por subproceso o por estado de tarea. Con el estado por módulo como predeterminado, estos aún son posibles, pero debe tratarlos como casos excepcionales: si los necesita, debe brindarles atención y pruebas adicionales. (Tenga en cuenta que esta guía no los cubre).

2.2 Objetos módulos aislados

El punto clave a tener en cuenta al desarrollar un módulo de extensión es que se pueden crear varios objetos de módulo a partir de una única biblioteca compartida. Por ejemplo:

```
>>> import sys
>>> import binascii
>>> old_binascii = binascii
>>> del sys.modules['binascii']
>>> import binascii  # create a new module object
>>> old_binascii == binascii
False
```

Como regla general, los dos módulos deben ser completamente independientes. Todos los objetos y el estado específico del módulo deben encapsularse dentro del objeto del módulo, no compartirse con otros objetos del módulo y limpiarse cuando se desasigna el objeto del módulo. Dado que esto es solo una regla general, las excepciones son posibles (consulte *Managing Global State*), pero necesitarán más reflexión y atención en los casos extremos.

Si bien algunos módulos podrían funcionar con restricciones menos estrictas, los módulos aislados facilitan el establecimiento de expectativas y pautas claras que funcionan en una variedad de casos de uso.

2.3 Casos extremos sorprendentes

Tenga en cuenta que los módulos aislados crean algunos casos extremos sorprendentes. En particular, cada objeto de módulo normalmente no compartirá sus clases y excepciones con otros módulos similares. Continuando con *example above*, tenga en cuenta que `old_binascii.Error` y `binascii.Error` son objetos separados. En el código siguiente, se detecta la excepción *not*:

```
>>> 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
```

Esto se espera. Tenga en cuenta que los módulos de Python puro se comportan de la misma manera: es una parte de cómo funciona Python.

El objetivo es hacer que los módulos de extensión sean seguros en el nivel C, no hacer que los piratas informáticos se comporten de manera intuitiva. Mutar `sys.modules` «manualmente» cuenta como un hack.

3 Cómo hacer que los módulos sean seguros con varios intérpretes

3.1 Administrar el estado global

A veces, el estado asociado con un módulo de Python no es específico de ese módulo, sino de todo el proceso (o algo más «más global» que un módulo). Por ejemplo:

- El módulo `readline` gestiona *el* terminal.
- Un módulo que se ejecuta en una placa de circuito quiere controlar *el* LED integrado.

En estos casos, el módulo Python debería proporcionar *acceso* al estado global, en lugar de *poseerlo*. Si es posible, escriba el módulo para que varias copias del mismo puedan acceder al estado de forma independiente (junto con otras bibliotecas, ya sea para Python u otros lenguajes). Si eso no es posible, considere el bloqueo explícito.

Si es necesario usar el estado global del proceso, la forma más sencilla de evitar problemas con varios intérpretes es evitar explícitamente que un módulo se cargue más de una vez por proceso; consulte *Opt-Out: Limiting to One Module Object per Process*.

3.2 Administración del estado por módulo

Para usar el estado por módulo, use multi-phase extension module initialization. Esto indica que su módulo admite múltiples intérpretes correctamente.

Establezca `PyModuleDef.m_size` en un número positivo para solicitar tantos bytes de almacenamiento local para el módulo. Por lo general, esto se establecerá en el tamaño de algún `struct` específico del módulo, que puede almacenar todo el estado de nivel C del módulo. En particular, es donde debe colocar los punteros a las clases (incluidas las excepciones, pero excluyendo los tipos estáticos) y configuraciones (por ejemplo, `field_size_limit` de `csv`) que el código C necesita para funcionar.

Nota: Otra opción es almacenar el estado en el `__dict__` del módulo, pero debe evitar fallas cuando los usuarios modifican `__dict__` desde el código de Python. Esto generalmente significa verificación de errores y tipos en el nivel C, que es fácil equivocarse y difícil de probar lo suficiente.

Sin embargo, si el estado del módulo no es necesario en el código C, almacenarlo solo en `__dict__` es una buena idea.

