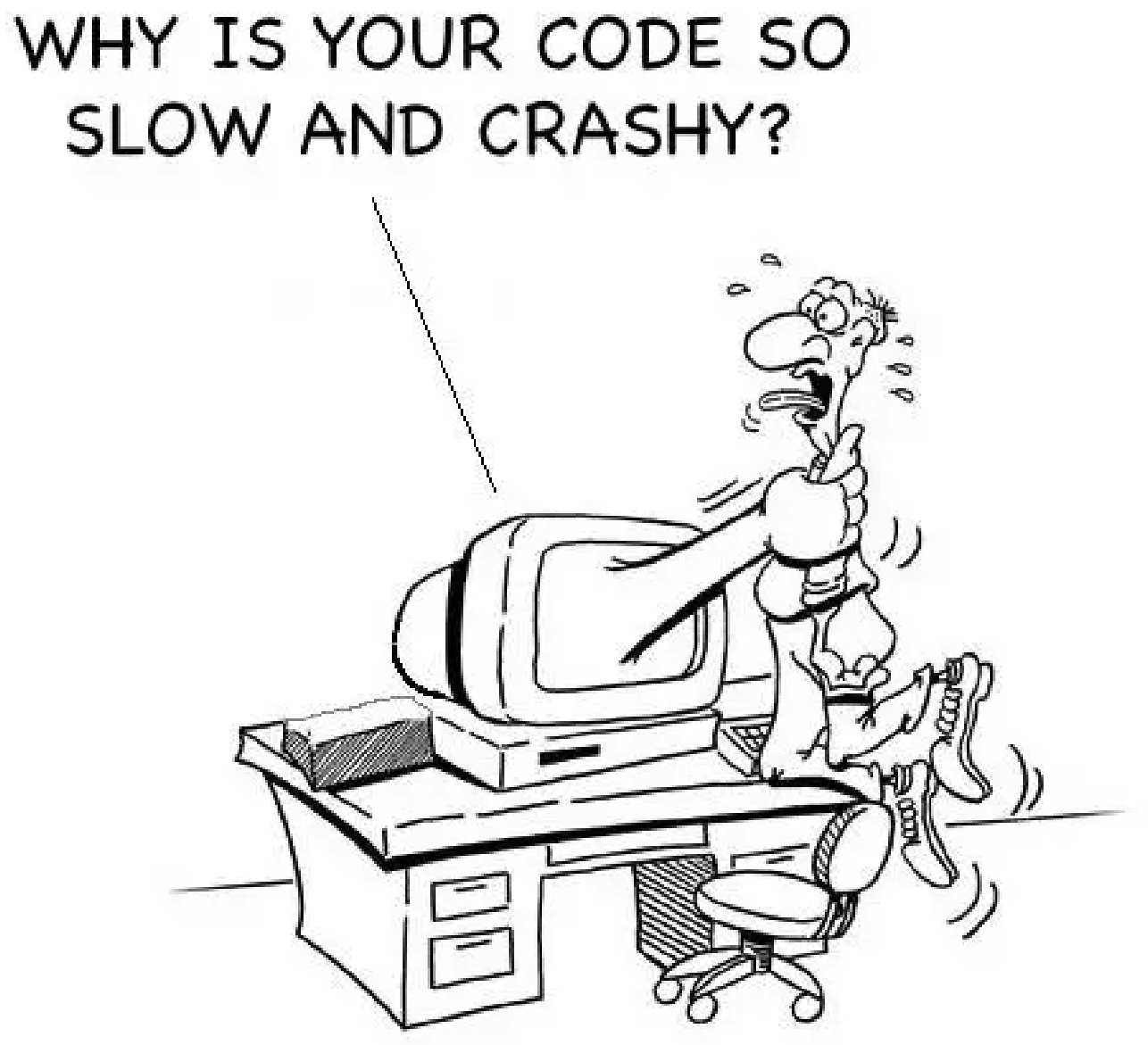
Python，你怎么那么慢？看看并行和并发

Original 2017-03-07 王栋 硅谷程序汪



之前写了一篇Python为什么运行效率低下的问题，一两周过去了，好消息现在自己主导的新项目用Go，学习新东西内心是非常兴奋的。但是既然之前开始研究《为什么Python这么慢》，那么就坚持把这一小系列第二篇写完，讲讲Python的并行和并发。

