内存管理

概述

在 Python 中，内存管理涉及到一个包含所有 Python 对象和数据结构的私有堆（heap）。这个私有堆的管理由内部的 *Python 内存管理器（Python memory manager）* 保证。Python 内存管理器有不同的组件来处理各种动态存储管理方面的问题，如共享、分割、预分配或缓存。

在最底层，一个原始内存分配器通过与操作系统的内存管理器交互，确保私有堆中有足够的空间来存储所有与 Python 相关的数据。在原始内存分配器的基础上，几个对象特定的分配器在同一堆上运行，并根据每种对象类型的特点实现不同的内存管理策略。例如，整数对象在堆内的管理方式不同于字符串、元组或字典，因为整数需要不同的存储需求和速度与空间的权衡。因此，Python 内存管理器将一些工作分配给对象特定分配器，但确保后者在私有堆的范围内运行。

Python 堆内存的管理是由解释器来执行，用户对它没有控制权，即使他们经常操作指向堆内内存块的对象指针，理解这一点十分重要。Python 对象和其他内部缓冲区的堆空间分配是由 Python 内存管理器按需通过本文档中列出的 Python/C API 函数进行的。

为了避免内存破坏，扩展的作者永远不应该试图用 C 库函数导出的函数来对 Python 对象进行操作，这些函数包括： `malloc()`, `calloc()`, `realloc()` 和 `free()`。这将导致 C 分配器和 Python 内存管理器之间的混用，引发严重后果，这是由于它们实现了不同的算法，并在不同的堆上操作。但是，我们可以安全地使用 C 库分配器为单独的目的分配和释放内存块，如下例所示：

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...执行一些涉及 buf 的 I/O 操作...
res = PyBytes_FromString(buf);
free(buf); /* 已分配的 */
return res;
```

在这个例子中，I/O 缓冲区的内存请求是由 C 库分配器处理的。Python 内存管理器只参与了分配作为结果返回的字节对象。

然而，在大多数情况下，都建议专门基于 Python 堆来分配内存，因为后者是由 Python 内存管理器控制的。 例如，当解释器使用 C 编写的新对象类型进行扩展时就必须这样做。 使用 Python 堆的另一个理由是需要能 *通知* Python 内存管理器有关扩展模块的内存需求。 即使所请求的内存全部只用于内部的、高度特定的目的，将所有的内存请求交给 Python 内存管理器能让解释器对其内存占用的整体情况有更准确的了解。 因此，在特定情况下，Python 内存管理器可能会触发或不触发适当的操作，如垃圾回收、内存压缩或其他的预防性操作。 请注意通过使用前面例子所演示的 C 库分配器，为 I/O 缓冲区分配的内存将完全不受 Python 内存管理器的控制。

参见： 环境变量 `PYTHONMALLOC` 可被用来配置 Python 所使用的内存分配器。

环境变量 PYTHONMALLOCSTATS 可以用来在每次创建和关闭新的 pymalloc 对象区域时打印 pymalloc 内存分配器 的统计数据。

分配器域

所有分配函数都归属于三个不同的"域"之一 (另请参阅 PyMemAllocatorDomain)。 这些域代表不同的分配策略并针对不同的目的进行了优化。 每个域如何分配内存及每个域会调用哪些内部函数的详情被认为是实现细节，但是出于调试目的可以在 这里 找到一张简化的表格。 用于分配和释放内存块的 API 必须来自同一个域。 例如，PyMem_Free() 必须被用来释放使用 PyMem_Malloc() 分配的内存。

三个分配域分别是：

- 原始域：用于为通用内存缓冲区分配内存，其分配 *必须* 转到系统分配器或者可在没有 attached thread state 的情况下使用的分配器。 内存将直接自系统请求。 参见 原始内存接口。
- "内存"域：用于为 Python 缓冲区和通用内存缓冲区分配内存，其分配必须在具有 attached thread state 的情况下进行。 内存将从 Python 私有堆获取。 参见 内存接口。
- 对象域：用于为 Python 对象分配内存。 内存将从 Python 私有堆获取。 参见 对象分配器。

备注： 自由线程 构建版要求仅 Python 对象使用"对象"域来分配并且所有 Python 对象都使用该域来分配。 这不同于之前的 Python 版本，在之前版本中这只是最佳实践而非硬件要求。

例如，缓冲区（非 Python 对象）的分配应当使用 PyMem_Malloc(), PyMem_RawMalloc() 或 malloc()，而不能用 PyObject_Malloc()。

参见 内存分配 API。

原始内存接口

以下函数集是系统分配器的包装器。 这些函数是线程安全的，因此不需要 附加 thread state。

默认原始内存分配器 使用以下函数: malloc(), calloc(), realloc() 和 free()；当请求零个字节时则调用 malloc(1) (或 calloc(1, 1))。

> *Added in version 3.4.*

void *PyMem_RawMalloc(size_t n)

属于 稳定 ABI 自 3.13 版起.

