21 GIL 全局锁，束缚Python的紧箍圈-慕课专栏

请问 *GIL* 全局锁的作用是什么？对 *Python* 程序有什么影响？*Python* 程序如何充分利用多核 *CPU* 的计算能力？这几个问题在 *Python* 面试基本都是必问的，然而有不少候选人对此只是一知半解，知其然而不知其所以然。

GIL 作为 *Python* 虚拟机最大的局限性，对 *Python* 程序运行性能影响深刻。因此，*Python* 工程师必须明白虚拟机引入 *GIL* 的前因后果，避免多线程程序被 *Python* 虚拟机戴上金箍圈，进而产生不可预料的性能问题。

由于 *GIL* 的存在不可避免，只有想方设法绕过 *GIL* 的限制才能最大限度地提升 *Python* 程序的多核执行能力。

GIL由来

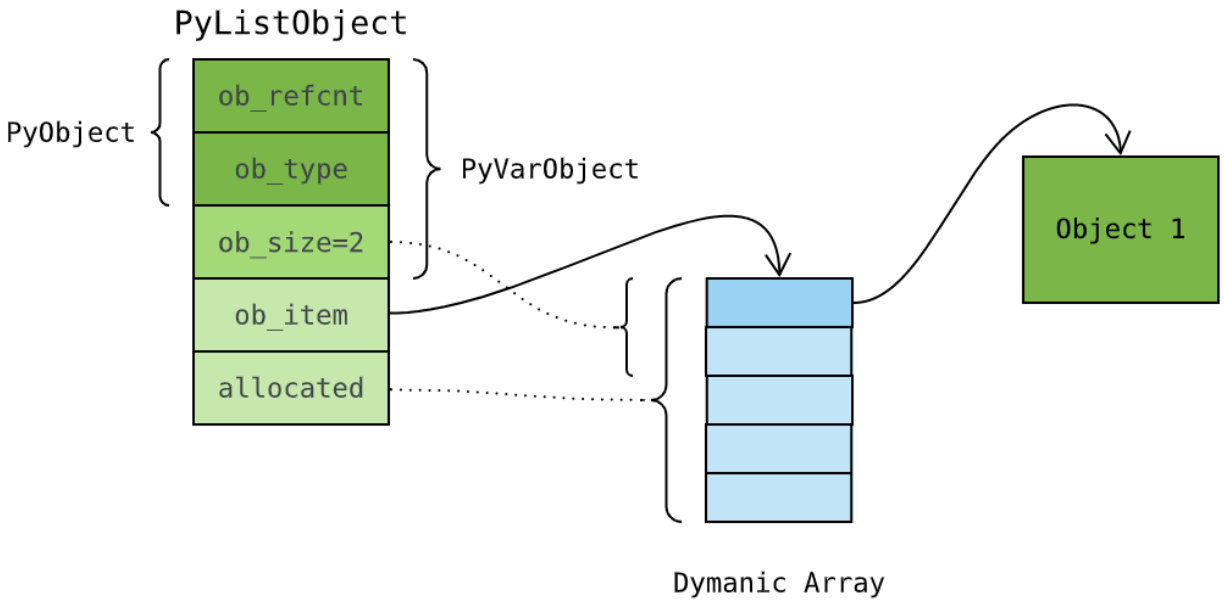
我们先思考一个问题：我们在前面介绍的 *list* 、 *dict* 等内建对象是 **线程安全** 的吗？

在 *Python* 层面，*list* 、*dict* 等内建对象是线程安全的，这是最基本的常识。研究 *list*、*dict* 等内建对象源码时，我们并没有看到任何 **互斥锁** 的痕迹，这多少有点令人意外。以 *list* 对象 *append* 方法为例，主要步骤如下：

```
static int
app1(PyListObject *self, PyObject *v)
{
   Py_ssize_t n = PyList_GET_SIZE(self);

   assert (v != NULL);
   if (n == PY_SSIZE_T_MAX) {
      PyErr_SetString(PyExc_OverflowError,
         "cannot add more objects to list");
      return -1;
   }

   if (list_resize(self, n+1) < 0)
      return -1;

   Py_INCREF(v);
   PyList_SET_ITEM(self, n, v);
   return 0;
}
```