从一个错误的例子看起

一个常常用来测试程序速度和并发能力的例子就是斐波那契数列，因为他有简单，并且效率低下的写法，容易看出不同程序的区别。

```python
import sys

def fib(k):
    return 1 if k <= 2 else fib(k-1) + fib(k-2)

def runx(x):
    for i in range(x):
        fib(33)

def multi_thread(n):
    import threading
    for i in range(n):
        t = threading.Thread(target=runx, args=(10 / n,))
        t.start()
    for t in threading.enumerate():
        if t is threading.currentThread():
            continue
        t.join()

multi_thread(int(sys.argv[1]))

```

一段简单的程序，多核的运行效率令人发指。

dwang@~$ time python test.py 1

real 0m8.706s

user 0m8.485s

sys 0m0.092s

dwang@~$ time python test.py 2

real 0m11.977s

user 0m10.450s

sys 0m4.661s

dwang@~$ time python test.py 5

real 0m35.898s

user 0m15.996s

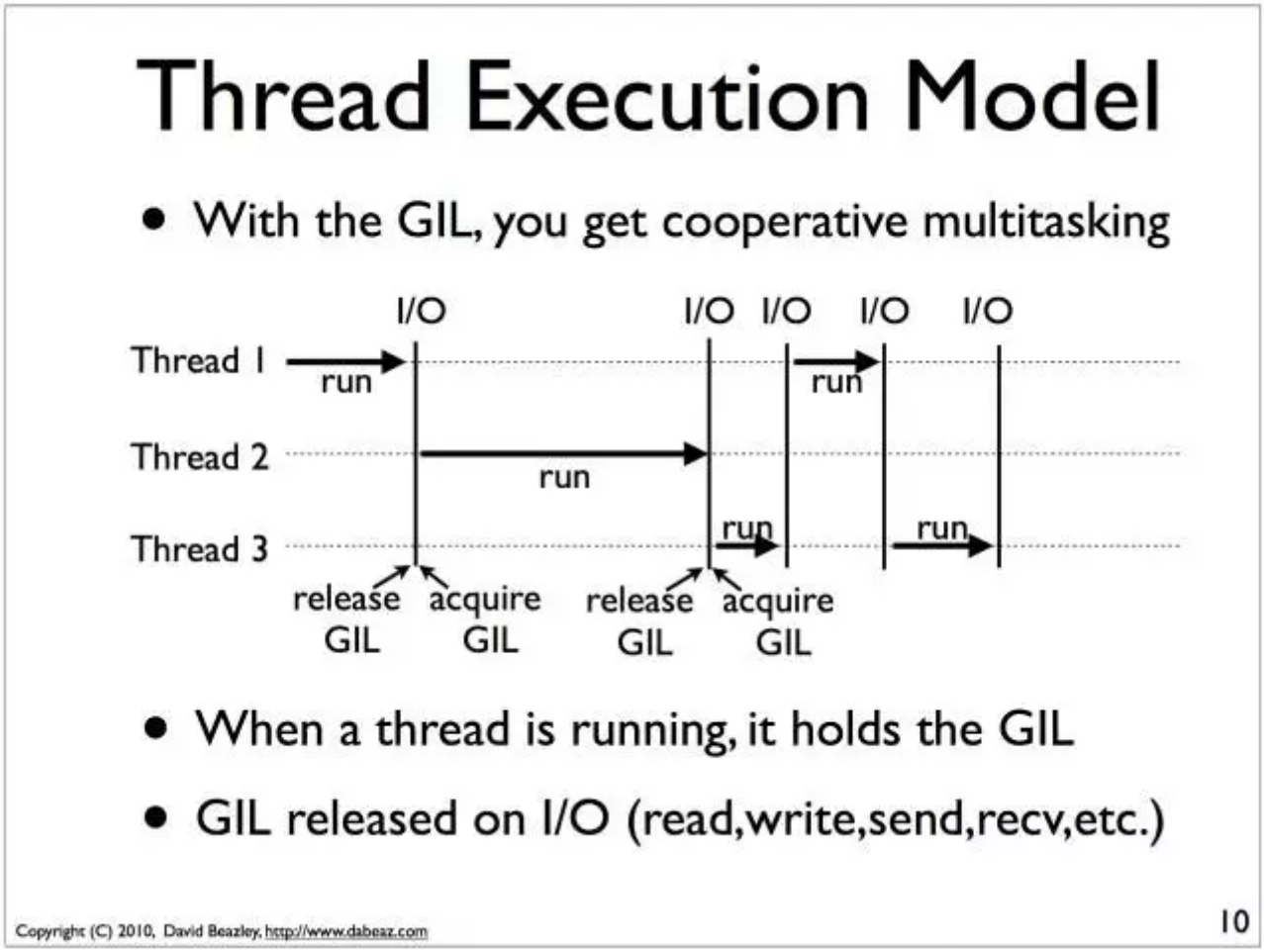
sys 0m6.326s

接触过一段时间Python的同学都知道这样的多线程代码会让程序变得更慢，它并不能利用额外的CPU提高整体计算性能，这是为什么？从前我只是听说Python没有多线程，现在阅读了一些资料后终于明白是Global Interpreter Lock(GIL)，也叫全局解释锁捣的鬼。

白话Python线程与GIL

我不想像大多数博客一样对GIL一笔带过，但是做了一些研究后发现自己也没有深刻的理解Python解释器的工作原理，看了一下Python大神*David Beazley*写的*Understanding the Python GIL*后有了一些浅薄认识。

Python的设计是任何一个进程在任何一个时刻只运行一个线程，不管你的系统内有多少CPU，程序控制是通过GIL实现的，这一设计的目的下面会讲。



从*David Beazley*的slide可以看出，如果一个Python程序里有多个线程，一个程序运行的时候会拿着GIL，当遇到I/O的时候会放开GIL，但是CPU-bound的线程通常不会进行I/O。Python切换线程的一种作法是每100 ticks检查一下，可以通过sys.setcheckinterval()修改这个数值。

综上所述，因为Python线程不能有效利用多核，但是增加了CPU context switch的消耗，所以对于CPU-bound的程序表现很差。更糟糕的是，在多核情况下可能表现会更差，因为系统支持多线程运行但是GIL保证只有一个线程运行，这时候多线程会反复的检查GIL是否被释放，但是拿不到GIL(因为有太多线程竞争)，有可能导致系统发生Trashing现象。

