《源码探秘 CPython》84. 初识GIL、以及多个线程之间的调度机制

原创 古明地觉 古明地觉的编程教室 2022-05-09 08:30 发表于北京

楔子

这次我们来说一下Python的多线程，上篇文章提到了Python的线程是对OS线程进行了一个封装，并提供了一个线程状态对象 PyThreadState，来记录OS线程的一些状态信息。

那什么是多线程呢？首先线程是操作系统调度 cpu 工作的最小单元，同理进程则是操作系统资源分配的最小单元，线程是需要依赖于进程的，并且每一个进程只少有一个线程，这个线程我们称之为主线程。而主线程则可以创建子线程，一个进程中如果有多个线程去工作，我们就称之为多线程。

开发一个多线程应用程序是很常见的事情，很多语言都支持多线程，有的是原生支持，有的是通过库来支持。而 Python 毫无疑问也支持多线程，并且它是通过标准库 threading 实现的。

当然标准库 threading 底层依赖了 _thread，而 _thread 是一个用 C 实现的库，位于 Modules/_threadmodule.c 中。还记得这个 Modules 目录是做什么的吗？它也是 Python 源码的一部分，里面存放的都是一些用 C 实现、并且对性能要求较为苛刻的库，编译之后就内嵌在解释器里面了。

另外提到Python的多线程，总会让人想到 GIL（global interpreter lock）这个万恶之源，我们后面会详细介绍。目前我们知道Python的多线程是不能利用多核的，因为虚拟机使用一个全局解释器锁（GIL）来控制线程对程序的执行，这个结果就使得无论你的CPU有多少核，但是同时被线程调度的 CPU 只有一个。不过底层是怎么做的呢？我们下面就来分析一下。

GIL 与线程调度

首先我们来分析一下为什么会有GIL这个东西存在？举个栗子：

```
import dis

dis.dis("del obj")
"""
 0 DELETE_NAME              0 (obj)
 2 LOAD_CONST               0 (None)
 4 RETURN_VALUE
"""
```

当我们使用 del 删除一个变量的时候，对应的指令是 DELETE_NAME，这个指令对应的源码可以自己去查看。总之这条指令做的事情就是通过宏 Py_DECREF 减少一个对象的引用计数，并且判断减少之后其引用计数是否为 0，如果为 0 就进行回收。伪代码如下：

```
--obj->ob_refcnt
if (obj -> ob_refcnt == 0){
    销毁obj
}
```

所以总共是两步：第一步先将对象的引用计数减 1；第二步判断引用计数是否为 0，为 0则进行销毁。那么问题来了，假设有两个线程 A 和 B，内部都引用了某个变量 obj，此时 obj 指向的对象的

引用计数为 2，然后让两个线程都执行 del obj 这行代码。

其中 A 线程先执行，A 线程在执行完 **--obj->ob_refcnt** 之后，会将对象的引用计数减一，但不幸的是，这个时候调度机制将 A 挂起了，唤醒了 B。而 B 也执行 del obj，但是它比较幸运，将两步都一块执行完了。而由于之前 A 已经将引用计数减 1，所以 B 再减 1 之后会发现对象的引用计数为 0，从而执行了对象的销毁动作（tp_dealloc），内存被释放。

然后 A 又被唤醒了，此时开始执行第二个步骤，但由于 **obj->ob_refcnt** 已经被减少到 0，所以条件满足，那么 A 依旧会对 obj 指向的对象进行释放。但是这个对象所占的内存已经被释放了，所以 obj 此时就成了悬空指针。如果再对 obj 指向的对象进行释放，最终会引发什么结果，只有天知道，这也是臭名昭著的二次释放。

关键来了，所以 CPython 引入了 GIL，GIL 是解释器层面上的一把超级大锁，它是字节码级别的互斥锁。作用就是：在同时一刻，只让一个线程执行字节码，并且保证每一条字节码在执行的时候都不会被打断。

因此由于 GIL 的存在，会使得线程只有把当前的某条字节码指令执行完毕之后才有可能会发生调度。因此无论是 A 还是 B，线程调度时，要么发生在 DELETE_NAME 这条指令执行之前，要么发生在 DELETE_NAME 这条指令执行完毕之后，但是不存在指令（不仅是 DELETE_NAME，而是所有指令）执行到一半的时候发生调度。

因此 GIL 才被称之为是字节码级别的互斥锁，它保护每条字节码指令只有在执行完毕之后才会发生线程调度。

所以回到上面那个 del obj 这个例子中来，由于引入了 GIL，所以就不存在我们之前说的：在A将引用计数减一之后，挂起A、唤醒B这一过程。因为A已经开始了DELETE_NAME这条指令的执行，而在没执行完之前是不会发生线程调度的，后面会通过源码进行分析，总之此时就不会发生悬空指针的问题了。