1. 调用 *list_resize* 将列表长度扩大 *1* (第 *13-14* 行)；
2. 将被追加元素设置到末尾位置(第 *16-17* 行)；

由此可见，*append* 方法不是一个 **原子操作** 。假设线程 *A* 调用 *append* 方法，执行长度扩容后便发生 **线程调度** ；系统唤醒线程 *B* 开始执行 *l[-1]* 语句访问 list 末尾元素，会怎样呢？由于 *list* 长度已经扩容但追加元素尚未设置，线程 B 将得到一个非法对象！



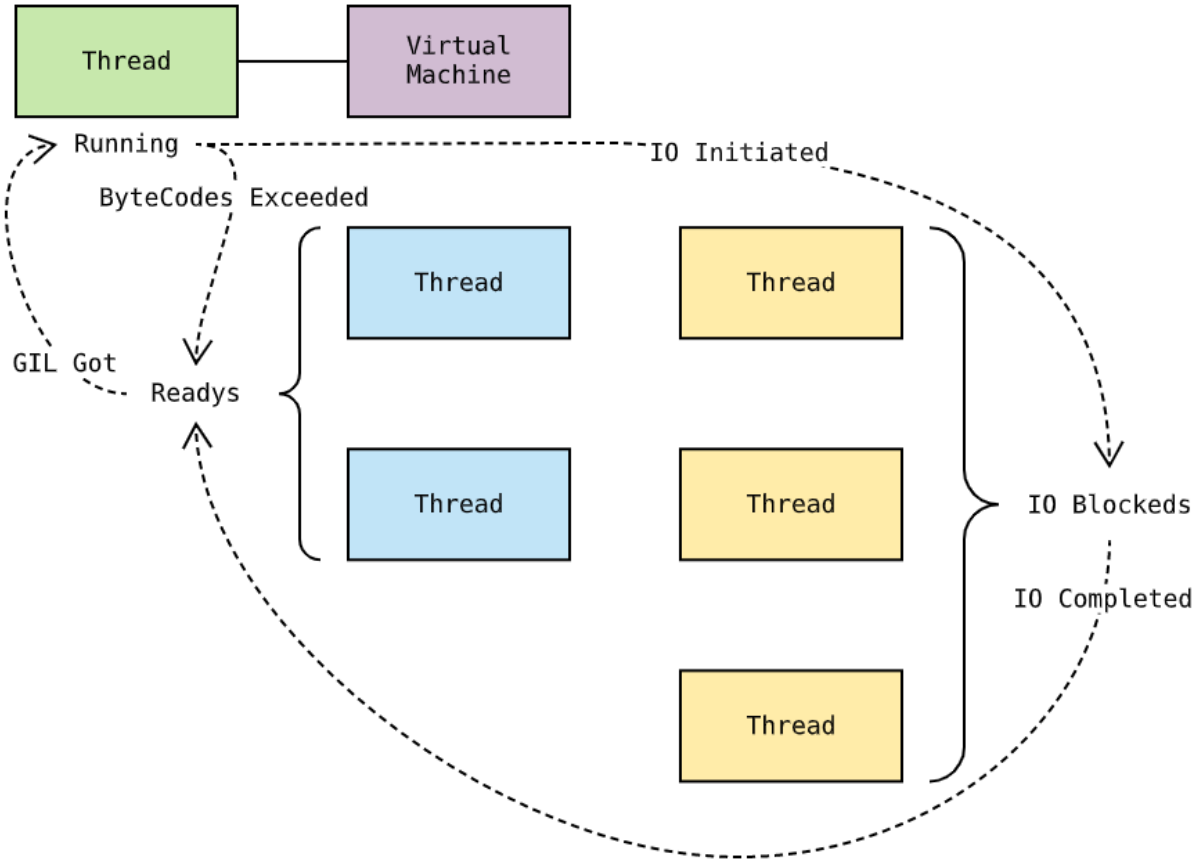
这种有多线程并发操作导致的 **竞争条件** ，一般通过互斥锁加以解决。我们可以为每个 *list* 对象分配一个互斥锁，当一个*list* 操作开始执行前，先获取锁；执行完毕后，再释放锁。进入 **对象锁** 后，竞争条件便消除了。

```
static int
some_op(PyListObject *self, ...)
{

   lock(self);




   unlock(self);
}
```