分配 *n* 个字节并返回一个指向所分配内存的 void* 类型指针，如果请求失败则返回 NULL。

请求零字节可能返回一个独特的非 NULL 指针，就像调用了 PyMem_RawMalloc(1) 一样。但是内存不会以任何方式被初始化。

void *PyMem_RawCalloc(size_t nelem, size_t elsize)

属于 稳定 ABI 自 3.13 版起.

分配 *nelem* 个元素，每个元素的大小为 *elsize* 个字节，并返回指向所分配的内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。 内存会被初始化为零。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_RawCalloc(1, 1)` 一样。

> *Added in version 3.5.*

`void *PyMem_RawRealloc(void *p, size_t n)`

属于 稳定 ABI 自 3.13 版起.

将 *p* 指向的内存块大小调整为 *n* 字节。以新旧内存块大小中的最小值为准，其中内容保持不变，

如果 *p* 是 `NULL`，则相当于调用 `PyMem_RawMalloc(n)`；如果 *n* 等于 0，则内存块大小会被调整，但不会被释放，返回非 `NULL` 指针。

除非 *p* 是 `NULL`，否则它必须是之前调用 `PyMem_RawMalloc()`、`PyMem_RawRealloc()` 或 `PyMem_RawCalloc()` 所返回的。

如果请求失败，`PyMem_RawRealloc()` 返回 `NULL`，*p* 仍然是指向先前内存区域的有效指针。

`void PyMem_RawFree(void *p)`

属于 稳定 ABI 自 3.13 版起.

释放 *p* 指向的内存块。 *p* 必须是之前调用 `PyMem_RawMalloc()`、`PyMem_RawRealloc()` 或 `PyMem_RawCalloc()` 所返回的指针。否则，或在 `PyMem_RawFree(p)` 之前已经调用过的情况下，未定义的行为会发生。

如果 *p* 是 `NULL`，那么什么操作也不会进行。

内存接口

以下函数集，仿照 ANSI C 标准，并指定了请求零字节时的行为，可用于从Python堆分配和释放内存。

默认内存分配器 使用了 pymalloc 内存分配器.

警告: 在使用这些函数时必须有一个 attached thread state。

> *在 3.6 版本发生变更:* 现在默认的分配器是 pymalloc 而非系统的 `malloc()`。

`void *PyMem_Malloc(size_t n)`

属于 稳定 ABI.

分配 *n* 个字节并返回一个指向所分配内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_Malloc(1)` 一样。但是内存不会以任何方式被初始化。

`void *PyMem_Calloc(size_t nelem, size_t elsize)`

属于 稳定 ABI 自 3.7 版起.

分配 *nelem* 个元素，每个元素的大小为 *elsize* 个字节，并返回指向所分配的内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。 内存会被初始化为零。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_Calloc(1, 1)` 一样。

> *Added in version 3.5.*

`void *PyMem_Realloc(void *p, size_t n)`

属于 稳定 ABI.

将 *p* 指向的内存块大小调整为 *n* 字节。以新旧内存块大小中的最小值为准，其中内容保持不变，

如果 *p* 是 `NULL`，则相当于调用 `PyMem_Malloc(n)`；如果 *n* 等于 0，则内存块大小会被调整，但不会被释放，返回非 `NULL` 指针。

除非 *p* 是 `NULL`，否则它必须是之前调用 `PyMem_Malloc()`、`PyMem_Realloc()` 或 `PyMem_Calloc()` 所返回的。

如果请求失败，`PyMem_Realloc()` 返回 `NULL`，*p* 仍然是指向先前内存区域的有效指针。

`void PyMem_Free(void *p)`

属于 稳定 ABI.