所以 Python 的一条字节码指令会对应多行 C 代码，这其中可能会涉及很多个 C 函数的调用，我们举个栗子：

```
case TARGET(FOR_ITER): {
    PREDICTED(FOR_ITER);
    /* before: [iter]; after: [iter, iter()] *or* [] */
    PyObject *iter = TOP();
    PyObject *next = (*iter->ob_type->tp_iternext)(iter);
    if (next != NULL) {
        PUSH(next);
        PREDICT(STORE_FAST);
        PREDICT(UNPACK_SEQUENCE);
        DISPATCH();
    }
    if (_PyErr_Occurred(tstate)) {
        if (!_PyErr_ExceptionMatches(tstate, PyExc_StopIte
            goto error;
        }
        else if (tstate->c_tracefunc != NULL) {
            call_exc_trace(tstate->c_tracefunc, tstate->c_
        }
        _PyErr_Clear(tstate);
    }
    /* iterator ended normally */
    STACK_SHRINK(1);
    Py_DECREF(iter);
    JUMPBY(oparg);
    PREDICT(POP_BLOCK);
    DISPATCH();
}
```


这是 FOR_ITER 指令，里面的逻辑非常多，当然也涉及了多个函数调用，而且函数内部又会调用其它的函数。如果没有 GIL，那么这些逻辑在执行的时候，任何一处都可能被打断，发生线程调度。

但是有了 GIL 就不同了，它是施加在字节码层面上的互斥锁，保证每次只有一个线程执行字节码指令。并且不允许指令执行到一半时发生调度，因此 GIL 就保证了每条指令内部的 C 逻辑整体都是原子的。

而如果没有 GIL，那么即使是简单的引用计数，在计算上都有可能出问题。事实上，GIL 最初的目的就是为了解决引用计数的安全性问题。

因此 GIL 对于 Python 对象的内存管理来说是不可或缺的；但是还有一点需要注意，GIL 和 Python 语言本身没有什么关系，它只是官方在实现 CPython 时，为了方便管理内存所引入的一个实现。但是对于其它种类的 Python 解释器则不一定需要 GIL，比如 JPython。

GIL 有没有可能被移除

那么，CPython 解释器中的 GIL 将来是否会被移除呢？因为对于现在的多核 CPU 来说，GIL 无疑是进行了限制。

关于能否移除 GIL，就我本人来看不太可能（针对 CPython），这都几十年了，能移除早就移除了。

而且事实上，在Python诞生没多久，就有人发现了这一诡异之处，因为当时的人发现使用多线程在计算上居然没有任何性能上的提升，反而还比单线程慢了一点。而 Python 的官方人员回复的是：不要使用多线程，去使用多进程。

此时站在上帝视角的我们知道，因为 GIL 的存在使得同一时刻只有一个核被使用，所以对于纯计算的代码来说，理论上多线程和单线程是没有区别的。但是由于多线程涉及上下文的切换，会额外有一些开销，反而还慢一些。

因此在得知 GIL 的存在之后，有两位勇士站了出来表示要移除 GIL，当时 Python 还是 1.5 的版本，非常的古老了。当他们在去掉 GIL 的时候，发现多线程的效率相比之前确实提升了，但是单线程的效率只有原来的一半，这显然是不能接受的。因为把 GIL 去掉了，就意味着需要更细粒度的锁来解决共享数据的安全问题，这就会导致大量的加锁、解锁。而加锁、解锁对于操作系统来说是一个比较重量级的操作，所以 GIL 的移除是极其困难的。

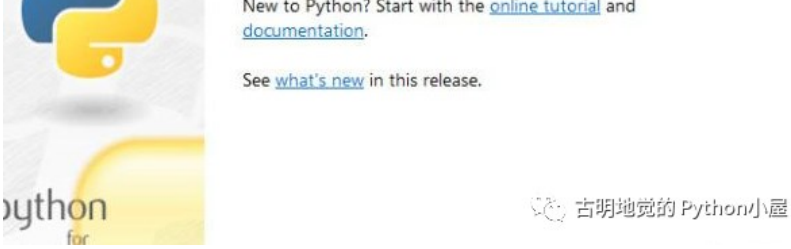
另外还有一个关键，就是当 GIL 被移除之后，会使得扩展模块的编写难度大大增加。因为 GIL 保护的不仅仅是 Python 解释器，还有 Python/C API。像很多现有的 C 扩展，在很大程度上都依赖 GIL 提供的解决方案，如果要移除 GIL，就需要重新解决这些库的线程安全性问题。