可我们在源码中并没有发现，这又是为什么呢？莫非 *Python* 采用了其他什么野路子？

的确如此。*Python* 虚拟机维护了一个 **全局锁** ，这就是众所周知的 *GIL*。*Python* 线程想在虚拟机中执行字节码，必须取得全局锁。这样一来，不管任何时刻，只有一个线程在虚拟机中运行。那么，虚拟机如何交替执行不同线程呢？

Python 线程调度实现方式参考了操作系统进程调度中 **时间片** 的思路，只不过将时间片换成 **字节码** 。当一个线程取得 *GIL* 全局锁并开始执行字节码时，对已执行字节码进行计数。当执行字节码达到一定数量(比如 100 条)时，线程主动释放 *GIL* 全局锁并唤醒其他线程。其他等待 *GIL* 全局锁的线程取得锁后，将得到虚拟机控制权并开始执行。因此，虚拟机就像一颗软件 *CPU* ，*Python* 线程交替在虚拟机上执行：



如图，绿色为当前正在虚拟机中执行的线程；蓝色为等待到虚拟机上执行的线程；黄色为阻塞在 *IO* 操作上的线程。对于 *Running* 线程，如果已执行字节码达到一定数量，则自动让出 *GIL* 并唤醒其他线程，状态变更为 *Ready* ；如果执行 *IO* 操作，在执行阻塞型系统调用前先让出 *GIL* ，状态变更为 *IO Blocked* 。当 Ready 线程取得 GIL 后，获得虚拟机控制权并开始执行字节码，状态变更为 *Running* 。 *IO Blocked* 线程一开始阻塞在系统调用上，当系统调用返回后，状态变更为 *Ready* ，再次等待 *GIL* 以便获得虚拟机执行权。

那么，*Python* 为啥采用 *GIL* 这么简单粗暴的解决方案，而不是对象锁呢？首先，对象锁方案为每个需要保护的对象分配一个互斥锁，内存开销巨大。其次，频繁的加解锁操作严重影响执行效率，特别是 *dict* 等对象使用频率很高。

线程安全方案	优势	劣势
对象锁	多线程可并行执行	加解锁开销巨大
GIL	无需频繁加解锁	多进程只能交替执行

有测试结果表明，引入对象锁获得的多线程 **并行执行** 能力，几乎被加解锁开销完全抵消。而在单线程环境下， *GIL* 方案优势明显。这样看来，*Python* 作者们采用 *GIL* 方案也就理所当然了。

GIL影响

在 *GIL* 的束缚下，*Python* 虚拟机同一时刻只能执行一个线程。这是否意味着多线程完全无法优化程序性能呢？由于程序运行特征千差万别，这个问题得分情况讨论。开始之前，我们先来了解两种不同的运行特征：

程序在运行时，一般都处于两种状态：

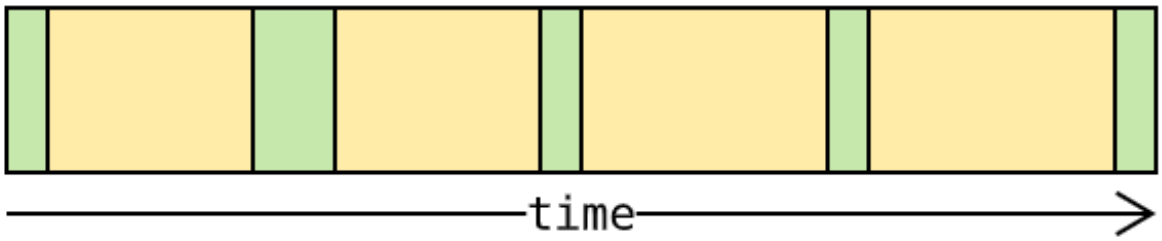
1. **可执行** 状态，包括 *Running* 以及 *Ready* 两种情况，这时竞争处理器资源；
2. **阻塞** 状态，一般为等待 *IO* 处理，这时让出处理器资源；