Si el estado del módulo incluye punteros `PyObject`, el objeto del módulo debe contener referencias a esos objetos e implementar los enlaces de nivel de módulo `m_traverse`, `m_clear` y `m_free`. Estos funcionan como `tp_traverse`, `tp_clear` y `tp_free` de una clase. Agregarlos requerirá algo de trabajo y hará que el código sea más largo; este es el precio de los módulos que se pueden descargar limpiamente.

Un ejemplo de un módulo con estado por módulo está actualmente disponible como xxlimited; ejemplo de inicialización del módulo que se muestra en la parte inferior del archivo.

3.3 Exclusión voluntaria: limitación a un objeto de módulo por proceso

Un `PyModuleDef.m_size` no negativo indica que un módulo admite varios intérpretes correctamente. Si este aún no es el caso de su módulo, puede hacer que su módulo se pueda cargar explícitamente solo una vez por proceso. Por ejemplo:

```
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;
    // ... rest of initialization
}
```

3.4 Acceso al estado del módulo desde las funciones

Acceder al estado desde funciones a nivel de módulo es sencillo. Las funciones obtienen el objeto del módulo como su primer argumento; para extraer el estado, puede usar `PyModule_GetState`:

```
static PyObject *
func(PyObject *module, PyObject *args)
{
    my_struct *state = (my_struct*)PyModule_GetState(module);
    if (state == NULL) {
        return NULL;
    }
    // ... rest of logic
}
```

Nota: `PyModule_GetState` puede retornar `NULL` sin establecer una excepción si no hay un estado de módulo, es decir, `PyModuleDef.m_size` era cero. En su propio módulo, tiene el control de `m_size`, por lo que es fácil de evitar.

---

4 Tipos Heap

Tradicionalmente, los tipos definidos en código C son *estáticos*; es decir, estructuras `static PyTypeObject` definidas directamente en el código e inicializadas mediante `PyType_Ready()`.

Tales tipos son necesariamente compartidos a lo largo del proceso. Compartirlos entre objetos de módulo requiere prestar atención a cualquier estado que posean o al que accedan. Para limitar los posibles problemas, los tipos estáticos son inmutables en el nivel de Python: por ejemplo, no puede configurar `str.myattribute = 123`.

Detalles de implementación de CPython: Compartir objetos verdaderamente inmutables entre intérpretes está bien, siempre que no proporcionen acceso a objetos mutables. Sin embargo, en CPython, cada objeto de Python tiene un detalle de implementación mutable: el recuento de referencias. Los cambios en el refcount están protegidos por el GIL. Por lo tanto, el código que comparte cualquier objeto de Python entre intérpretes depende implícitamente del GIL actual de todo el proceso de CPython.

Debido a que son inmutables y globales de proceso, los tipos estáticos no pueden acceder a «su» estado de módulo. Si algún método de este tipo requiere acceso al estado del módulo, el tipo debe convertirse a *tipo almacenado en memoria dinámica (heap)* o *tipo heap* para abreviar. Estos se corresponden más estrechamente con las clases creadas por la instrucción `class` de Python.

Para los módulos nuevos, usar tipos heap de forma predeterminada es una buena regla general.

4.1 Cambio de tipos estáticos a tipos heap

Los tipos estáticos se pueden convertir en tipos heap, pero tenga en cuenta que la API de tipo heap no se diseñó para la conversión «sin pérdidas» de tipos estáticos, es decir, para crear un tipo que funcione exactamente como un tipo estático determinado. Por lo tanto, al reescribir la definición de clase en una nueva API, es probable que cambie sin querer algunos detalles (por ejemplo, capacidad de selección o espacios heredados). Siempre pruebe los detalles que son importantes para usted.

Tenga cuidado con los siguientes dos puntos en particular (pero tenga en cuenta que esta no es una lista completa):

- Unlike static types, heap type objects are mutable by default. Use the `Py_TPFLAGS_IMMUTABLETYPE` flag to prevent mutability.

- Heap types inherit `tp_new` by default, so it may become possible to instantiate them from Python code. You can prevent this with the `Py_TPFLAGS_DISALLOW_INSTANTIATION` flag.