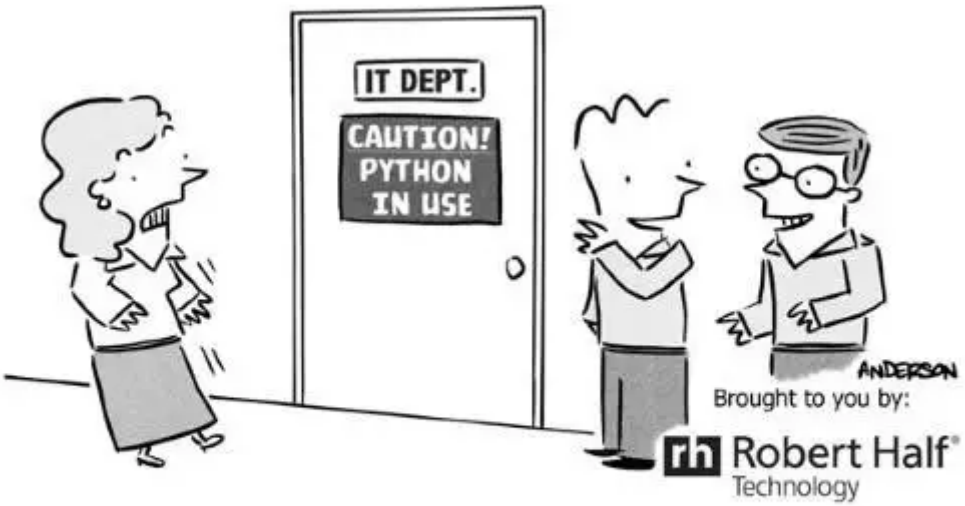
GIL存在的意义

既然GIL有这么多问题，那为什么我们还要用它呢？其实GIL的设计使解释器变得更加容易实现了，可以不用考虑线程的安全性，在任何时刻只有一个线程能够获得Python对象或者嵌入的C/C++ API。

Python 3以及后人其实也做了很多尝试想要去掉GIL，加入更多的细粒度锁，这个设计需要考虑非常多的特殊情况，保证不会影响单线程的效率，不会影响I/O-bound的程序效率，与现有的无数C/C++库兼容等等，具体可以参考下文给出的链接*Efficiently Exploiting Multiple Cores with Python*。

他们考虑了很多，也有人进行过测试，然而对单核版本影响非常大，只有在能有效利用多核CPU的程序下才能看出性能的提升。现实生活中，大家感觉99%的Python程序都是普通的脚本，而追求高效率的代码都用C/C++，Java，Go了，所以去掉GIL的意义不大。

最后，留给我们的办法就是因情况而异，当程序是I/O-bound的时候，比如网络爬虫，或者重定向别的服务，这时候Python解释器的大部分时间都利用在等待函数调用返回上，可以利用多线程库提高服务器并发。



用资源换速度与其他

当然，并不是所有人都会一直对程序效率不闻不问的，从Python 2.6开始引入了multiprocessing(多进程处理库)。这一库的使用方法虽然和线程库相似，但是工作原理完全不同。

多进程处理实际上对每个任务都会生成一个操作系统的进程，并且每一个进程都被单独赋予了Python的解释器和GIL，所以你的程序在实际运行中有多个GIL存在，每个运行

者的线程都会拿到一个GIL，在不同的环境下向前进，自然也可以被分配到不同的处理器上。这一点对于CPU-bound的任务程序非常有帮助。

这一解决方案提高了计算能力，但与此同时也引入了其他的一些问题。因为Python解释器是跟着进程走的，所以新的进程就引入了新的解释器，工作消耗的内存也成倍增长。这点对于那些需要存放很大的全局字典的互联网应用，是致命伤。

另外回到刚才GIL的讨论，其实有不少人投入大量精力打造高吞吐量的网络服务，其中效果非常好的有Stackless Python，Tornado等等，虽然不能像Java一样一秒钟处理上万条消息，但是在4个线程下Tornado能每秒也能处理7000~8000条简单消息，已经很不错了。

结语

最近读了这些资料，自己简单概括就是线程适合I/O-bound的应用，进程适合CPU-bound的应用，但是大部分情况下面对高效率，高并发的需求，Python可能不是最好的语言。

最近开始写Go，对Go产生了很深的兴趣。听说Go 1.8对GC等问题做了很大的优化，但是目前我刚起步还没有感受到，等过段时间新的项目做完有自己的实战经验后再与大家分享。下面是一些关于Python多线程，多进程的干货资料，希望喜欢的人分享并关注我的公众号。

Efficiently Exploiting Multiple Cores with Python： http://python-notes.curiousefficiency.org/en/latest/python3/multicore_python.html
Understanding the Python GIL： http://www.dabeaz.com/GIL/

python 线程，GIL 和 ctypes： http://zhuoqiang.me/python-thread-gil-and-ctypes.html