根据程序分别处于 *Running* 以及 *IO Blocked* 两种状态的时间占比，可分为两种：

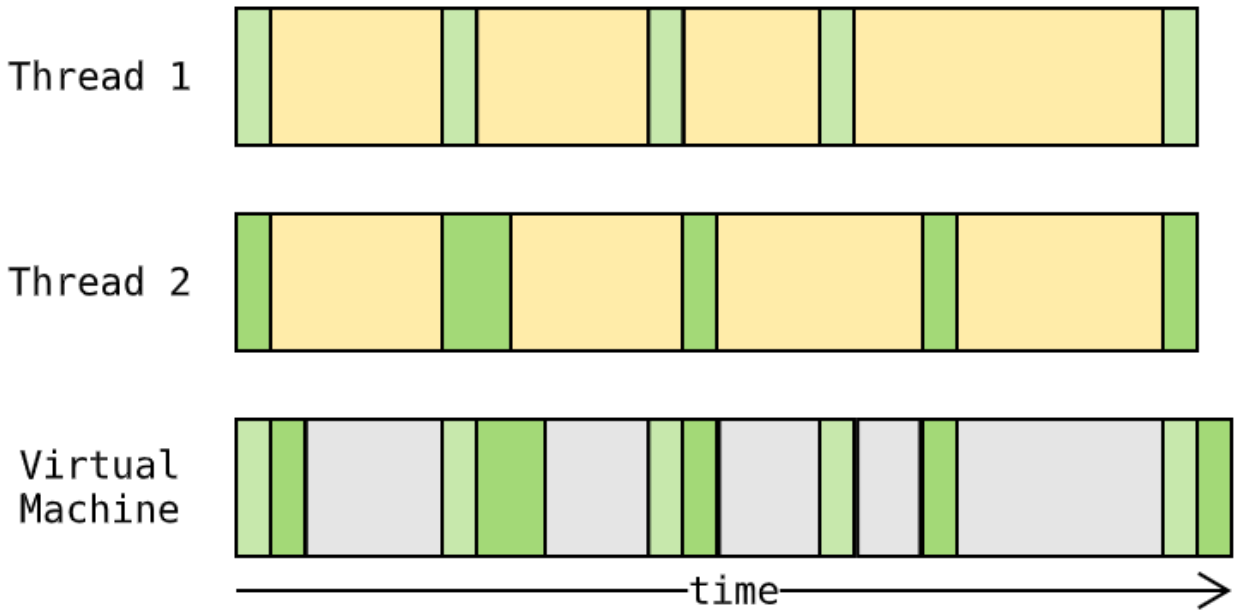
- **计算密集型** ，程序执行时大部分时间处于 *Running* 状态；
- **IO密集型** ，程序执行时大部分时间处于 *IO Blocked* 状态；