释放 *p* 指向的内存块。 *p* 必须是之前调用 `PyMem_Malloc()`、`PyMem_Realloc()` 或 `PyMem_Calloc()` 所返回的指针。否则，或在 `PyMem_Free(p)` 之前已经调用过的情况下，未定义的行为会发生。

如果 *p* 是 `NULL`，那么什么操作也不会进行。

以下面向类型的宏为方便而提供。 注意 *TYPE* 可以指任何 C 类型。

`PyMem_New(TYPE, n)`

与 `PyMem_Malloc()` 相同，但会分配 `(n * sizeof(TYPE))` 字节的内存。 返回一个转换为 `TYPE*` 的指针。 内存不会以任何方式被初始化。

`PyMem_Resize(p, TYPE, n)`

与 `PyMem_Realloc()` 类似，但内存块的大小被调整为 `(n * sizeof(TYPE))` 个字节。 返回一个转换为 `TYPE*` 的指针。 在返回时，*p* 将是一个指向新内存区域的指针，或者如果执行失败则为 `NULL`。

这是一个 C 预处理宏， *p* 总是被重新赋值。请保存 *p* 的原始值，以避免在处理错误时丢失内存。

`void PyMem_Del(void *p)`

与 `PyMem_Free()` 相同

此外，我们还提供了以下宏集用于直接调用 Python 内存分配器，而不涉及上面列出的 C API 函数。但是请注意，使用它们并不能保证跨 Python 版本的二进制兼容性，因此在扩展模块被弃用。

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

对象分配器

以下函数集，仿照 ANSI C 标准，并指定了请求零字节时的行为，可用于从Python堆分配和释放内存。

> **备注：** 当通过 自定义内存分配器 部分描述的方法拦截该域中的分配函数时，无法保证这些分配器返回的内存可以被成功地转换成 Python 对象。

默认对象分配器 使用 pymalloc 内存分配器.

> **警告：** 在使用这些函数时必须有一个 attached thread state。

`void *PyObject_Malloc(size_t n)`
属于 稳定 ABI.

分配 *n* 个字节并返回一个指向所分配内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyObject_Malloc(1)` 一样。但是内存不会以任何方式被初始化。

`void *PyObject_Calloc(size_t nelem, size_t elsize)`
属于 稳定 ABI 自 3.7 版起.

分配 *nelem* 个元素，每个元素的大小为 *elsize* 个字节，并返回指向所分配的内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。 内存会被初始化为零。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyObject_Calloc(1, 1)` 一样。

> *Added in version 3.5.*

`void *PyObject_Realloc(void *p, size_t n)`
属于 稳定 ABI.

将 *p* 指向的内存块大小调整为 *n* 字节。以新旧内存块大小中的最小值为准，其中内容保持不变，

如果*p*是 `NULL`，则相当于调用 `PyObject_Malloc(n)`；如果 *n* 等于 0，则内存块大小会被调整，但不会被释放，返回非 `NULL` 指针。

除非 *p* 是 `NULL`，否则它必须是之前调用 `PyObject_Malloc()`、`PyObject_Realloc()` 或 `PyObject_Calloc()` 所返回的。

如果请求失败，`PyObject_Realloc()` 返回 `NULL`，*p* 仍然是指向先前内存区域的有效指针。

`void PyObject_Free(void *p)`

属于 稳定 ABI.

释放 *p* 指向的内存块。*p* 必须是之前调用 `PyObject_Malloc()`、`PyObject_Realloc()` 或 `PyObject_Calloc()` 所返回的指针。否则，或在 `PyObject_Free(p)` 之前已经调用过的情况下，未定义行为会发生。

如果 *p* 是 `NULL`，那么什么操作也不会进行。

请不要直接调用此函数来释放对象的内存；而应调用类型的 `tp_free` 槽位。

请不要为 `PyObject_GC_New` 或 `PyObject_GC_NewVar` 所分配的内存使用此宏；而应改用 `PyObject_GC_Del()`。

参见:

- `PyObject_GC_Del()` 是该函数针对由支持垃圾回收的类型分配的内存的等价物。
- `PyObject_Malloc()`
- `PyObject_Realloc()`
- `PyObject_Calloc()`
- `PyObject_New`
- `PyObject_NewVar`
- `PyType_GenericAlloc()`
- `tp_free`

默认内存分配器

默认内存分配器：

配置	名称	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
发布版本	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
调试构建	`"pymalloc_debug"`	`malloc` + debug	`pymalloc` + debug	`pymalloc` + debug
没有 pymalloc 的	`"malloc"`	`malloc`	`malloc`	`malloc`

配置	名称	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
发布版本				
没有 pymalloc 的调试构建	`"malloc_debug"`	`malloc` + debug	`malloc` + debug	`malloc` + debug

说明：

- 名称: `PYTHONMALLOC` 环境变量的值。
- `malloc`：来自 C 标准库的系统分配器，C 函数：`malloc()`、`calloc()`、`realloc()` 和 `free()`。
- `pymalloc`： pymalloc 内存分配器.