比如我们熟知的 numpy，numpy 的速度之所以这么快，就是因为底层是 C 写的，然后封装成 Python 的扩展模块。而其它的库，像 pandas、scipy、sklearn 都是在 numpy 之上开发的，如果把GIL移除了，那么这些库就都不能用了。

还有深度学习，像 tensorflow、pytorch 等框架所使用的底层算法也都不是Python编写的，而是 C 和 C++，Python 只是起到了一个包装器的作用。Python在深度学习领域很火，主要是它可以和 C 无缝结合，如果 GIL 被移除，那么这些框架也没法用了。

因此在 2022 年的今天，生态如此成熟的 Python，几乎是不可能摆脱 GIL 了。否则这些知名的科学计算相关的库就要重新洗牌了，可想而知这是一个什么样的工作量。

小插曲：我们说去掉GIL的老铁有两位，分别是Greg Stein和Mark Hammond，这个Mark Hammond估计很多人都见过。



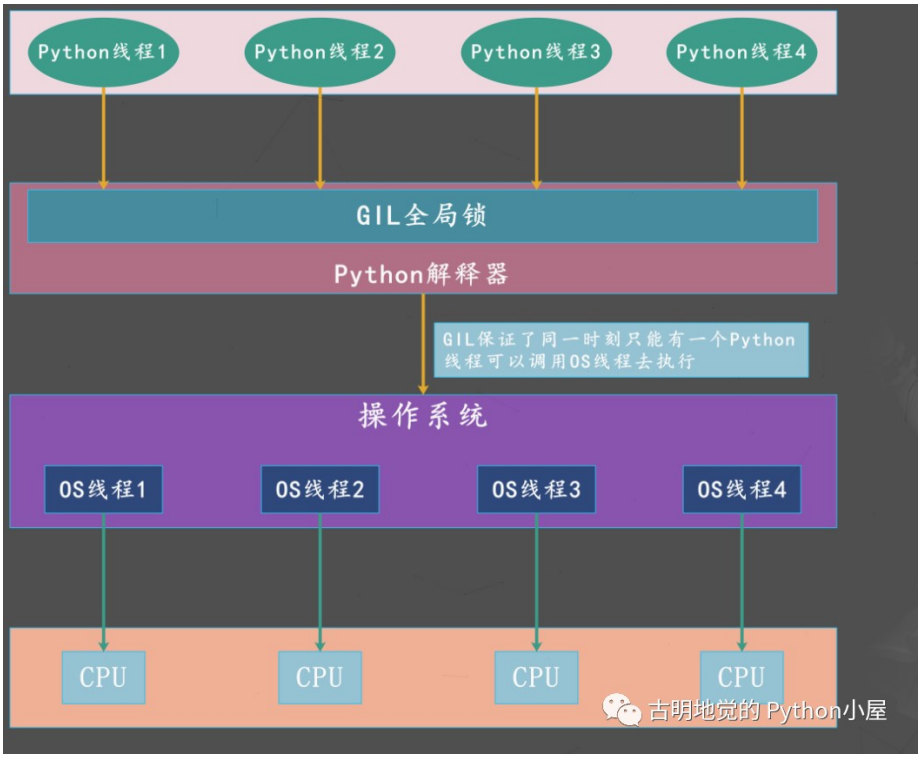
特别感谢 Mark Hammond，没有它这些年无偿分享的Windows专业技术，那么Python如今仍会运行在DOS上。

图解 GIL

Python启动一个线程，底层会启动一个C线程，最终启动一个操作系统的线程。所以还是那句话，Python的线程实际上是封装了C的线程，进而封装了OS线程，一个Python线程对应一个OS线程。

实际执行的肯定是OS线程，而OS线程Python解释器是没有权限控制的，它能控制的只是Python的线程。假设有 4 个Python线程，那么肯定对应 4 个OS线程，但是解释器每次只让一个Python线程调用OS线程去执行，其它的线程只能干等着，只有当前的Python线程将GIL释放了，其它的某个线程在拿到GIL时，才可以调用相应的OS线程去执行。

总结一下就是，没有拿到 GIL 的 Python 线程，对应的 OS 线程会处于休眠状态；拿到 GIL 的 Python 线程，对应的 OS 线程会从休眠状态被唤醒。

