《源码探秘 CPython》4. 对象是怎么被创建的?

原创 古明地觉 古明地觉的编程教室 2022-01-05 09:30

收录于合集
#CPython
97个 >

微信扫一扫
关注该公众号

Python 为什么慢?

前面我们介绍了 Python 对象在底层的数据结构，我们知道 Python 底层通过**PyObject**和**PyTypeObject**完成了 C++ 所提供的对象的多态特性。在 Python 中创建一个对象，会分配内存并进行初始化，然后 Python 会用一个 PyObject * 来保存和维护这个对象，当然所有对象都是如此。因为指针是可以相互转化的，所以变量在保存一个对象的指针时，会将该指针转成 PyObject * 之后再交给变量保存。

因此在 Python 中，变量的传递（包括函数的参数传递）实际上传递的都是一个泛型指针：PyObject *。这个指针具体指向什么类型的对象我们并不知道，只能通过其内部的 ob_type 成员进行动态判断，而正是因为这个 ob_type，Python 实现了多态机制。

比如：a.pop()，我们不知道这个 a 指向的对象到底是什么类型，但只要 a 可以调用 pop 方法即可，因此 a 可以是一个列表、也可以是一个字典、或者是我们实现了 pop 方法的自定义类的实例对象。所以如果 a 的 ob_type 是一个**PyList_Type ***，那么就调用 PyList_Type 中定义的 pop 操作；如果 a 的 ob_type 是一个 **PyDict_Type ***，那么就调用 PyDict_Type 中定义的 pop 操作。

> 所以变量 a 在不同的情况下，会表现出不同的行为，这正是 Python 多态的核心所在。
> 再比如列表，其内部的元素也都是 PyObject *，当我们通过索引获取到该指针进行操作的时候，也会先通过 ob_type 获取其类型指针，判断它的类型。然后再获取该操作对应的 C 一级的函数、进行执行，如果不支持相应的操作便会报错。所以操作容器内的某个元素，和操作一个变量并无本质上的区别。

从这里我们也能看出来 Python 为什么慢了，因为有相当一部分时间浪费在类型和属性的查找上面。

以变量 a + b 为例，这个 a 和 b 指向的对象可以是整数、浮点数、字符串、列表、元组、甚至是我们自己实现了 __add__ 方法的类的实例对象。因为我们说 Python 中的变量都是一个 PyObject *，所以它可以指向任意的对象，因此 Python 就无法做基于**类型**方面的优化。

首先 Python 底层要通过 ob_type 判断变量指向的对象到底是什么类型，这在 C 的层面上至少需要一次属性查找。然后 Python 将每一个操作都抽象成了一个魔法方法，所以实例相加时要在类型对象中找到该方法对应的函数指针，这又是一次属性查找。找到了之后将 a、b 作为参数传递进去，这会发生一次函数调用，会将对象维护的值拿出来进行运算，然后根据相加的结果创建一个新的对象，再返回其对应的 PyObject * 指针。

所以一个简单的加法运算，Python 在底层做了很多的工作，如果是放在一个循环中呢？那么上面的步骤要重复 N 次。

而对于 C 来讲，由于已经规定好了类型，所以 a + b 在编译之后就是一条简单的机器指令，因此两者在效率上差别很大。

当然我们不是来吐槽 Python 效率的问题，因为任何语言都擅长的一面和不擅长的一面，这里只是通过回顾前面的知识来解释为什么 Python 效率低。

因此当别人问你 Python 为什么效率低的时候，希望你能从这个角度来回答它，主要就两点：

- Python 无法基于类型做优化
- Python 所有的对象都存储在堆上

建议不要一上来就谈 GIL，那是在多线程情况下才需要考虑的问题。而且我相信大部分

觉得 Python 慢的人，都不是因为 Python 无法利用多核才觉得 Python 慢的。

简单回顾了前面的内容，下面我们说一说 Python 的对象从创建到销毁的过程，了解一下对象的生命周期。不过由于这部分内容比较多，我们会分开说，先来看看对象是如何创建的。

Python / C API

当我们在控制台敲下这个语句的时候，Python 内部是如何从无到有创建一个浮点数对象的？

```
>>> e = 2.71
```

另外 Python 又是怎么知道该如何将它打印到屏幕上面呢？

```
>>> print(e)
2.71
```

对象使用完毕时，Python 还要将其销毁，那么销毁的时机又该如何确定呢？带着这些问题，我们来探寻一个对象从创建到销毁整个生命周期中的行为表现，然后从中寻找答案。

不过在探寻对象的创建之前，需要先介绍 Python 提供的 **C API**，也叫**Python/C API**。

Python 对外提供了 C API，让用户可以从 C 环境中与其交互。实际上，由于 Python 解释器是用 C 写成的，所以 Python 内部也在大量使用这些 **C API**。为了更好的研读源码，系统地了解这些 API 的组成结构是很有必要的，而 C API 分为两类，分别是**泛型API**和**特型API**。