- `mimalloc`: mimalloc 内存分配器。 如果 mimalloc 不受支持则将使用 pymalloc 分配器。
- "+ debug": 附带 Python 内存分配器的调试钩子.
- “调试构建”： 调试模式下的 Python 构建。

自定义内存分配器

> *Added in version 3.4.*

type **PyMemAllocatorEx**

用于描述内存块分配器的结构体。 该结构体下列字段:

域	含意
`void *ctx`	作为第一个参数传入的用户上下文
`void* malloc(void *ctx, size_t size)`	分配一个内存块
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	分配一个初始化为 0 的内存块
`void* realloc(void *ctx, void *ptr, size_t new_size)`	分配一个内存块或调整其大小
`void free(void *ctx, void *ptr)`	释放一个内存块

> *在 3.5 版本发生变更:* `PyMemAllocator` 结构被重命名为 `PyMemAllocatorEx` 并新增了一个 `calloc` 字段。

type **PyMemAllocatorDomain**

用来识别分配器域的枚举类。域有：

PYMEM_DOMAIN_RAW

函数

- PyMem_RawMalloc()
- PyMem_RawRealloc()
- PyMem_RawCalloc()
- PyMem_RawFree()

PYMEM_DOMAIN_MEM

函数

- PyMem_Malloc(),
- PyMem_Realloc()
- PyMem_Calloc()
- PyMem_Free()

PYMEM_DOMAIN_OBJ

函数

- PyObject_Malloc()
- PyObject_Realloc()
- PyObject_Calloc()
- PyObject_Free()

void **PyMem_GetAllocator**(PyMemAllocatorDomain domain, PyMemAllocatorEx *allocator)

获取指定域的内存块分配器。

void **PyMem_SetAllocator**(PyMemAllocatorDomain domain, PyMemAllocatorEx *allocator)

设置指定域的内存块分配器。

当请求零字节时，新的分配器必须返回一个独特的非 `NULL` 指针。

对于 PYMEM_DOMAIN_RAW 域，分配器必须是线程安全的：当分配器被调用时 thread state 是没有 附加的。

对于其余的域，分配器也必须是线程安全的：分配器可以在不共享 GIL 的不同解释器中被调用。

如果新的分配器不是钩子（不调用之前的分配器），必须调用 PyMem_SetupDebugHooks() 函数在新分配器上重新安装调试钩子。

另请参阅 PyPreConfig.allocator 和 Preinitialize Python with PyPreConfig。

> **警告：** PyMem_SetAllocator() 没有以下合约:
>
> - 可以在 Py_PreInitialize() 之后和 Py_InitializeFromConfig() 之前调用它来安装自定义的内存分配器。 对于所安装的分配器除了域的规定以外没有任何其他限制（例如 Raw

Domain 允许分配器在没有 attached thread state 的情况下被调用）。 请参阅 有关分配器域的章节 来了解详情。

- 如果在 Python 已完成初始化之后（即 `Py_InitializeFromConfig()` 被调用之后）被调用则自定义分配器 **must** 必须包装现有的分配器。 将现有分配器替换为任意的其他分配器是 **不受支持的。**

> *在 3.12 版本发生变更:* 所有分配器都必须是线程安全的。

`void` **`PyMem_SetupDebugHooks`**`(void)`

设置 Python 内存分配器的调试钩子 以检测内存错误。

Python 内存分配器的调试钩子

当 Python 在调试模式下构建，`PyMem_SetupDebugHooks()` 函数在 Python 预初始化 时被调用，以在 Python 内存分配器上设置调试钩子以检测内存错误。

`PYTHONMALLOC` 环境变量可被用于在以发行模式下编译的 Python 上安装调试钩子 (例如: `PYTHONMALLOC=debug`)。

`PyMem_SetupDebugHooks()` 函数可被用于在调用了 `PyMem_SetAllocator()` 之后设置调试钩子。

这些调试钩子用特殊的、可辨认的位模式填充动态分配的内存块。新分配的内存用字节 `0xCD` （`PYMEM_CLEANBYTE` ）填充，释放的内存用字节 `0xDD` （`PYMEM_DEADBYTE` ）填充。内存块被填充了字节 `0xFD` （`PYMEM_FORBIDDENBYTE` ）的"禁止字节"包围。这些字节串不太可能是合法的地址、浮点数或ASCII字符串