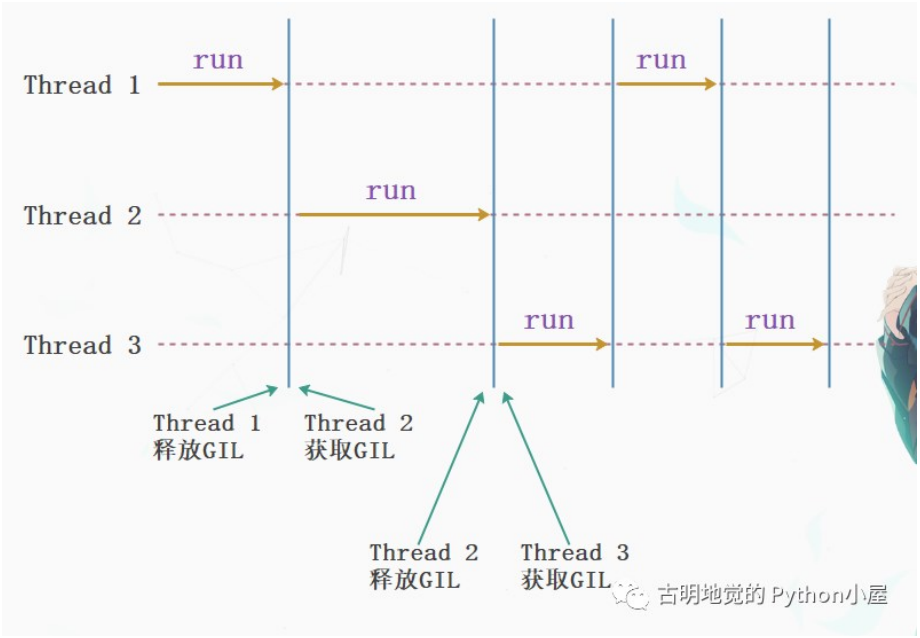


所以Python线程是调用C的线程、进而调用操作系统的OS线程，而OS线程在执行过程中解释器是控制不了的。因为解释器的控制范围只有Python，它无权干预C的线程、更无权干预OS线程。

再次强调：GIL并不是Python语言的特性，它是CPython解释器开发人员为了方便内存管理才加上去的，只不过我们大部分用的都是CPython解释器，所以很多人认为CPython和Python是等价的，但其实不是的。

Python是一门语言，而CPython是对使用Python语言编写的源代码进行解释执行的一个解释器。而解释器不止CPython一种，还有JPython，JPython解释器就没有GIL。因此Python语言本身是和GIL无关的，只不过我们平时在说Python的GIL的时候，指的都是CPython解释器里面的GIL，这

一点要注意。



所以就类似于上图，一个线程执行一会儿，另一个线程执行一会儿，至于线程怎么切换、什么时候切换，我们后面会说。

对于Python而言，解释执行字节码是其核心所在，所以通过GIL来互斥不同线程执行字节码。如果一个线程想要执行，就必须拿到GIL，而一旦拿到GIL，其他线程就无法执行了，如果想执行，那么只能等GIL释放、被自己获取之后才可以执行。并且我们说GIL保护的不仅仅是Python的解释器，还有 Python 的 C API，在 C/C++和 Python混合开发，涉及到原生线程和 Python 线程相互合作时，也需要通过 GIL 进行互斥。

那么问题来了，有了GIL，在编写多线程代码的时候是不是就意味着不需要加锁了呢？

答案显然不是的，因为GIL保护的是每条字节码不会被打断，而很多代码一般都是一行对应多条字节码，所以每行代码是可以被打断的。比如：a = a + 1这样一条语句，它对应4条字节码：LOAD_NAME, LOAD_CONST, INARY_ADD, STORE_NAME。

假设此时 a = 8，两个线程同时执行 a = a + 1，线程 A 执行的时候已经将 a 和 1 压入运行时栈，栈里面的 a 指向的是 8。但是还没有执行BINARY_ADD的时候，发生线程切换，轮到线程 B 执行，此时 B 得到 a 显然还是指向 8，因为线程 A 还没有对变量 a 做加法操作。然后 B 比较幸运，它一次性将这 4 条字节码全部执行完了，所以 a 应该指向 9。

然后线程调度再切换回 A，此时会执行 BINARY_ADD，不过注意：栈里面的 a 目前指向的还是 8，所以加完之后还是 9。

因此本来 a 应该指向10，但是却指向 9，就是因为在执行的时候发生了线程调度。所以我们在编写多线程代码的时候还是需要加锁的，GIL 只是保证每条字节码执行的时候不会被打断，但是一行代码往往对应多条字节码，所以我们会通过 threading.Lock() 再加上一把锁。这样即便发生了线程调度，但由于我们在 Python 的层面上又加了一把锁，别的线程依旧无法执行，这样就保证了数据的安全。