IO密集型

典型 **IO密集型** 程序时间轴如下，绿色为 *Running* 状态，黄色为 *IO Blocked* 状态：



IO密集型 程序受 GIL 影响相对有限，因为线程在等待 IO 处理时可以让出 GIL 以便其他线程拿到虚拟机执行权：

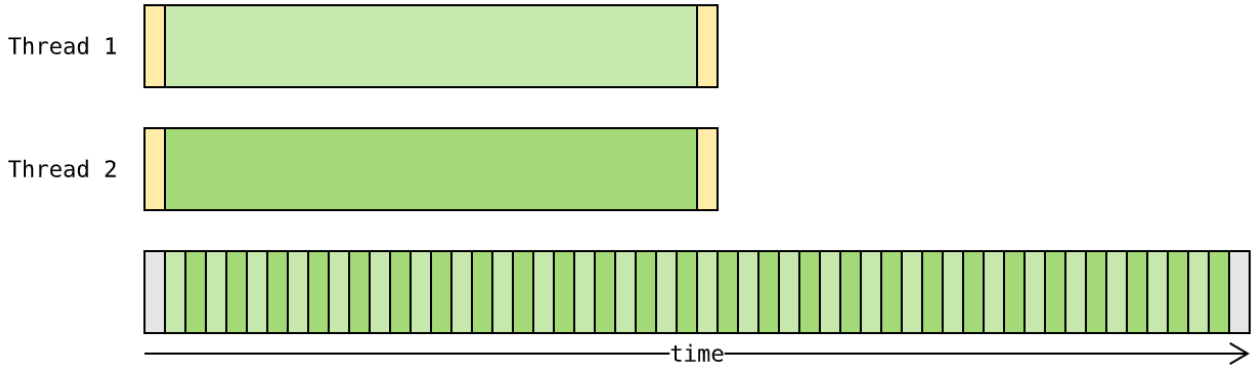


批量下载网页就是一个典型的 **IO密集型** 场景，大部分时间花在等待服务器响应，请求发起以及网页处理所占的时间非常少。因此，一个多线程网络爬虫程序可以极大缩短程序运行时间。

计算密集型

诸如科学计算这样的 **计算密集型** 程序就不一样了，除了开始前读取参数，结束后保存结果，大部分时间都在运算：

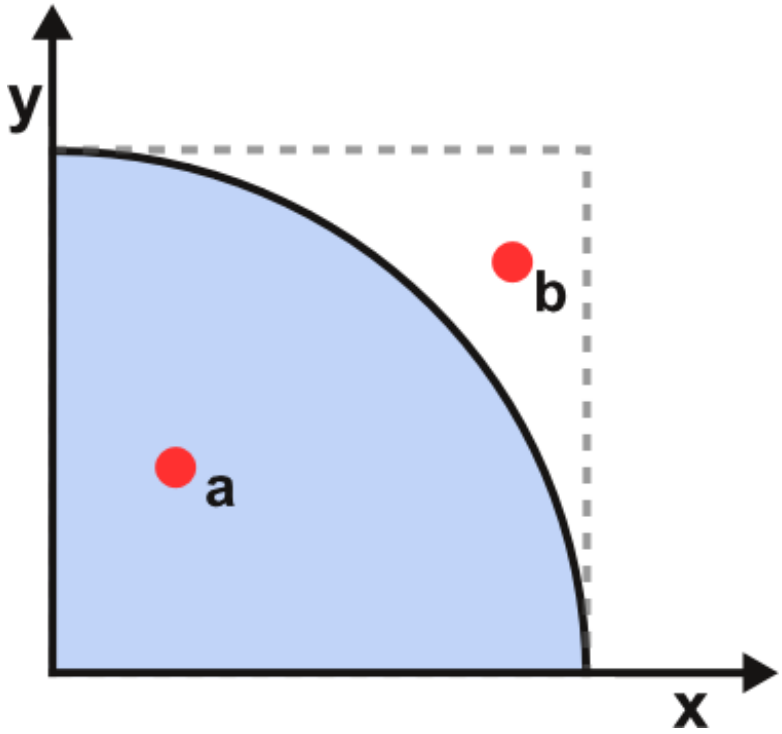
计算密集型 程序受 *GIL* 影响就大了——在 *GIL* 约束下，虚拟机只能交替执行不同的线程：