泛型API

泛型API与类型无关，属于**抽象对象层(Abstract Object Layer，AOL)**，这类 API 的第一个参数是**PyObject ***，可以处理任意类型的对象，API 内部会根据对象的类型进行区别处理。而且泛型 API 名称也是有规律的，具有**PyObject_Xxx**这种形式。

以对象打印函数为例：

```
int PyObject_Print(PyObject *op, FILE *fp, int flags)
```

接口的第一个参数为待打印的对象的指针，可以是任意类型的对象的指针，因此参数类型是**PyObject ***。而我们说**PyObject ***是 Python 底层的一个泛型指针，通过这个泛型指针来实现多态的机制。第二个参数是文件句柄，表示输出的位置，默认是 stdout、即控制台；而 flags 表示是要以 __str__ 打印还是要以 __repr__ 打印。

```
// 假设有两个PyObject *, fo和lo
// fo指向 PyFloatObject, lo指向 PyLongObject
// 但是它们在打印的时候都可以调用这个相同的打印方法
PyObject_Print(fo, stdout, 0);
PyObject_Print(lo, stdout, 0);
```

PyObject_Print接口内部会根据对象类型，决定如何输出对象。

特型API

特型API与类型相关，属于**具体对象层(Concrete Object Layer，COL)**。这类 API 只能作用于某种具体类型的对象，比如：浮点数 PyFloatObject，而 Python 内部为每一种内置类型的实例对象都提供了很多的特型 API。比如：

```
// 通过C的double创建PyFloatObject
PyObject* PyFloat_FromDouble(double v);

// 通过C的Long创建PyLongObject
PyObject* PyLong_FromLong(long v);
// 通过C的char *来创建PyLongObject
PyObject* PyLong_FromString(const char *str, char **pend, int base)
```

特型API也是有规律的，尤其是关于C类型和Python类型互转的时候，会用到以下两种特型API：

- Py###_From@@@：根据**C的对象**创建**Python的对象**，**###**表示Python的类型， **@@@**表示C的类型，比如**PyFloat_FromDouble**表示根据C的double创建Python的float
- Py###_As@@@：根据**Python的对象**创建**C的对象**， **###**表示Python的类型，**@@@**表示C的类型，比如**PyFloat_AsDouble**表示根据Python的float创建C的double; PyLong_AsLong表示根据 Python 的 int 创建 C 的 long，注意：int 对象在底层的名字是PyLongObject，而不是PyIntObject

了解了 Python / C API 之后，我们再来看看对象是如何创建的。

对象是如何创建的

经过前面的理论学习，我们知道对象的元数据保存在对应的类型对象中，元数据当然也包括对象要如何创建等信息。

比如执行 pi = 3.14，那么这个过程都发生了什么呢？首先解释器会根据 3.14 推断出要创建的对象是浮点数，所以会创建出维护的值为3.14的PyFloatObject，并将其指针转化成 PyObject * 交给变量 pi。

> 我们说对象的元数据保存在对应的类型对象中，这就意味着对象想要被创建是需要借助其类型对象的，但这是针对于创建自定义类的实例对象而言。创建内置类型的实例对象是直接创建的，至于为什么，我们下面会说。

而创建对象的方式有两种，一种是通过**泛型API**创建，另一种是通过**特型API**创建。比如创建一个浮点数：

使用泛型 API 创建

```
PyObject* pi = PyObject_New(PyObject, &PyFloat_Type);
```

使用特型 API 创建

```
// 创建浮点数
PyObject* pi = PyFloat_FromDouble(3.14);

//创建一个内部可以容纳5个元素的元组
PyObject* tpl = PyTuple_New(5);
//创建一个内部可以容纳5个元素的列表
//当然这是初始容量，列表可以扩容的
PyObject* lst = PyList_New(5);
```

通过泛型 API 可以创建任意的对象，因为该类 API 和类型无关。而使用特性 API 只能创建指定的对象，因为该类 API 是和类型绑定的。比如我们可以用 PyDict_New 创建一个字典，但不可能创建一个集合出来。

而不管采用哪种方式创建，最终的关键步骤都是**分配内存**，创建内置类型的实例对象，Python是可以直接分配内存的。因为它们有哪些成员在底层都是写死的，Python对它们了如指掌，因此可以通过 **Python/C API** 直接分配内存并初始化。以**PyFloat_FromDouble**为例，直接在接口内部为**PyFloatObject**结构体实例分配内存，并初始化相关字段即可。

比如：pi = 3.14，解释器通过 3.14 知道要创建的对象是**PyFloatObject**，那么直接根据**PyFloatObject**里面的成员算一下就可以了，一个**引用计数(ob_refcnt)** + 一个**指针(ob_type)** + 一个**double(ob_fval)** 显然是24个字节，所以直接就分配了。然后将ob_refcnt初始化为1，ob_type设置为**&PyFloat_Type**，ob_fval 设置为 3.14 即可。

同理可变对象也是一样，因为成员都是固定的，类型、以及内部容纳的元素有多少个也可以根据赋的值得到，所以内部的**所有成员**占用了多少内存也是可以算出来的，因此也是可以直接分配内存的。