4.2 Definición de tipos heap

Los tipos heap se pueden crear completando una estructura `PyType_Spec`, una descripción o «modelo» de una clase y llamando a `PyType_FromModuleAndSpec()` para construir un nuevo objeto de clase.

Nota: Otras funciones, como `PyType_FromSpec()`, también pueden crear tipos heap, pero `PyType_FromModuleAndSpec()` asocia el módulo con la clase, lo que permite el acceso al estado del módulo desde los métodos.

La clase generalmente debe almacenarse en *ambos*, el estado del módulo (para acceso seguro desde C) y el `__dict__` del módulo (para acceso desde código Python).

4.3 Protocolo de recolección de basura

Las instancias de tipos heap contienen una referencia a su tipo. Esto garantiza que el tipo no se destruya antes de que se destruyan todas sus instancias, pero puede generar ciclos de referencia que el recolector de elementos no utilizados debe interrumpir.

Para evitar pérdidas de memoria, las instancias de los tipos heap deben implementar el protocolo de recolección de elementos no utilizados. Es decir, los tipos heap deben:

- Have the `Py_TPFLAGS_HAVE_GC` flag.
- Define a traverse function using `Py_tp_traverse`, which visits the type (e.g. using `Py_VISIT(Py_TYPE(self))`).

Please refer to the the documentation of `Py_TPFLAGS_HAVE_GC` and `tp_traverse` for additional considerations.

The API for defining heap types grew organically, leaving it somewhat awkward to use in its current state. The following sections will guide you through common issues.

`tp_traverse` in Python 3.8 and lower

The requirement to visit the type from `tp_traverse` was added in Python 3.9. If you support Python 3.8 and lower, the traverse function must *not* visit the type, so it must be more complicated:

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

Unfortunately, `Py_Version` was only added in Python 3.11. As a replacement, use:

- `PY_VERSION_HEX`, if not using the stable ABI, or
- `sys.version_info` (via `PySys_GetObject()` and `PyArg_ParseTuple()`).

Delegating `tp_traverse`

If your traverse function delegates to the `tp_traverse` of its base class (or another type), ensure that `Py_TYPE(self)` is visited only once. Note that only heap type are expected to visit the type in `tp_traverse`.

Por ejemplo, si su función poligonal incluye:

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

…y `base` puede ser un tipo estático, entonces también debe incluir:

```
if (base->tp_flags & Py_TPFLAGS_HEAPTYPE) {
    // a heap type's tp_traverse already visited Py_TYPE(self)
} else {
    if (Py_Version >= 0x03090000) {
        Py_VISIT(Py_TYPE(self));
    }
}
```

It is not necessary to handle the type's reference count in `tp_new` and `tp_clear`.

Defining `tp_dealloc`

If your type has a custom `tp_dealloc` function, it needs to:

- call `PyObject_GC_UnTrack()` before any fields are invalidated, and
- decrement the reference count of the type.

To keep the type valid while `tp_free` is called, the type's refcount needs to be decremented *after* the instance is deallocated. For example:

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

The default `tp_dealloc` function does this, so if your type does *not* override `tp_dealloc` you don't need to add it.

Not overriding `tp_free`

The `tp_free` slot of a heap type must be set to `PyObject_GC_Del()`. This is the default; do not override it.

Avoiding `PyObject_New`

GC-tracked objects need to be allocated using GC-aware functions.

If you use use `PyObject_New()` or `PyObject_NewVar()`:

- Get and call type's `tp_alloc` slot, if possible. That is, replace `TYPE *o = PyObject_New(TYPE, typeobj)` with:

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

  Replace `o = PyObject_NewVar(TYPE, typeobj, size)` with the same, but use size instead of the 0.

- If the above is not possible (e.g. inside a custom `tp_alloc`), call `PyObject_GC_New()` or `PyObject_GC_NewVar()`:

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

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

4.4 Acceso al estado del módulo desde las clases

Si tiene un objeto de tipo definido con `PyType_FromModuleAndSpec()`, puede llamar a `PyType_GetModule()` para obtener el módulo asociado y luego a `PyModule_GetState()` para obtener el estado del módulo.