那么问题来了，GIL 啥时候会被释放呢？关于这一点，Python 有一个自己的调度机制：

- 1）当遇见 io 阻塞的时候会把锁释放，因为 io 阻塞是不耗费 CPU 的，所以此时虚拟机会把该线程的锁释放；
- 2）即便是耗费 CPU 的运算，也不会一直执行，会在执行一小段时间之后释放锁，为了保证其他线程都有机会执行，就类似于 CPU 时间片轮转的方式；

调度机制虽然简单，但是这背后还隐藏着两个问题：

- 在何时挂起线程，选择处于等待状态的下一个线程？；
- 在众多处于等待状态的候选线程中，选择激活哪一个线程？

在Python的多线程机制中，这两个问题分别是由不同的层次解决的。对于何时进行线程调度问题，是由 Python 自身决定的。考虑一下操作系统是如何进行进程切换的，当一个进程运行了一段时间之后，发生了时钟中断，操作系统响应时钟，并开始进行进程的调度。

同样，Python也是模拟了这样的时钟中断，来激活线程的调度。我们知道Python解释字节码的原理就是按照指令的顺序一条一条执行，而解释器内部维护着一个数值，这个数值就是Python内部的时钟。在Python2中如果一个线程执行的字节码指令数达到了这个值，那么会进行线程切换，并且这个值在Python3中仍然存在。

```
import sys
# 我们看到默认是执行100条字节码之后启动线程调度机制，进行切换
print(sys.getcheckinterval())  # 100

# 但是在python3中，改成了时间间隔
# 表示一个线程在执行0.005s之后进行切换
print(sys.getswitchinterval())  # 0.005

# 上面的方法我们都可以手动设置
# 通过sys.setcheckinterval(N)和sys.setswitchinterval(N)设置即可
```

在Python3.8的时候，使用 sys.getcheckinterval 和 sys.setcheckinterval会被警告，表示这两个方法已经废弃了。因为线程发生调度不再取决于执行的字节码条数，而是时间间隔。

除了执行时间之外，还有就是我们之前说的遇见 IO 阻塞的时候会进行切换，所以多线程在 IO 密集型还是很有用处的。说实话如果 IO 都不会自动切换的话，那么我觉得Python的多线程才是真的没有用，至于为什么IO会切换后面说，总是现在我们知道Python会在什么时候进行线程切换了。

那么下面的问题就是，Python在切换的时候会从等待的线程中选择哪一个呢？对于这个问题，Python则是借用了底层操作系统所提供的调度机制来决定下一个进入Python解释器的线程究竟是谁。

目前介绍了很多关于 GIL 的内容，主要是为了解释 GIL 到底是个什么东西（底层就是一个结构体实例），以及为什么要有 GIL。那么重点来了，我们能不能手动释放 GIL 呢？

答案是可以的，在用 C 写扩展的时候可以这么做。因为 GIL 是为了解决 Python 的内存管理而引入的，但如果是那些不需要和 Python 代码一起工作的 C 代码，那么是可以在没有 GIL 的情况下运行的。

我们知道 Python 的动态性，是解释器在解释字节码的时候所赐予的。而用 C 写的扩展模块在经过编译之后直接指向了 C 一级的结构，所以它相当于绕过了解释器解释执行这一步，因此也就失去了相应动态特性（换来的是速度的提升）。同理，既然能绕过解释执行这一步，那么就意味着也能绕过 GIL 的限制，因为 GIL 同样是在解释执行字节码的时候施加的。

因此当我们在写扩展时，如果创建了不绑定任何 Python 对象的 C 级结构时，也就是在处理 C-only 部分时，可以将全局解释器锁给释放掉。换句话说，我们可以绕过 GIL，实现基于线程的并行。

> 注意：GIL 是为了保护 Python 对象的内存管理而设置的，如果我们尝试释放 GIL，那么一定一定一定不能和 Python 对象发生任何的交互，必须是纯 C 的数据结构。

《源码探秘 CPython》系列完结之后，会来聊一聊如何使用 C 来给 Python 写扩展。

到目前为止，我们算是以高维的视角理解了什么是 GIL，以及线程切换又是怎么一回事。那么下一

篇文章，我们就从源代码的角度，来分析 GIL 到底是如何实现的。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》85. Python线程的创建、销毁、调度，以及 GIL 的实现原理

下一篇 ›
《源码探秘 CPython》83. 激活 Python 虚拟机