但对于我们自定义的类型就不行了，假设我们通过 **class Girl:** 定义了一个类，显然实例化的时候不可能通过**PyGirl_New**、或者**PyObject_New(PyObject,&PyGirl_Type)** 这样的 API 去创建，因为底层根本就没有**PyGirl_New**这样的API，也没有**PyGirl_Type**这个类型对象。这种情况下，创建 Girl 的实例对象就需要 Girl 这个类型对象来创建了。因此自定义类的实例对象如何分配内存、如何进行初始化，答案是需要在对应的类型对象里面寻找的。

总的来说，Python 内部创建一个对象的方法有两种：

- 通过 Python/C API，可以是泛型API、也可以是特型API，用于内置类型
- 通过对应的类型对象去创建，多用于自定义类型

到这里我们抛出个问题: **e = 2.71** 和 **e = float(2.71)** 得到的结果都是2.71，但它们之间有什么不同呢。或者说列表: **lst = []** 和 **lst = list()**得到的 lst 也都是一个空列表，但这两种方式有什么区别呢?

我们说创建实例对象可以通过 Python/C API，用于内置类型；也可以通过对应的类型对象去创建，多用于自定义类型。但是通过对应类型对象去创建实例对象其实是一个更加通用的流程，因为它除了支持自定义类型之外、还支持内置类型。比如：

```
>>> lst = []  # 通过Python/C API创建
>>> lst
[]
>>> lst = list()  # 通过类型对象创建
>>> lst
[]
>>> e = 2.71  # 通过Python/C API创建
>>> e
2.71
>>> e = float(2.71)  # 通过类型对象创建
>>> e
2.71
>>>
```

所以我们看到了对象的两种创建方式，我们写上 2.71、或者 []，Python 会直接解析成底层对应的数据结构；而 float(2.71)、或者 list()，虽然结果是一样的，但我们看到这是一个调用，因此要进行参数解析、类型检测、创建栈帧、销毁栈帧等等，所以开销会大一些。

```
import time

t1 = time.perf_counter()
for _ in range(10000000):
    lst = []
t2 = time.perf_counter()
print(t2 - t1)  # 0.5595989


t3 = time.perf_counter()
for _ in range(10000000):
    lst = list()
t4 = time.perf_counter()
print(t4 - t3)  # 1.1722419999999998
```

通过 [] 的方式创建一千万次空列表需要 0.56 秒，但是通过 list() 的方式创建一千万次空列表需要 1.17 秒，主要就在于 list() 是一个调用，而 [] 直接会被解析成底层对应的 PyListObject，因此 [] 的速度会更快一些。同理 3.14 和 float(3.14) 也是如此。

所以对于内置类型的实例对象而言，使用 Python / C API 创建要更快一些。而且事实上使用类型对象去创建的话，会先调用 **tp_new(也就是__new__ 方法)**，然后在 tp_new 内部还是调用了 Python / C API。

比如创建列表：可以**list()**、也可以**[ ]**；创建元组：可以**tuple()**、也可以**()**；创建字典：可以是**dict()**、也可以**{}**。前者是通过类型对象去创建的，后者是通过 Python/C API 创建。但对于内置类型而言，我们推荐使用 Python/C API 创建，会直接解析为对应的 C 一级数据结构，因为这些结构在底层都是已经实现好了的，是可以直接用的，无需通过诸如**list()**这种调用类型对象的方式来创建，因为它们内部还是使用了 Python/C API。

不过以上都是内置类型，而我们自定义的类型就没有这个待遇了，它的实例对象只能通过它自己创建。比如 Girl 这个类，Python 不可能在底层定义一个 PyGirlObject 然后将 API 提供给我们，所以我们只能通过**Girl()**这种调用类型对象的方式来创建它的实例对象。

另外内置类型被称为**静态类**，它和它的实例对象在底层已经被定义好了，我们无法动态修改它们。我们自定义类型被称为**动态类**，它是在解释器运行的过程中动态翻译成 C 的数据结构的，所以我们可以对其进行动态修改。

这里需要再强调一点，Python 的动态性、GIL 等特性，都是解释器在将字节码翻译成 C 代码时动态赋予的，而内置类型（以及我们用 C、Cython 写的扩展类型）它们事先已经被编译好，已经是指向 C 一级的数据结构，因此也就丧失了相应的动态性，不过与之对应的就是效率上的提升。因为**运行效率**和**动态性**本身就是鱼与熊掌的关系。

小结

以上我们就简单分析了 Python 对象被创建的过程，当然这只是一个开头，其背后还隐藏了大量的细节，我们后续会慢慢说。

那么下一篇文章我们就来聊一聊，对象是如何被调用的。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》5. 对象是如何被调用的

下一篇 ›
《源码探秘 CPython》3. type 和 object 的恩怨纠葛

喜欢此内容的人还喜欢

A tour of gRPC：01 - 基础理论
BUG侦探

PASTIS：从HiC矩阵推断染色体3D结构
生信杂记

软件质效领航者 | 优秀案例•蚂蚁集团单元测试用例自动生成平台 SmartUnit
云上软件工程社区