Para ahorrar un tedioso código repetitivo de manejo de errores, puede combinar estos dos pasos con `PyType_GetModuleState()`, lo que da como resultado:

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

4.5 Acceso al estado del módulo desde métodos regulares

Acceder al estado de nivel de módulo desde los métodos de una clase es algo más complicado, pero es posible gracias a la API introducida en Python 3.9. Para obtener el estado, primero debe obtener la *clase de definición* y luego obtener el estado del módulo.

El obstáculo más grande es obtener *la clase en la que se definió un método*, o la «clase de definición» de ese método para abreviar. La clase de definición puede tener una referencia al módulo del que forma parte.

Do not confuse the defining class with `Py_TYPE(self)`. If the method is called on a *subclass* of your type, `Py_TYPE(self)` will refer to that subclass, which may be defined in different module than yours.

Nota: El siguiente código de Python puede ilustrar el concepto. `Base.get_defining_class` retorna `Base` incluso si `type(self) == Sub`:

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

    def get_defining_class(self):
        return __class__
```

(continúe en la próxima página)

(proviene de la página anterior)

```
class Sub(Base):
    pass
```

For a method to get its «defining class», it must use the METH_METHOD | METH_FASTCALL | METH_KEYWORDS `calling convention` and the corresponding `PyCMethod` signature:

```
PyObject *PyCMethod(
    PyObject *self,               // object the method was called on
    PyTypeObject *defining_class, // defining class
    PyObject *const *args,        // C array of arguments
    Py_ssize_t nargs,             // length of "args"
    PyObject *kwnames)            // NULL, or dict of keyword arguments
```

Una vez que tenga la clase de definición, llame a `PyType_GetModuleState()` para obtener el estado de su módulo asociado.

Por ejemplo:

```
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, "...");

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

4.6 Acceso al estado del módulo desde métodos de Slot, Getters y Setters

Nota: Esto es nuevo en Python 3.11.

Slot methods—the fast C equivalents for special methods, such as `nb_add` for `__add__` or `tp_new` for initialization—have a very simple API that doesn't allow passing in the defining class, unlike with `PyCMethod`. The same goes for getters and setters defined with `PyGetSetDef`.

Para acceder al estado del módulo en estos casos, utilice la función `PyType_GetModuleByDef()` y pase la definición del módulo. Una vez que tenga el módulo, llame a `PyModule_GetState()` para obtener el estado:

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

`PyType_GetModuleByDef()` works by searching the method resolution order (i.e. all superclasses) for the first superclass that has a corresponding module.

Nota: In very exotic cases (inheritance chains spanning multiple modules created from the same definition), `PyType_GetModuleByDef()` might not return the module of the true defining class. However, it will always return a module with the same definition, ensuring a compatible C memory layout.

4.7 Vida útil del estado del módulo

Cuando un objeto de módulo se recolecta como basura, se libera su estado de módulo. Para cada puntero a (una parte de) el estado del módulo, debe tener una referencia al objeto del módulo.

Por lo general, esto no es un problema, porque los tipos creados con `PyType_FromModuleAndSpec()` y sus instancias contienen una referencia al módulo. Sin embargo, debe tener cuidado en el recuento de referencias cuando hace referencia al estado del módulo desde otros lugares, como devoluciones de llamada para bibliotecas externas.

5 Problemas abiertos

Varios problemas relacionados con el estado por módulo y los tipos heap todavía están abiertos.

Las discusiones sobre cómo mejorar la situación se llevan a cabo mejor en el capi-sig mailing list.

5.1 Alcance por clase

Actualmente (a partir de Python 3.11) no es posible adjuntar estado a *tipos* individuales sin depender de los detalles de implementación de CPython (que pueden cambiar en el futuro, tal vez, irónicamente, para permitir una solución adecuada para el alcance por clase).

5.2 Conversión sin pérdidas a tipos heap

La API de tipo heap no se diseñó para la conversión «sin pérdidas» de tipos estáticos; es decir, crear un tipo que funcione exactamente como un tipo estático determinado.