运行时检查：

- 检测对 API 的违反。例如：检测对 `PyMem_Malloc()` 分配的内存块调用 `PyObject_Free()`。
- 检测缓冲区起始位置前的写入（缓冲区下溢）。
- 检测缓冲区终止位置后的写入（缓冲区溢出）。
- 检测当调用 `PYMEM_DOMAIN_OBJ` (如: `PyObject_Malloc()`) 和 `PYMEM_DOMAIN_MEM` (如: `PyMem_Malloc()`) 域的分配器函数时是否持有 attached thread state。

在出错时，调试钩子使用 `tracemalloc` 模块来回溯内存块被分配的位置。只有当 `tracemalloc` 正在追踪 Python 内存分配，并且内存块被追踪时，才会显示回溯。

让 *S* = `sizeof(size_t)`。 将 `2*S` 个字节添加到每个被请求的 *N* 字节数据块的两端。 内存的布局像是这样，其中 p 代表由类似 malloc 或类似 realloc 的函数所返回的地址 (`p[i:j]` 表示从 `*(p+i)` 左侧开始到 `*(p+j)` 左侧止的字节数据切片；请注意对负索引号的处理与 Python 切片是不同的)：

`p[-2*S:-S]`

最初所要求的字节数。 这是一个 size_t，为大端序（易于在内存转储中读取）。

`p[-S]`

API 标识符（ASCII 字符）：

- `'r'` 表示 `PYMEM_DOMAIN_RAW`。

- `'m'` 表示 `PYMEM_DOMAIN_MEM`。
- `'o'` 表示 `PYMEM_DOMAIN_OBJ`。

`p[-S+1:0]`
: PYMEM_FORBIDDENBYTE 的副本。 用于捕获下层的写入和读取。

`p[0:N]`
: 所请求的内存，用 PYMEM_CLEANBYTE 的副本填充，用于捕获对未初始化内存的引用。 当调用 realloc 之类的函数来请求更大的内存块时，额外新增的字节也会用 PYMEM_CLEANBYTE 来填充。 当调用 free 之类的函数时，这些字节会用 PYMEM_DEADBYTE 来重写，以捕获对已释放内存的引用。 当调用 realloc 之类的函数来请求更小的内存块时，多余的旧字节也会用 PYMEM_DEADBYTE 来填充。

`p[N:N+S]`
: PYMEM_FORBIDDENBYTE 的副本。 用于捕获超限的写入和读取。

`p[N+S:N+2*S]`
: 仅当定义了 `PYMEM_DEBUG_SERIALNO` 宏时会被使用（默认情况下将不定义）。

  一个序列号，每次调用 malloc 或 realloc 之类的函数时都会递增 1。 大端序的 `size_t`。 如果之后检测到了“被破坏的内存”，此序列号提供了一个很好的手段用来在下次运行时设置中断点，以捕获该内存块被破坏的瞬间。 obmalloc.c 中的静态函数 bumpserialno() 是唯一会递增序列号的函数，它的存在让你可以轻松地设置这样的中断点。

一个 realloc 之类或 free 之类的函数会先检查两端的 PYMEM_FORBIDDENBYTE 字节是否完好。 如果它们被改变了，则会将诊断输出写入到 stderr，并且程序将通过 Py_FatalError() 中止。 另一种主要的失败模式是当程序读到某种特殊的比特模式并试图将其用作地址时触发内存错误。 如果你随即进入调试器并查看该对象，你很可能会看到它已完全被填充为 PYMEM_DEADBYTE (意味着已释放的内存被使用) 或 PYMEM_CLEANBYTE (意味着未初始货摊内存被使用)。

> *在 3.6 版本发生变更:* `PyMem_SetupDebugHooks()` 函数现在也能在使用发布模式编译的 Python 上工作。 当发生错误时，调试钩子现在会使用 `tracemalloc` 来获取已分配内存块的回溯信息。 调试钩子现在还会在 `PYMEM_DOMAIN_OBJ` 和 `PYMEM_DOMAIN_MEM` 作用域的函数被调用时检查是否有 attached thread state。

> *在 3.8 版本发生变更:* 字节模式 `0xCB` (`PYMEM_CLEANBYTE`)、 `0xDB` (`PYMEM_DEADBYTE`) 和 `0xFB` (`PYMEM_FORBIDDENBYTE`) 已被 `0xCD` 、 `0xDD` 和 `0xFD` 替代以使用与 Windows CRT 调试 `malloc()` 和 `free()` 相同的值。

pymalloc 分配器

Python 有一个针对短生命周期的小对象（小于或等于 512 字节）进行了优化的 *pymalloc* 分配器。它使用名为“arena”的内存映射，在 32 位平台上的固定大小为 256 KiB，在 64 位平台上的固定大小为 1 MiB。 对于大于 512 字节的分配，它会回退为 `PyMem_RawMalloc()` 和 `PyMem_RawRealloc()`。

pymalloc 是 PYMEM_DOMAIN_MEM (例如: PyMem_Malloc()) 和 PYMEM_DOMAIN_OBJ (例如: PyObject_Malloc()) 域的 默认分配器。

