## GIL全局解释器锁

CPython 在解释器进程级别有一把锁，叫做GIL，即全局解释器锁。
GIL 保证CPython进程中，只有一个线程执行字节码。甚至是在多核CPU的情况下，也只允许同时只能有一个CPU核心上运行该进程的一个线程。

CPython中

- IO密集型，某个线程阻塞，GIL会释放，就会调度其他就绪线程
- CPU密集型，当前线程可能会连续的获得GIL，导致其它线程几乎无法使用CPU
- 在CPython中由于有GIL存在，IO密集型，使用多线程较为合算；CPU密集型，使用多进程，要绕开GIL

新版CPython正在努力优化GIL的问题，但不是移除。
如果在意多线程的效率问题，请绕行，选择其它语言erlang、Go等。

> Python中绝大多数内置数据结构的**读、写操作**都是原子操作。
> 由于GIL的存在，Python的内置数据类型在多线程编程的时候就变成了安全的了，但是实际上它们本身 **不是线程安全类型。**

保留GIL的原因：
GvR坚持的简单哲学，对于初学者门槛低，不需要高深的系统知识也能安全、简单的使用Python。
而且移除GIL，会降低CPython单线程的执行效率。

测试下面2个程序，请问下面的程序是计算密集型还是IO密集型?

```python
import logging
import datetime

logging.basicConfig(level=logging.INFO, format="%(thread)s %(message)s")
start = datetime.datetime.now()

# 计算
def calc():
    sum = 0
    for _ in range(1000000000): # 10亿
        sum += 1

calc()
calc()
calc()
calc()

delta = (datetime.datetime.now() - start).total_seconds()
logging.info(delta)
```

```python

import threading
import logging
import datetime

logging.basicConfig(level=logging.INFO, format="%(thread)s %(message)s")
start = datetime.datetime.now()

```

```
# 计算
def calc():
    sum = 0
    for _ in range(1000000000): # 10亿
        sum += 1

t1 = threading.Thread(target=calc)
t2 = threading.Thread(target=calc)
t3 = threading.Thread(target=calc)
t4 = threading.Thread(target=calc)

t1.start()
t2.start()
t3.start()
t4.start()

t1.join()
t2.join()
t3.join()
t4.join()

delta = (datetime.datetime.now() - start).total_seconds()
logging.info(delta)
```

注意，不要在代码中出现print等访问IO的语句。访问IO，线程阻塞，会释放GIL锁，其他线程被调度。

程序1是单线程程序，所有calc()依次执行，根本就不是并发。在主线程内，函数串行执行。
程序2是多线程程序，calc()执行在不同的线程中，但是由于GIL的存在，线程的执行变成了假并发。但是这些线程可以被调度到不同的CPU核心上执行，只不过GIL让同一时间该进程只有一个线程被执行。

从两段程序测试的结果来看，CPython中多线程根本没有任何优势，和一个线程执行时间相当。因为GIL的存在，尤其是像上面的计算密集型程序，和单线程串行效果相当。这样，实际上就没有用上CPU多核心的优势。