12.9 Python的全局锁问题¶

问题¶

你已经听说过全局解释器锁GIL，担心它会影响到多线程程序的执行性能。

解决方案¶

尽管Python完全支持多线程编程， 但是解释器的C语言实现部分在完全并行执行时并不是线程安全的。 实际上，解释器被一个全局解释器锁保护着，它确保任何时候都只有一个Python线程执行。 GIL最大的问题就是Python的多线程程序并不能利用多核CPU的优势 （比如一个使用了多个线程的计算密集型程序只会在一个单CPU上面运行）。

在讨论普通的GIL之前，有一点要强调的是GIL只会影响到那些严重依赖CPU的程序（比如计算型的）。 如果你的程序大部分只会涉及到I/O，比如网络交互，那么使用多线程就很合适， 因为它们大部分时间都在等待。实际上，你完全可以放心的创建几千个Python线程， 现代操作系统运行这么多线程没有任何压力，没啥可担心的。

而对于依赖CPU的程序，你需要弄清楚执行的计算的特点。 例如，优化底层算法要比使用多线程运行快得多。 类似的，由于Python是解释执行的，如果你将那些性能瓶颈代码移到一个C语言扩展模块中， 速度也会提升的很快。如果你要操作数组，那么使用NumPy这样的扩展会非常的高效。 最后，你还可以考虑下其他可选实现方案，比如PyPy，它通过一个JIT编译器来优化执行效率 （不过在写这本书的时候它还不能支持Python 3）。

还有一点要注意的是，线程不是专门用来优化性能的。 一个CPU依赖型程序可能会使用线程来管理一个图形用户界面、一个网络连接或其他服务。 这时候，GIL会产生一些问题，因为如果一个线程长期持有GIL的话会导致其他非CPU型线程一直等待。 事实上，一个写的不好的C语言扩展会导致这个问题更加严重， 尽管代码的计算部分会比之前运行的更快些。

说了这么多，现在想说的是我们有两种策略来解决GIL的缺点。 首先，如果你完全工作于Python环境中，你可以使用 `multiprocessing` 模块来创建一个进程池， 并像协同处理器一样的使用它。例如，假如你有如下的线程代码：

```
# Performs a large calculation (CPU bound)
def some_work(args):
    ...
    return result

# A thread that calls the above function
def some_thread():
    while True:
        ...
        r = some_work(args)
    ...
```

修改代码，使用进程池：

```
# Processing pool (see below for initiazation)
pool = None

# Performs a large calculation (CPU bound)
def some_work(args):
    ...
    return result

# A thread that calls the above function
def some_thread():
    while True:
        ...
        r = pool.apply(some_work, (args))
        ...

# Initiaze the pool
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()
```

这个通过使用一个技巧利用进程池解决了GIL的问题。 当一个线程想要执行CPU密集型工作时，会将任务发给进程池。 然后进程池会在另外一个进程中启动一个单独的Python解释器来工作。 当线程等待结果的时候会释放GIL。 并且，由于计算任务在单独解释器中执行，那么就不会受限于GIL了。 在一个多核系统上面，你会发现这个技术可以让你很好的利用多CPU的优势。

另外一个解决GIL的策略是使用C扩展编程技术。 主要思想是将计算密集型任务转移给C，跟Python独立，在工作的时候在C代码中释放GIL。 这可以通过在C代码中插入下面这样的特殊宏来完成：

```
#include "Python.h"
...

PyObject *pyfunc(PyObject *self, PyObject *args) {
   ...
   Py_BEGIN_ALLOW_THREADS
   // Threaded C code
   ...
   Py_END_ALLOW_THREADS
   ...
}
```

如果你使用其他工具访问C语言，比如对于Cython的ctypes库，你不需要做任何事。 例如，ctypes在调用C时会自动释放GIL。

讨论¶

许多程序员在面对线程性能问题的时候，马上就会怪罪GIL，什么都是它的问题。 其实这样子太不厚道也太天真了点。 作为一个真实的例子，在多线程的网络编程中神秘的 `stalls` 可能是因为其他原因比如一个DNS查找延时，而跟GIL毫无关系。 最后你真的需要先去搞懂你的代码是否真的被GIL影响到。 同时还要明白GIL大部分都应该只关注CPU的处理而不是I/O.

如果你准备使用一个处理器池，注意的是这样做涉及到数据序列化和在不同Python解释器通信。 被执行的操作需要放在一个通过def语句定义的Python函数中，不能是lambda、闭包可调用实例等， 并且函数参数和返回值必须要兼容pickle。 同样，要执行的任务量必须足够大以弥补额外的通信开销。

另外一个难点是当混合使用线程和进程池的时候会让你很头疼。 如果你要同时使用两者，最好在程序启动时，创建任何线程之前先创建一个单例的进程池。 然后线程使用同样的进程池来进行它们的计算密集型工作。

C扩展最重要的特征是它们和Python解释器是保持独立的。 也就是说，如果你准备将Python中的任务分配到C中去执行， 你需要确保C代码的操作跟Python保持独立， 这就意味着不要使用Python数据结构以及不要调用Python的C API。 另外一个就是你要确保C扩展所做的工作是足够的，值得你这样做。 也就是说C扩展担负起了大量的计算任务，而不是少数几个计算。

这些解决GIL的方案并不能适用于所有问题。 例如，某些类型的应用程序如果被分解为多个进程处理的话并不能很好的工作， 也不能将它的部分代码改成C语言执行。 对于这些应用程序，你就要自己需求解决方案了（比如多进程访问共享内存区，多解析器运行于同一个进程等）。 或者，你还可以考虑下其他的解释器实现，比如PyPy。

了解更多关于在C扩展中释放GIL，请参考15.7和15.10小节。