由此可见，*Python* 线程并不具备多核并行执行能力，不能缩短计算密集型程序的运行时间。

如何提升多核执行能力

我们以 **随机数估算π值** 这个典型的 **计算密集型** 场景为例，探索 *Python* 多核执行能力提升之道：



πππ 值可以通过随机试验进行估算，具体做法是：制作一个边长为 *1* 的正方形，绘出内切扇形，如上图。正方形的面积为 *1* ，内切扇形的面积为 πr24\frac{πr^2 }{4}4πr2，即 π4\frac{π }{4}4π。随机往正方形发射一个点，该点落在扇形内的概率为π4\frac{π }{4}4π。现随机往正方形内发射 nnn 个点，统计其中落在扇形内的点个数 in_sectorsin_sectorsin_sectors，那么 πππ 值可以这样估算：π=4∗in_sectorsnπ = \frac{4 * in_sectors}{n}π=n4∗in_sectors。试验次数越大，估算出来的 πππ 值也就越准确，当然也意味着巨大的计算量。

现在，借助 *random* 模块生成随机数，编写一个试验函数 *sample*，以试验次数 nnn 为参数，返回试验次数 nnn 以及落在扇形内的次数 in_sectorsin_sectorsin_sectors：

```
import random

def sample(n):
    in_sectors = 0

    for _ in range(n):

        x, y = random.random(), random.random()

        if x*x + y*y < 1:
            in_sectors += 1
    return n, in_sectors
```

根据试验结果，*eval_pi* 计算 πππ 的近似值：

```
def eval_pi(n, in_sectors):
    return 4. * in_sectors / n
```

现在，我们以单线程模式执行 *1* 亿次随机试验，*estimate_pi* 执行耗时 *4.59* 秒：

```
def estimate_pi(n):
    return eval_pi(*sample(n))
```

接下来，我们再以多线程模式执行 *1* 亿次随机试验，并与单线程模式进行对比。为此我们设计了多线程版的估算函数 *estimate_pi_with_threads* ，*thread_num* 参数为线程数：

```
from queue import Queue
from threading import Thread

def estimate_pi_with_threads(n, thread_num):

    queue = Queue()


    def thread_routine():


        queue.put(sample(n // thread_num))


    threads = [
        Thread(target=thread_routine)
        for _ in range(thread_num)
    ]


    for thread in threads:
        thread.start()


    total_n, total_in_sectors = 0, 0
    for thread in threads:
        n, in_sectors = queue.get()
        total_n += n
        total_in_sectors += in_sectors


    for thread in threads:
        thread.join()

    return eval_pi(total_n, total_in_sectors)
```

estimate_pi_with_threads **估算函数执行 *1* 亿次试验耗时 *4.63* 秒，多线程对程序没有任何提升。**

那么，是否意味着 *Python* 程序无法充分利用多核 *CPU* 的执行能力呢？

肯定不是。虽然我们没有办法避免 *GIL* 的影响，但是我们可以想法设法绕过它，例如采用 多进程模式 。在 *Python* 程序中，每个进程独立运行一个虚拟机。因此，不同 *Python* 进程可以在多核 *CPU* 上并行运行，不受 *GIL* 限制。

最后，我们再以 **多进程模式** 执行 *1* 亿次随机试验，看执行效率是否如预期有所提升。为此我们设计了多进程版的估算函数 *estimate_pi_with_processes* ，*process_num* 参数为子进程数：

```
from multiprocessing import Process, Queue as ProcessQueue

def estimate_pi_with_processes(n, process_num):

    queue = ProcessQueue()


    def process_routine():


        queue.put(sample(n // process_num))


    processes = [
        Process(target=process_routine)
        for _ in range(process_num)
    ]


    for process in processes:
        process.start()


    total_n, total_in_sectors = 0, 0
    for process in processes:
        n, in_sectors = queue.get()
        total_n += n
        total_in_sectors += in_sectors


    for process in processes:
        process.join()

    return eval_pi(total_n, total_in_sectors)
```

estimate_pi_with_processes 估算函数执行 *1* 亿次试验耗时 *2.56* 秒，几乎比单线程版快了一倍！如果增加子进程数，程序执行速度还可以进一步提升。需要特别注意，超出 *CPU* 核数的进程数则没有任何意义了。

模式	试验函数	耗时(秒)
单线程	estimate_pi(100000000)	4.59
多线程	estimate_pi_with_threads(100000000, 2)	4.63
多进程	estimate_pi_with_processes(100000000, 2)	2.56