arena 分配器使用以下函数:

- Windows 上的 `VirtualAlloc()` 和 `VirtualFree()`,
- `mmap()` 和 `munmap()`，如果可用的话,
- 否则， `malloc()` 和 `free()` 。

如果 Python 配置了 `--without-pymalloc` 选项，那么此分配器将被禁用。也可以在运行时使用 PYTHONMALLOC (例如: `PYTHONMALLOC=malloc`) 环境变量来禁用它。

通常，在使用 AddressSanitizer（通过 `--with-address-sanitizer` 选项）构建 Python 时，建议禁用 pymalloc 内存分配器，这有助于发现 C 代码中的底层错误。

自定义 pymalloc Arena 分配器

> *Added in version 3.4.*

type **PyObjectArenaAllocator**
用来描述一个 arena 分配器的结构体。这个结构体有三个字段:

域	含意
`void *ctx`	作为第一个参数传入的用户上下文
`void* alloc(void *ctx, size_t size)`	分配一块 size 字节的区域
`void free(void *ctx, void *ptr, size_t size)`	释放一块区域

void **PyObject_GetArenaAllocator**(PyObjectArenaAllocator *allocator)
获取 arena 分配器

void **PyObject_SetArenaAllocator**(PyObjectArenaAllocator *allocator)
设置 arena 分配器

mimalloc 分配器

> *Added in version 3.13.*

Python 会在下层平台提供支持的情况下支持 mimalloc 分配器。 mimalloc "是一个具有优良运行效率特性的通用分配器。 它最初由 Daan Leijen 针对 Koka 和 Lean 语言运行时系统开发。"

tracemalloc C API

> *Added in version 3.7.*

int **PyTraceMalloc_Track**(unsigned int domain, uintptr_t ptr, size_t size)

在 tracemalloc 模块中跟踪一个已分配的内存块。

成功时返回 `0`，出错时返回 `-1` (无法分配内存来保存跟踪信息)。 如果禁用了 tracemalloc 则返回 `-2`。

如果内存块已被跟踪，则更新现有跟踪信息。

`int` **PyTraceMalloc_Untrack**(`unsigned int domain, uintptr_t ptr`)

在 tracemalloc 模块中取消跟踪一个已分配的内存块。 如果内存块未被跟踪则不执行任何操作。

如果 tracemalloc 被禁用则返回 `-2`，否则返回 `0`。

例子

以下是来自 概述 小节的示例，经过重写以使 I/O 缓冲区是通过使用第一个函数集从 Python 堆中分配的:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...执行一些涉及 buf 的 I/O 操作... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* 使用 PyMem_Malloc 分配的 */
return res;
```

使用面向类型函数集的相同代码:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* 用于 I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...执行一些涉及 buf 的 I/O 操作... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* 使用 PyMem_New 分配的 */
return res;
```

请注意在以上两个示例中，缓冲区总是通过归属于相同集的函数来操纵的。 事实上，对于一个给定的内存块必须使用相同的内存 API 族，以便使得混合不同分配器的风险减至最低。 以下代码序列包含两处错误，其中一个被标记为 *fatal* 因为它混合了两种在不同堆上操作的不同分配器。

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* 错误 -- 应为 PyMem_Free() */
free(buf2);       /* 正确 -- 通过 malloc() 分配的 */
free(buf1);       /* 致命错误 -- 应为 PyMem_Free()  */
```

除了用于处理来自 Python 堆的原始内存块的函数，Python 中的对象还通过 `PyObject_New`, `PyObject_NewVar` 和 `PyObject_Free()` 进行分配和释放。

这些将在有关如何在 C 中定义和实现新对象类型的下一章中讲解。